OPTIMIZATION OF TRANSITION STATE STRUCTURES USING GENETIC ALGORITHMS

SHARENE D. BUNGAY

Optimization of Transition State Structures Using Genetic Algorithms

by

© Sharene D. Bungay

B.Sc. (Memorial University of Newfoundland, St. John's, Canada) 1998

A thesis submitted to the
School of Graduate Studies
in partial fulfilment of the
requirements for the degree of
Master of Science

Departments of Chemistry, Mathematics and Statistics, and Computer Science
Memorial University of Newfoundland

(September, 2000)

St. John's Newfoundland

Abstract

Geometry optimization has long been an active research area in theoretical chemistry. Many algorithms currently exist for the optimization of minima (reactants, intermediates, and products) on a potential energy surface. However, determination of transition state structures (first order saddle points) has been an ongoing problem. The computational technique of genetic algorithms has recently been applied to optimization problems in many disciplines. Genetic algorithms are a type of evolutionary computing in which a population of individuals, whose genes collectively encode candidate solutions to the problem being solved, evolve toward a desired objective. Each generation is biased towards producing individuals which closely resemble the known desired features of the optimum. This thesis contains a discussion of existing techniques for geometry optimization, a description of genetic algorithms, and an explanation of how the genetic algorithm technique was applied to transition state optimization and incorporated into the existing *ab initio* package Mungauss. Results from optimizing mathematical functions, demonstrating the effectiveness of the genetic algorithm implemented to optimize first order saddle points, are presented, followed by results from the optimization of standard chemical structures used for the testing of transition state optimization methods. Finally, some ideas for future method modifications to increase the efficiency of the genetic algorithm implementation used are discussed.

Acknowledgements

I would like to express my gratitude to the many people that have helped me in the preparation of this thesis. First I would like to thank my co-supervisors, Dr. R. A. Poirier and Dr. R. Charron for giving me the opportunity to begin this project and providing constant guidance throughout my research.

I would also like to thank my office mates, Michelle Shaw, Tammy Gosse, and James Xidos, for their assistance during my leap into quantum chemistry, as well as many stress relieving conversations.

I would like to express utmost appreciation to Sam Bromley for constant encouragement, meticulous proofreading, and endless technical help.

I am very grateful to my parents, who have provided encouragement and supported me in all my decisions along the way.

Many thanks is expressed to the Natural Sciences and Engineering Research Council (NSERC) and Memorial University of Newfoundland for financial support. Also. computational facilities were provided by the Departments of Mathematics, and Computing and Communications at Memorial, for which I am thankful.

To my grandmother,

Annie Catherine Tiller

Contents

List of Tables

List of Figures

Chapter 1

Optimization Background

"For the things of this world cannot be made known without a knowledge of mathematics."

-Roger Bacon

1.1 Introduction

The most predominant problem in theoretical chemistry has been, for quite some time, the nonlinear, unconstrained geometry optimization of molecular structures. The ever increasing computational power and the ability to calculate numerical derivatives gave rise to many methods and algorithms for optimization. The structures being optimized represent stationary points on a potential energy surface (PES), and hence consists of minima, maxima, and saddle points of varying order, whose energies are described as a function of geometric parameters such as bond lengths, angles, and dihedral angles (torsions). The stationary points of most interest include minima, representing reactants, products, and intermediates in a chemical reaction, as well

as first-order saddle points corresponding to transition state structures. Higher order saddle points are of no chemical interest. Many algorithms currently exist for the optimization of minima. The optimization of transition state structures however, has presented much difficulty, and continues to be a major area of research. A computational approach for optimizing such structures is required since they have a fleeting existence experimentally and are difficult (and sometimes impossible) to isolate. Transition state structures are required for understanding reaction mechanisms, and in turn, activation energies and reaction rates.

Since the actual potential energy surface is not available, let us conceptualize a potential energy surface to illustrate the inhibitive factors with respect to transition state structures.

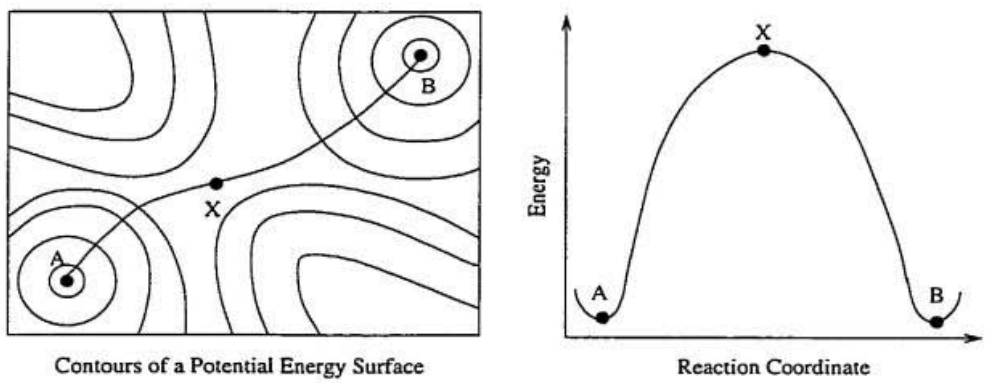


Figure 1.1: A contour plot of a conceptualized potential energy surface (left) and its reaction coordinate diagram (right). The reaction path includes points A (reactants), X (transition state structure), and B (products).

A contour plot of a possible PES is shown in Figure 1.1. The reaction coordinate diagram represents the path taken across the potential energy surface and is indicative

of the reaction mechanism which takes place along the lowest energy path connecting reactants and products. As minima, A and B can be characterized as having a zero gradient (first derivative vector) and a Hessian (second derivative matrix) which is positive definite (all positive eigenvalues). However, as a first-order saddle point, the transition state structure, X, is a maximum along the reaction path, and a minimum in all other directions. Like A and B, X has a zero gradient, however, the Hessian matrix has one, and only one, negative eigenvalue. For the purposes of optimization problems this poses a great difficulty, and there is currently no method which can *guarantee* convergence to a transition state structure.

1.2 Mathematical Representation

Let the PES be given by some unknown function, $f(\vec{x}) : \mathbb{R}^n \mapsto \mathbb{R}$, where the components of $\vec{x}$ are the geometric parameters that characterize a geometry of the system. Finding minima on this surface is equivalent to the optimization problem

$$\min(f(\vec{x})), \quad f : \mathbb{R}^n \mapsto \mathbb{R}. \tag{1.1}$$

Equivalently, the objective is to find $\vec{x}^*$ such that

$$f(\vec{x}^*) \leq f(\vec{x}), \quad \forall \vec{x} \in \{\vec{x} \in \mathbb{R}^n, \|\vec{x} - \vec{x}^*\| < \epsilon\} \tag{1.2}$$

for some ϵ. Given a starting point $\vec{x}_0$, the direction of maximum decrease of the objective function, $f(\vec{x})$, is the negative of the gradient vector, $-\nabla f(\vec{x}_0)$. A possible search procedure is to use the direction of $-\nabla f(\vec{x}_0)$ to define a path towards a minimum. This leads to the steepest descent method:

Algorithm 1 (Steepest Descent)

Given $\vec{x}_k$ as a point on the surface,

1. *compute the direction vector $\vec{d}_k = -\nabla f(\vec{x}_k)$.*
2. *determine the positive real number α_k such that $f(\vec{x}_k + \alpha_k \vec{d}_k)$ is minimized, using a line search. The value of α_k corresponds to the step size.*
3. *step: $\vec{x}_{k+1} = \vec{x}_k + \alpha_k \vec{d}_k$.*

This method tends to work well when $\vec{x}_k$ is far removed from a stationary point but does not behave well as it approaches an optimum, as it often steps past the region of the optimum.

Alternatively, the objective function can be expanded as a Taylor series about a point $\vec{x}_0$ on the surface,

$$f(\vec{x}_0 + \Delta\vec{x}) = f(\vec{x}_0) + \vec{g}^{\,\mathrm{T}}\Delta\vec{x} + \frac{1}{2}(\Delta\vec{x}^{\,\mathrm{T}}\mathrm{H}\Delta\vec{x}) + \cdots \tag{1.3}$$

where $\Delta\vec{x}$ is the step taken on the surface, $\vec{g}$ is the gradient vector $\nabla f(\vec{x})$ (where $g_i = \frac{\partial f}{\partial x_i}$), H is the Hessian matrix $\nabla^2 f(\vec{x})$, (where $\mathrm{H}_{i,j} = \frac{\partial^2 f}{\partial x_i \partial x_j}$), and T is the

standard transpose operation. Taking a quadratic approximation to the surface near $\vec{x}_0$ we can truncate the series, to give,

$$f(\vec{x}_0 + \Delta\vec{x}) \approx f(\vec{x}_0) + \vec{g}^{\,\mathrm{T}}\Delta\vec{x} + \frac{1}{2}(\Delta\vec{x}^{\,\mathrm{T}}\mathrm{H}\Delta\vec{x}). \tag{1.4}$$

For a stationary point, $\nabla f(\vec{x}^{\,\star}) = \vec{0}$, which gives,

$$\vec{g} + \mathrm{H}\cdot(\Delta\vec{x}) = \vec{0}. \tag{1.5}$$

This results in a step,

$$\Delta\vec{x} = -\mathrm{H}^{-1}\cdot\vec{g} \tag{1.6}$$

toward a stationary point on the potential energy surface, where it is assumed that H is invertible. This is the basis of the well known optimization technique *Newton's method*, which can be implemented using the following iterative algorithm:

Algorithm 2 (Newton's method)

Given $\vec{x}_k$ as a point on the surface,

1. *compute $\vec{g}(\vec{x}_k)$ and $\mathrm{H}(\vec{x}_k)$*
2. *solve $\mathrm{H}\cdot(\Delta\vec{x}) = -\vec{g}$ for $\Delta\vec{x}$*
3. *step: $\vec{x}_{k+1} = \vec{x}_k + \Delta\vec{x}$ or $\vec{x}_{k+1} = \vec{x}_k + \alpha_k\Delta\vec{x}$, where α_k is calculated such that $f(\vec{x}_k + \alpha_k\Delta\vec{x})$ is minimized by using a line search method.*

This algorithm will generate a series of steps toward a stationary point with quadratic convergence provided the initial guess, $\vec{x}_0$ is within the quadratic region of the optimum. Outside of this region, Newton's method converges very slowly. A major disadvantage of the method is that both first and second derivatives are required at each iteration. These calculations can be computationally expensive depending on the number of variables in the system.

1.3 Existing Methods

Newton's method provides the basis for many existing optimization methods. A number of optimization methods employ a modification of Newton's method focusing on approximating the Hessian matrix and updating it during each iteration to avoid the expense of recalculation. These methods are called *quasi*-Newton, or *variable metric methods*. The current gradient and parameter information is used to form the approximate Hessian, as in the Murtagh-Sargent update [1],

$$\mathrm{H}' = \mathrm{H} + \frac{(\Delta\vec{g} - \mathrm{H}\Delta\vec{x})(\Delta\vec{g} - \mathrm{H}\Delta\vec{x})^{\mathrm{T}}}{(\Delta\vec{g} - \mathrm{H}\Delta\vec{x})^{\mathrm{T}}\Delta\vec{x}}, \tag{1.7}$$

or the symmetric Powell update [2],

$$\mathrm{H}' = \mathrm{H} + \frac{(\Delta\vec{g} - \mathrm{H}\Delta\vec{x})\Delta\vec{x}^{\,\mathrm{T}} + \Delta\vec{x}(\Delta\vec{g} - \mathrm{H}\Delta\vec{x})^{\,\mathrm{T}}}{\Delta\vec{x}^{\,\mathrm{T}}\Delta\vec{x}} - \frac{[(\Delta\vec{g} - \mathrm{H}\Delta\vec{x})^{\,\mathrm{T}}\Delta\vec{x}]\Delta\vec{x}\Delta\vec{x}^{\,\mathrm{T}}}{(\Delta\vec{x}^{\,\mathrm{T}}\Delta\vec{x})^2}, \tag{1.8}$$

where H′ is the approximate Hessian. The Broyden family of updates given by,

$$H' = H + \frac{\Delta\vec{g}\Delta\vec{g}^{T}}{\Delta\vec{g}^{T}\Delta\vec{x}} - \frac{H\Delta\vec{x}\Delta\vec{x}^{T}H}{\Delta\vec{x}^{T}H\Delta\vec{x}} + \eta(\Delta\vec{x}^{T}H\Delta\vec{x})\vec{w}\vec{w}^{T} \tag{1.9}$$

where

$$\vec{w} = \frac{\Delta\vec{g}}{\Delta\vec{g}^{T}\Delta\vec{x}} - \frac{H\Delta\vec{x}}{\Delta\vec{x}^{T}H\Delta\vec{x}} \tag{1.10}$$

includes the Davidon - Fletcher - Powell update [3, 4] (DFP) when $\eta = 0$. and the Broyden - Fletcher - Goldfarb - Shanno update [5, 6, 7, 8] (BFGS) when $\eta = 1$. The optimally conditioned (OC) method by Davidon [9] chooses η such as to minimize the condition number of the Hessian update, that is, the ratio of the largest to smallest eigenvalues.

One of the problems associated with using a quadratic approximation of the PES, is the sensitivity of convergence to the choice of step size. For example, even if a calculated direction is correct, the step size used can result in slow convergence or stepping beyond the region of interest. To circumvent this problem some methods focus on mechanisms of step size control. One such method is the Trust Radius Method [10] which restricts the step to be smaller than a defined trust radius, τ. This yields a step,

$$\Delta\vec{x} = -(H - \nu I)^{-1}\Delta\vec{g}, \tag{1.11}$$

where ν is adjusted to satisfy the trust radius condition. The value of τ can be changed dynamically during the optimization as the local surface changes. Another such method is the Rational Function Optimization (RFO) method [11, 12] which minimizes the function approximation,

$$f(\vec{x}_0 + \Delta\vec{x}) \approx f(\vec{x}_0) + \frac{\vec{g}^{\,\mathrm{T}}\Delta\vec{x} + \frac{1}{2}(\Delta\vec{x}^{\,\mathrm{T}}\mathrm{H}\Delta\vec{x})}{(1 + \alpha\Delta\vec{x}^{\,\mathrm{T}}\Delta\vec{x})} \tag{1.12}$$

where α is chosen to decrease the objective function while restricting the step to less than the trust radius τ.

Both the trust radius method and the RFO method guarantee a decrease in the objective function, and will step toward a minimum regardless of the initial structure of the Hessian. In contrast, the direction of the step taken with Newton's method is dependent on the number of negative eigenvalues of the Hessian. For example, if the Hessian has one negative eigenvalue the step will be in the direction of a transition state rather than a minimum. Some of the Hessian updating algorithms can prevent this however. For example, OC, DFP, and BFGS were strictly formulated to locate a minimum since the Hessian is forced to remain positive definite during the optimization. Since none of the methods mentioned were specifically designed for locating transition state structures, and since many are forced toward minima, the optimization of transition state structures is a problem that remains open.

An alternative technique to quasi-Newton methods is Direct Inversion in the It-

erative Subspace (DIIS) [13] which performs geometry optimization by taking a step which is a linear combination of $\vec{x}_k$ and $\vec{x}_{k-1}$ such as to minimize the norm of an error vector. However, for transition state structures DIIS presents the problem of being an interpolation like scheme, which will be somewhat dependent on the placement of x_k's. If the current and previous geometries are consistently on the same side of the transition state, interpolation will result in never converging to a transition state structure. The current use of DIIS as a transition state method will be discussed in Section 1.4.1.

1.4 Methods for Transition State Structures

Although many of the methods already mentioned are able to find a transition state structure, most are not biased toward these first-order saddle points. Quasi-Newton optimization will require that the initial guess lie very close to the final geometry in order to converge on a transition state, and also that the initial Hessian have the required single negative eigenvalue. Techniques to move into the region around the transition state include Linear Synchronous Transit (LST), and Quadratic Synchronous Transit (QST). LST searches for a maximum along a line connecting reactants and products. QST goes a step further by searching for a maximum along a parabola connecting reactants and products while searching for a minimum in all other orthogonal directions. Once the geometry falls in the region of the transition state, the quasi-Newton methods will give satisfactory convergence.

If one is aware of a geometric parameter whose change dominates the reaction, a technique known as coordinate driving can be used to move toward the transition state along this direction while minimizing with respect to all other parameters. Methods such as eigenvector following or "walking up valleys" [14] can locate transition state structures by stepping toward a maximum in the direction corresponding to the lowest eigenvalue while minimizing along all other directions.

Another way to locate transition state structures is to minimize the gradient norm. However, this characteristic is not unique to first order saddle points and a gradient norm approach will not selectively converge to transition state structures. In addition, points other than minima, maxima, and saddle points can satisfy this criteria.

A modification of the trust region method, trust region image minimization [15] (TRIM) performs a minimization of an image function formed by reversing the sign of the lowest eigenmode. Hence, the saddle points of the original function are minima of the image function, and can be obtained via minimization using the trust region method.

Various combinations of the previously mentioned Hessian update formulas have also been used to optimize transition state structures. The Bofill update [16] is a combination of Murtagh-Sargent and symmetric Powell updates,

$$\mathrm{H}' = \mathrm{H} + [\phi \Delta \mathrm{H}_{SP} + (1 - \phi) \Delta \mathrm{H}_{MS}] \tag{1.13}$$

where

$$\phi = 1 - \frac{[\Delta\vec{x}^{\,\mathrm{T}}(\Delta\vec{g} - \mathrm{H}\Delta\vec{x})]^{\,2}}{\|\Delta\vec{x}\|^2 \cdot \|\Delta\vec{g} - \mathrm{H}\Delta\vec{x}\|^2}. \tag{1.14}$$

A modification to the BFGS Hessian update formula proposed by Anglada, et al. [17], was formulated specifically for transition state structures,

$$\begin{aligned} \mathrm{H}'_{\mathrm{TS-BFGS}} = \mathrm{H} &+ \frac{(\Delta\vec{g} - \mathrm{H}\Delta\vec{x})[(1-\phi)|\mathrm{H}|\Delta\vec{x} + \phi(\Delta\vec{x}^{\,\mathrm{T}}\Delta\vec{g})\Delta\vec{g}]^{\,\mathrm{T}}}{\Delta\vec{x}^{\,\mathrm{T}}|\mathrm{H}|\Delta\vec{x} + [(\Delta\vec{x}^{\,\mathrm{T}}\Delta\vec{g})^2 - \Delta\vec{x}^{\,\mathrm{T}}|\mathrm{H}|\Delta\vec{x}]\phi} \\ &+ \frac{[(1-\phi)|\mathrm{H}|\Delta\vec{x} + \phi(\Delta\vec{x}^{\,\mathrm{T}}\Delta\vec{g})\Delta\vec{g}](\Delta\vec{g} - \mathrm{H}\Delta\vec{x})^{\,\mathrm{T}}}{\Delta\vec{x}^{\,\mathrm{T}}|\mathrm{H}|\Delta\vec{x} + [(\Delta\vec{x}^{\,\mathrm{T}}\Delta\vec{g})^2 - \Delta\vec{x}^{\,\mathrm{T}}|\mathrm{H}|\Delta\vec{x}]\phi} \\ &- \frac{\Delta\vec{x}^{\,\mathrm{T}}(\Delta\vec{g} - \mathrm{H}\Delta\vec{x})}{\{\Delta\vec{x}^{\,\mathrm{T}}|\mathrm{H}|\Delta\vec{x} + [(\Delta\vec{x}^{\,\mathrm{T}}\Delta\vec{g})^2 - \Delta\vec{x}^{\,\mathrm{T}}|\mathrm{H}|\Delta\vec{x}]\phi\}^2} \\ &\times [(1-\phi)|\mathrm{H}|\Delta\vec{x} + \phi(\Delta\vec{x}^{\,\mathrm{T}}|\mathrm{H}|\Delta\vec{g})\Delta\vec{g}] \\ &\times [(1-\phi)|\mathrm{H}|\Delta\vec{x} + \phi(\Delta\vec{x}^{\,\mathrm{T}}\Delta\vec{g})\Delta\vec{g}]^{\,\mathrm{T}} \end{aligned} \tag{1.15}$$

where $|\mathrm{H}|$ is the Hessian matrix made positive definite. This formula is known as the TS-BFGS update and is based on the standard rank one updating procedure. Investigation into this updating formula revealed that the steps taken with the approximate Hessian do not lead one to a stationary point. This behaviour was determined to be due to an error in the units of the following equation,

$$\mathrm{M} = (1-\phi)|\mathrm{H}| + \phi\Delta\vec{g}\Delta\vec{g}^{\,\mathrm{T}} \tag{1.16}$$

used to formulate the Hessian update, where the two terms have different units and

therefore cannot be added to yield a physically meaningful result.

Two of the commonly used transition state methods that have been implemented in Mungauss [18] are DIIS, and VA. These methods are discussed further in the following sections.

1.4.1 Direct Inversion in the Iterative Subspace (DIIS)

Like Newton-Raphson methods, DIIS was designed for near quadratic potential energy surfaces. Denoting the energy surface as $E(\vec{q})$ where $\vec{q}$ is a vector of molecular parameters, take the final solution $\vec{q}^*$ to be a linear combination of the $\vec{q}$ vectors from the m previous iterations,

$$\vec{q}^* = \sum_{i=1}^{m} c_i \vec{q}_i. \tag{1.17}$$

This is analogous to taking each $\vec{q}_i$ as a perturbation from the desired solution.

$$\vec{q}_i = \vec{q}^* + \vec{e}_i, \tag{1.18}$$

and requiring that,

$$\sum_{i=1}^{m} c_i \vec{e}_i = \vec{0}, \tag{1.19}$$

and,

$$\sum_{i=1}^{m} c_i = 1. \tag{1.20}$$

Following this formulation the error vectors $\vec{e}_i, i = 1, \ldots, m$ are unknown. Assuming a nearly quadratic energy surface, we can take,

$$\vec{e}_i = -\mathrm{H}^{-1}\vec{g}_i, \tag{1.21}$$

where the gradient vector $\vec{g}_i$ corresponds to the parameter vector $\vec{q}_i$ at iteration i, and H is an approximate Hessian. Minimization of $\sum c_i\vec{e}_i$ in the least squares sense (see Equation 1.19) and satisfying Equation 1.20 produces a system of equations that can be expressed in matrix form as,

$$\begin{bmatrix} B_{11} & B_{12} & \cdots & B_{1m} & 1 \\ B_{21} & B_{22} & & & 1 \\ \vdots & & \ddots & & \vdots \\ B_{n1} & & & B_{nm} & 1 \\ 1 & 1 & \cdots & 1 & 0 \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_m \\ -\lambda \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ 1 \end{bmatrix} \tag{1.22}$$

where,

$$B_{ij} = < \vec{e}_i | \vec{e}_j > \tag{1.23}$$

and λ is a Lagrange multiplier. Solution of this system gives values for the c_i's which are then used to form an intermediate interpolated parameter vector,

$$\vec{q}_{m+1} = \sum_{i=1}^{m} c_i \vec{q}_i, \tag{1.24}$$

as well as an interpolated gradient vector,

$$\vec{g}_{m+1} = \sum_{i=1}^{m} c_i \vec{g}_i. \tag{1.25}$$

Convergence is checked at this point and another iteration is started at Equation 1.22 with the new parameters added.

An iteration scheme for this method is as follows:

Algorithm 3 (DIIS)

1. *Starting with an approximate parameter set, $\vec{q}_0$, and an approximate H_0^{-1}. perform Newton-Raphson iterations until the quadratic region is reached.*

2. *Store the parameter vectors at each iteration after this point. Solve Equation 1.22 with m=2. Stop at this point if the error vector is sufficiently small.*

3. *Compute the interpolated parameter vector, step using Newton-Raphson and test convergence. If not converged, add the new vectors to the list and perform a new iteration. If converged, stop.*

Although DIIS will often optimize transition state structures, problems inherent

in the method can prevent convergence. This inhibition is due to the interpolation feature of the method, which can cause iterations to become "stuck" on one side of the transition state.

1.4.2 VA - A Least Squares Approach

The VA method is based on an algorithm developed by Powell [19], and is a hybrid method incorporating the methods of steepest descent and Newton's method. This method works relatively well for transition state structures but is not designed with transition state optimization as its sole purpose. Beginning with a system of equations,

$$G_i = g_i(\vec{x}) = 0, i = 0 \ldots \tag{1.26}$$

the derivative of these equations with respect x_i gives the Jacobian matrix J_{ij}. The truncated Taylor expansion gives,

$$G(\vec{x}^*) = G(\vec{x}) + J \cdot (\Delta\vec{x}) = 0 \tag{1.27}$$

which gives the step,

$$\Delta\vec{x}_{NR} = -J^{-1} \cdot G(\vec{x}). \tag{1.28}$$

If $G(\vec{x})$ is viewed as the gradient, this step resembles a Newton step, where the Jacobian is essentially the Hessian matrix. At this point the objective function is evaluated at $\vec{x}^*$ to determine whether it will decrease if the current step is taken. If a decrease occurs this step is taken, otherwise a steepest descent like step is examined. Defining the sum of squares,

$$F(\vec{x}) = \sum_{i=1}^{n} g_i(\vec{x})^2, \tag{1.29}$$

which is to be minimized, the steepest descent direction is given by the negative gradient of $F(\vec{x})$. This gives a step,

$$\Delta\vec{x}_{SD} = -J^{\mathrm{T}} \cdot \begin{bmatrix} g_1 \\ g_2 \\ \vdots \\ g_n \end{bmatrix} \tag{1.30}$$

The above two approaches are equal only if minimizing the sum of squares results in a value close to zero.

In VA, these two methods are combined into one step as,

$$\Delta\vec{x} = \Delta\vec{x}_{NR} + \mu\Delta\vec{x}_{SD}, \tag{1.31}$$

where determining μ requires extensive derivation. Note that setting μ to be small

results in a step more like Newton's method, whereas taking μ to be large favours the steepest descent like step. Since it is known that Newton's method requires an initial guess relatively close to the solution and that steepest descent performs best when well away from the solution, these two methods complement each other. Thus it is apparent that the choice of μ will depend on where on the potential energy surface the current point is. Hence, the value of μ should change dynamically as the optimization proceeds.

1.5 Genetic Algorithms

A method which has recently become popular for optimization problems in several disciplines is Genetic Algorithms (GA's). Genetic algorithms are a robust technique, in the sense that they have been successfully applied to a broad range of problems, including areas in which other methods have proved to be difficult or incapable of finding a solution. With respect to chemistry applications, GA's have been applied to various problems [20], including geometry minimization of clusters by Mestres and Scuseria [21], various conformational searches, and docking studies for drug design. However, the use of GA's for transition state structure optimization is new, and is the topic of the remainder of this thesis.

1.6 Outline

From the above discussion it is apparent that further research into the optimization of transition state structures is required, especially in comparison to optimization of minima. In the following chapters, the application of genetic algorithms to this problem will be discussed. Chapter 2 gives a brief overview of what genetic algorithms are and how they are used, followed by the presentation of the results obtained from optimization of a mathematical function with the current genetic algorithm implementation in Chapter 3. Chapter 4 gives several results obtained for various chemical reactions and compares these results with those obtained using the VA technique. Finally, a summary of the research performed, and some ideas for future work are discussed in Chapter 5.

Chapter 2

Genetic Algorithm Background

"...any variation, however slight and from whatever cause proceeding, if it be in any degree profitable to an individual of any species, in its infinitely complex relations to other organic beings and to external nature, will tend to the preservation of that individual and will generally be inherited by its offspring. The offspring, also, will thus have a better chance of surviving...I have called this principle, by which each slight variation, if useful, is preserved, by the term of Natural Selection, in order to mark its relation to man's power of selection. We have seen that man by selection can certainly produce great results, and can adapt organic beings to his own uses, through the accumulation of slight but useful variations, given to him by the hand of Nature."

-Charles Darwin, The Origin of the Species, Chapter 3.

2.1 Introduction

Based on population genetics and Darwin's theory of natural selection, genetic algorithms are a type of evolutionary computing that solves problems by probabilistically searching the solution space. In contrast to most algorithms which work by successively improving a single estimate of the desired optimum via iterations, GA's work with several estimates at once, which together form a *population*. Given an initial population of individuals representing possible solutions to the problem, genetic algorithms simulate evolution by allowing the *most fit* individuals to reproduce to

form subsequent generations. After several generations, convergence to an optimal solution is often accomplished. Determining the fitness of an individual is problem dependent and the fitness function usually incorporates *a priori* knowledge of the desired optimum. The basic genetic algorithm is improved by using problem specific knowledge in specifying the various operations required to direct the evolution. A discussion of the basic components will be given below in Section 2.2, followed by the incorporation of specific knowledge of first order saddle points in Chapter 3.

Genetic algorithms have been applied to a very broad range of problems, in particular, problems associated with searching and optimization. Increasing application complexity often requires larger and larger population sizes to sufficiently sample the search space and achieve a satisfactory solution.

The terminology associated with genetic algorithms is analogous to that of biological systems. A *generation* is defined as one cycle of fitness evaluation, selection of parents, and reproduction. See Figure 2.1 for a flow chart of a genetic algorithm, which will be described below. Individuals are usually represented by a single *chromosome*, given as a string of binary bits. Each bit represents a *gene*, and a given expression of that gene (0 or 1) is an *allele*. The bits of an individual encode the values for the variables of the problem, where the encoding scheme used is somewhat problem dependent and chosen by the implementor.

The selection of parents generally involves a random choice among the most fit individuals of the population, in an attempt to propagate good traits through the

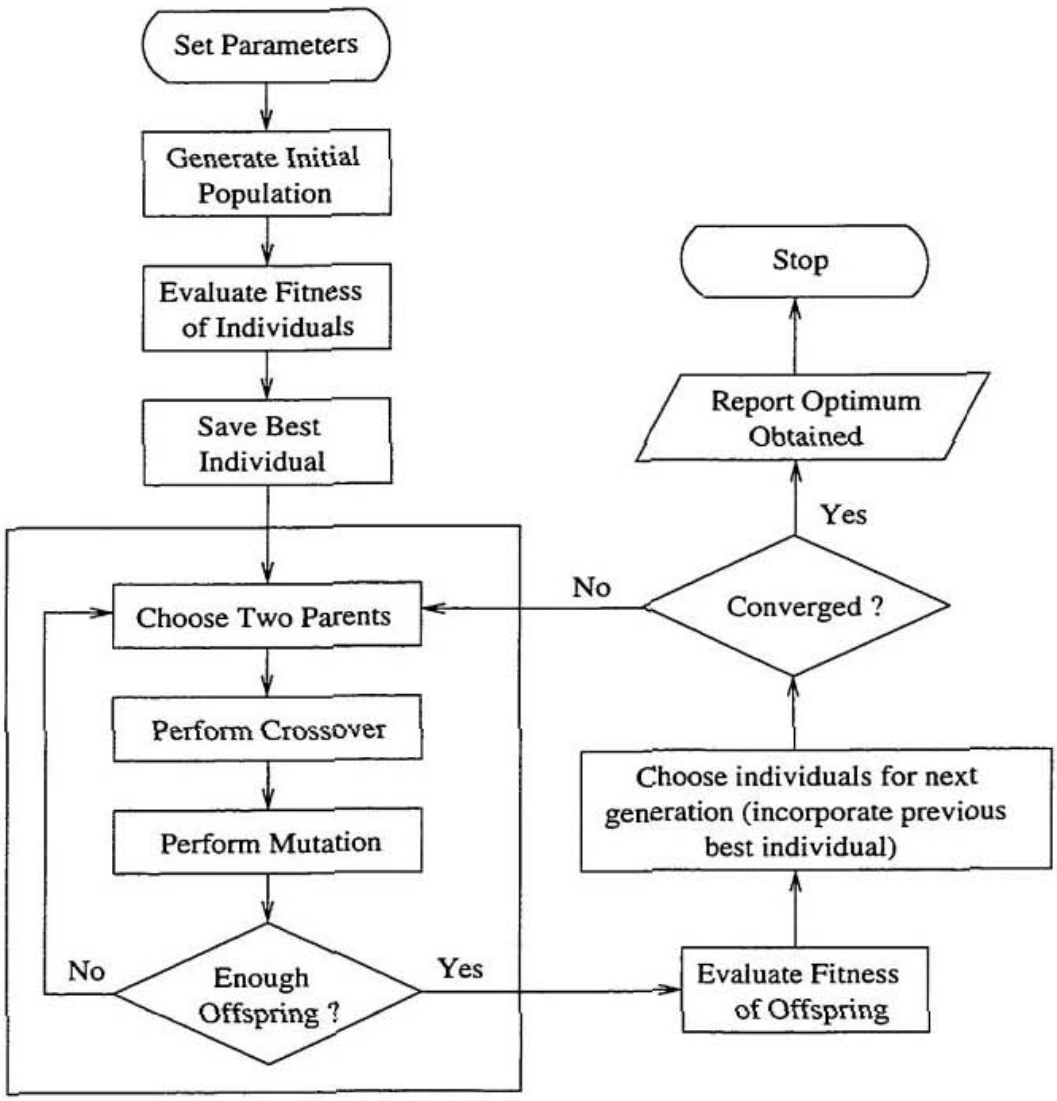


Figure 2.1: Flow chart for a Genetic Algorithm

population. The most fit individuals in the population tend to produce more offspring by being selected for reproduction many times. An individual's fitness value is determined by the evaluation of a problem dependent fitness function. The chromosome of an individual represents its *genotype*, while the fitness value represents its *phenotype*. Reproduction is performed by crossing over the chromosomes of the two parents to form offspring, followed by occasionally mutating some of the bits ($0 \leftrightarrow 1$) in the offspring. Reproduction accomplishes recombination of the genetic material, maintaining diversity in the population, thereby ensuring that the solution space is well sampled, and thus increasing the probability of obtaining an optimal solution.

Finally, after reproduction has taken place, a sample of individuals must be chosen to form the population for the next generation. After several generations, those individuals in the population that are most fit will tend to dominate, and, provided the algorithm has been well designed, the average fitness of the population will increase, leading to convergence to an optimal solution. The various operations in a genetic algorithm are discussed in more detail in the following section.

2.2 Basic Principles

The operations in a genetic algorithm are dependent upon the problem being solved, and many of the decisions are based on a combination of trial and error and previous experience. Some of the many ways to implement the various operators, and their advantages and disadvantages will be discussed. The detailed behaviour of

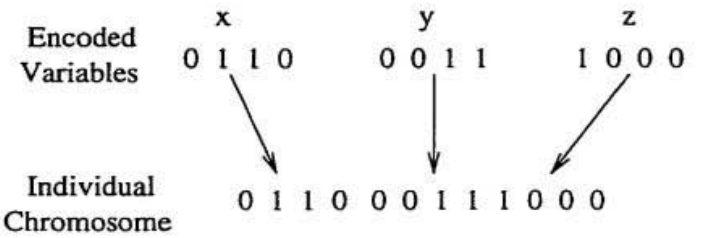


Figure 2.2: Example of forming a chromosome from encoded variables.

these operators, in turn, depend on several parameters, the values of which can lead to drastically different evolution. Hence, careful choices must be made. Previously documented experiments can help in making these choices. Some of the parameters involved, along with some issues that may help in choosing values suited to a particular problem, are presented below.

2.2.1 Encoding

The chromosome string of an individual contains an encoding of that individual's solution to the given problem. The solution of most problems consists of a set of values defining the problem variables. The variables involved in optimizing a function $f(x, y, z)$, for example, are x, y, and z. The values of the problem variables are often separately encoded as binary strings and concatenated to form the chromosome, as shown in Figure 2.2. The encoding scheme, along with the fitness function, are the two most important considerations for an efficient GA implementation. The encoding scheme is important when dealing with real life problems since a particular

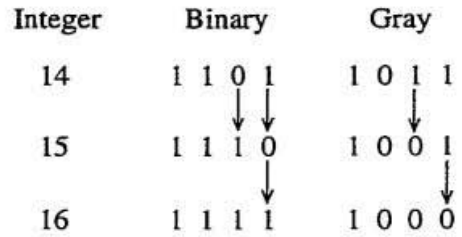


Figure 2.3: Example of the Gray code characteristic that two successive values differ by one bit flip.

chromosome may represent an invalid solution. An effective genetic algorithm must take this into consideration in some way, whether it be within the encoding scheme or in the reproduction operators. A possible alternative to standard binary encoding is Gray code. Gray code is similar to binary with the added feature that the Hamming distance, the number of bits that differ between the bit string representations of two adjacent numbers, is constant. In the case of Gray code, two successive integer values differ by only one bit flip, as shown in Figure 2.3. In most problems that require optimization, the variables involved are in real number space. Hence, the encoding scheme must also provide a way to store real number data in a binary string. The most common encoding scheme is interval (or range) encoding, where the domain and desired precision for the variables are specified initially, and the number of bits to be used for each variable is given by,

$$n = \log_2 \left(\frac{domain}{precision} \right). \tag{2.1}$$

The scaled decimal equivalent (D) of each variable x_i is given by,

$$D = \frac{x_i - a_i}{b_i - a_i} \cdot (2^n - 1) \tag{2.2}$$

where $[a_i, b_i]$ is the domain of x_i. The binary representation of this value is used as the encoded variable. The larger the number of bits used for each variable, the finer the resolution obtained for a given domain.

An alternative to the interval encoding is multiplicative encoding where each real valued variable is multiplied by $10^{accuracy}$ and truncated, where the accuracy is determined by the number of accurate decimal places required in the solution. The binary representation of the resulting integer is used as the encoded variable. The encoded variables used throughout the evolution will normally be decoded (by reversing the encoding process) for fitness evaluation.

2.2.2 Initial Population

Generation of an initial population of individuals is often performed in a stochastic manner. However, for many applications, several infeasible individuals may result. It is generally better to generate a population using *a priori* information of the optimum sought. For geometry optimization such as that being considered in this thesis, *a priori* knowledge of a chemical reaction can lead to a reasonable guess at the optimum. Thus, this initial guess provides a good starting point for the algorithm. One critical

factor is the number of individuals in the population, the population size, μ, which can range from just a few individuals to several thousands. The population size is generally kept constant throughout the optimization. A rule of thumb for choosing μ is 10 times the number of variables but no less than 100. A population that is too small will tend to give poor sampling and hence poor solutions, generally representing local rather than global optima. This concern is more detrimental for problems involving a large or convoluted search space, such as the optimization of proteins, which have complex structure and many conformations. However, there is usually a population size above which no improvement is seen, regardless of the number of generations completed. For the optimization of transition state structures, the initial geometry is usually relatively close to the optimum sought and hence the optimization is considered a local search. However, given the accuracy that is required in the optimized geometry ($\approx 10^{-6}$), a sufficiently large population will still be required.

2.2.3 Fitness Function

The fitness function is the most fundamental component of a genetic algorithm. It is the sole mechanism for directing the evolution toward the desired objective. An individual's fitness value should represent how good of a solution to the given problem it represents. The fitness function should take into consideration each variable to be optimized, and combine them in such a way to produce a suitable numerical fitness

value when applied to an individual. To ensure that the evolution is efficient, that is, the subsequent individuals are most likely to be biased toward better fitness values, the fitness function should contain few extrema, the ideal case being a monotonically increasing or decreasing function with a single maximum or minimum.

2.2.4 Selection

Selection is a means to favour the most fit individuals in the population in order to propagate good genes through the population. Some of the ways to select parents for reproduction are, roulette wheel selection, tournament selection, rank selection, sigma scaling, and Boltzmann selection. Although all methods use randomized processes, each has an ordering scheme with which to bias the choice of the most fit individuals.

Roulette wheel selection is equivalent to giving each individual a slice of a circle, with the size of the piece proportional to the individual's fitness. A point along the edge of the circle is randomly generated, and the individual whose slice of the circle contains this number is selected. Roulette wheel selection can cause problems if one individual in the population is much more fit than all of the others. In this case it can dominate the population resulting in *premature convergence*, that is, early convergence to a suboptimal solution.

Tournament selection involves randomly choosing a number of individuals to take part in a tournament. The individual with the largest fitness in this tournament pool is selected for reproduction. A larger tournament size results in a higher *selection*

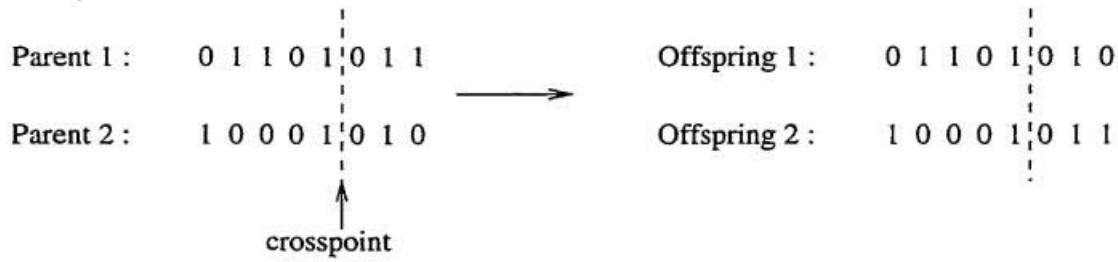


Figure 2.4: Application of the single-point crossover operator on 2, 8-bit individuals

pressure, which can be quantitatively viewed as the ratio of maximum to average fitness values of the current tournament. Selection pressure can also be increased by using a probabilistic tournament selection. In this case, the best fit individual in a tournament is selected with a user defined probability. A larger probability results in a higher selection pressure, a probability of 0.5 represents no selection pressure. Too high of a selection pressure leads to fast convergence to solutions that are suboptimal. while too low of a selection pressure leads to long execution times for convergence.

2.2.5 Crossover

Crossover is used as a means to generate better individuals than those that were present in the population previously. Two individuals are chosen as parents, their bit strings are aligned, and a crossover point is chosen randomly (see Figure 2.4). The strings are then crossed by exchanging the bits to the right of the crossover point, forming two new individuals. The crossover just described is called *single-point crossover*. Multi-point crossover can also be used, and is implemented in a similar

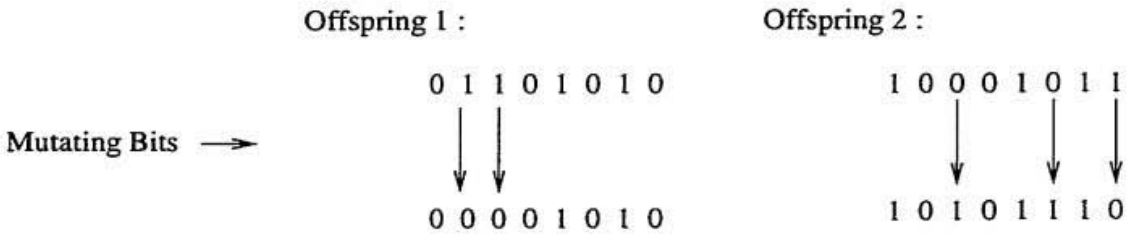


Figure 2.5: Application of the mutation operator on offspring produced by crossover

way, swapping portions between crossover points. Crossover is performed based on a user defined probability, p_c, which is usually set between 0.60 and 1.0. Hence, some parents pass on their genetic information directly to their offspring, without modification by crossover.

2.2.6 Mutation

Mutation is used to maintain diversity in the population. Although it does not necessarily generate better individuals, mutation prevents stagnation in a population of like individuals, which otherwise may not evolve to an optimum. The genes chosen for mutation have their state reversed by changing 0's to 1's, and 1's to 0's, as required (see Figure 2.5). Mutation is performed with a user defined probability, p_m, which is usually set very low (0.001 – 0.01) to prevent a large disruption of the genes in the population. A mutation probability of 0.5 results in the generation of offspring in a manner akin to a random walk. The issue to be addressed when choosing a mutation rate is to strike a balance between destroying good genes and using mutation as a

beneficial search operator. Too high of a mutation rate results in an undirected search, while too low of a mutation rate can lead to premature convergence or stagnation. Gray code can sometimes help moderate mutation effects in this respect, since one bit flip changes the number by the smallest integer amount, regardless of which bit is flipped. In contrast, flipping a more significant bit in standard binary results in a big change in the variable value. [1]

2.2.7 Incorporating Offspring

After reproduction is complete, λ offspring and $\mu - \lambda$ parents must be chosen to become the population for the next generation. Since some of the offspring may be less fit than the parents, most implementations choose a combination of offspring and parents for the next generation. To bias the evolution toward the desired optimum, the most fit of the two groups can be chosen, until the required number of individuals is reached. Hence, the most fit of the offspring replace the least fit of the parents. It is normal for the single best individual from the previous generation to be copied back into the population for the next generation. This practice is known as *elitism*.

[1] An example of the effect of flipping a single bit in standard binary versus gray encoding is shown, where the value 1.54 is encoded by multiplying by 10^2. The original binary strings as well as the result of flipping a single bit are shown.

Encoding	Bit String	Mutated	Decoded Value
Binary	10011010	00011010	0.26
Gray	11010110	01010110	0.86

Note the much larger difference that results from flipping the most significant bit of the binary encoded variable compared to the Gray encoded variable.

Methods for incorporating offspring can become rather sophisticated and vary widely.

2.2.8 Convergence

For the purposes of optimization, genetic algorithms are usually terminated when the solution has converged. Convergence can be determined in several ways. One common method is to terminate evolution when the quality of the solutions have not improved for a number of generations. Yet another, is to terminate when a given individual, with a high fitness value, has occurred a number of times. In general, evolution can also be terminated after a user defined number of generations has evolved. As the algorithm converges, the average fitness, and the fitness of the best individual increases, with the average approaching the highest fitness value as the optimum is reached. It is likely that some problem specific variable could be used to determine convergence.

2.3 Why Genetic Algorithms Work

When considering the question of why GA's work one must not forget that genetic algorithms were modelled after natural biological processes, which have proved their efficiency as demonstrated by *our own* human evolution.

Following the development of practical application details, the theoretical foundations for why genetic algorithms are able to mimic nature should be addressed. The basis for a formal means to do this was provided by John Holland [22], who introduced

the *schema theorem.* A schemata is defined as a bit pattern that is represented as a binary string of 0's, 1's, and $*$'s, where a $*$ is a "don't care" symbol, and can thus represent a 0 or a 1. A given chromosome contains many schema. For example, the chromosome 1101 contains the schema $11**$, $1*01$, etc. Two properties of schema are: The *order* of a schemata, which is the number of static bits, or the number of non-$*$ bits. The *defining length* of a schemata is the distance between the furthest static bits.

For each generation, individuals are considered for reproduction, with higher fit individuals more likely to pass on their traits to the next generation. Since selection is based on fitness, and one assumes that higher fitness values are a direct result of good schema, the representation of good schema in the population should increase exponentially in successive generations. These are the ideas of Holland's schema theorem. As an implication of this theorem, if one considers the number of schema compared to the number of individuals, the ratio is very large. According to Holland, the number of schema processed for each generation is $\propto \mu^3$, where μ is the population size.

In addition to Holland's schema theorem, a well-known approach by Goldberg [23], called the Building Block Hypothesis, also attempts to explain why GA's work. Goldberg defines the term *building block* as a "highly fit schemata of low defining length and low order." The idea behind the building block hypothesis is that, in a GA, convergence to the optimum occurs because of the placement of building blocks.

This hypothesis leads to encoding criteria for efficient performance of a genetic algorithm. These criteria include placing related genes close together in the chromosome, and having little interaction between the genes. If both criteria are satisfied the effectiveness of the GA is determined by the schema theorem. However, these criteria are difficult to satisfy. In the majority of cases there is some interaction between genes, that is, the amount that a given gene contributes to the fitness value depends on the value of its interacting gene(s). Satisfaction of both criteria is also snagged by lack of *a priori* knowledge of interaction between genes, and, in order to fill the first criterion, the second must be met. The best resort is to come as close as possible to satisfying Goldberg's encoding criteria.

As part of this thesis, an implementation of genetic algorithms for optimization of chemical structures was written in C, interfacing with Fortran 90 in the Mungauss package. Details of this implementation can be found in Appendix A. Chapter 3 presents results and a discussion on parameter selection.

Chapter 3

Results From Mathematical Functions

"In theory, there is no difference between theory and practice. In practice there is."

-Unknown

In this chapter a discussion of the results obtained from the optimization of mathematical functions using genetic algorithms is given. The purpose of these results is to demonstrate the effectiveness of the algorithm as well as to analyze its behaviour.

3.1 The Sample Problem

Two-dimensional mathematical functions provide a good testing medium since their surfaces are easily constructed and their stationary points can be visualized. This is unlike chemical reactions, where the surface is not known, and in most cases neither are the stationary points. As a means of testing the genetic algorithm code implemented, the following mathematical function given by Chong and Zak [24] was

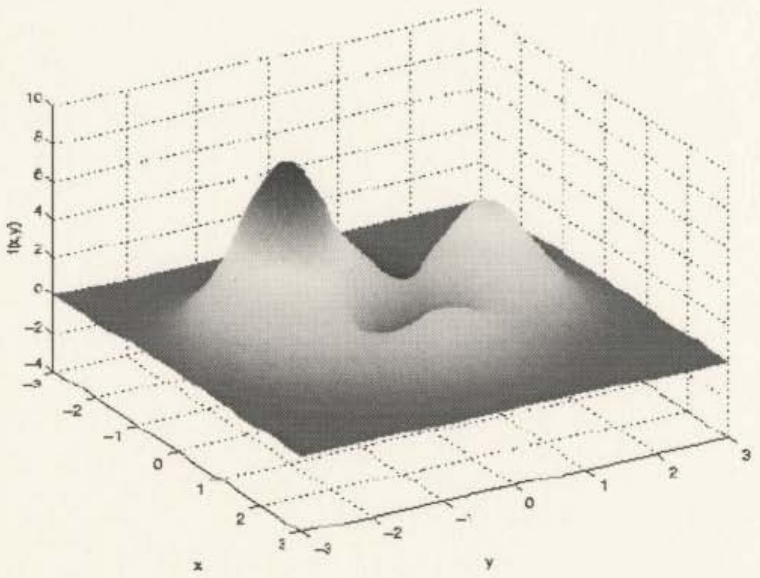


Figure 3.1: Surface of the Chong-Zak function given in Equation 3.1.

used,

$$f(x,y) = 3(1-x)^2 e^{-x^2-(y+1)^2} - 10\left(\frac{x}{2} - x^3 - y^4\right) \cdot e^{-x^2-y^2} - \frac{e^{-(x+1)^2-y^2}}{3}, \qquad (3.1)$$

whose surface is shown in Figure 3.1[1]. This function contains two minima, three maxima, and three saddle points, as can be seen in Figure 3.2. The coordinate values and characteristics of these stationary points are listed in Table 3.1

For the given objective function two fitness functions were constructed appropriately to seek out both minima and first order saddle points respectively. Note that

[1]This surface looks slightly different from the one mistakenly shown in [24]. Despite this fact, the function used here is identical to the one stated in the reference. Furthermore, the authors of [24] identify the stationary points of their imaged function, not of the function given in the text. However, the correct stationary points for Equation 3.1 are given in this text in Table 3.1.

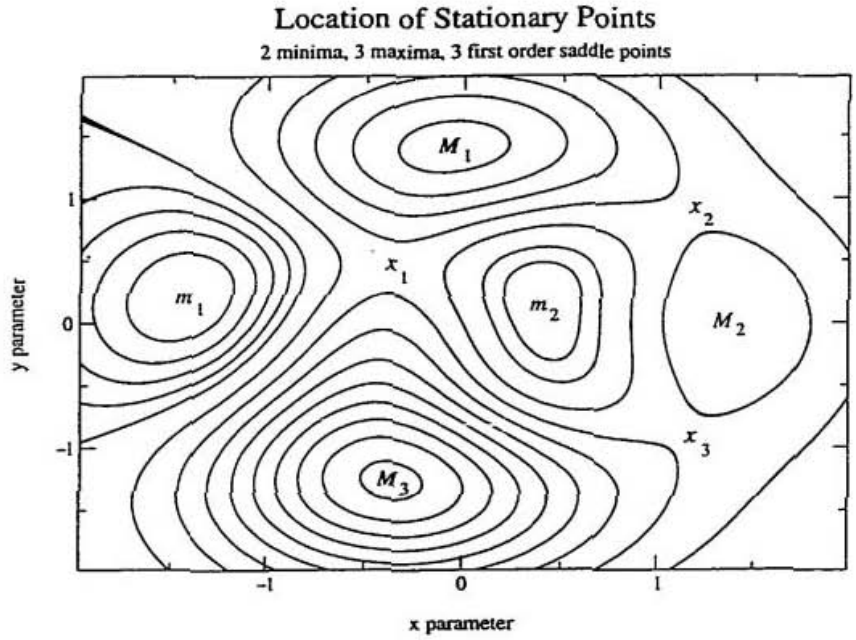


Figure 3.2: Contour plot of Equation 3.1 showing minima (m), maxima (M), and saddle points (x) (see Table 3.1). The stationary points are labelled for future reference.

Table 3.1: Location and characteristics of the stationary points for the Chong-Zak function. (x, y) is the coordinate location, $f(x, y)$ is the function value, $\|\vec{g}\|$ is the gradient length, λ_1 and λ_2 are the eigenvalues of the Hessian matrix. Note that $\|\vec{g}\|$ would be exactly 0 at the stationary points if calculated analytically.

label	x	y	$f(x,y)$	$\|\vec{g}\|$	λ_1	λ_2
m_1	-1.431359	0.206945	-2.467838	3.246533×10^{-6}	7.4497	14.3481
m_2	0.404936	0.166523	-0.930778	1.703883×10^{-6}	5.4577	23.8348
M_1	-0.059953	1.409113	5.425638	3.490810×10^{-6}	-21.6508	-11.1346
M_2	1.370701	-0.008093	2.909429	2.018528×10^{-6}	-14.7846	-5.7225
M_3	-0.365185	-1.263316	9.276397	4.078678×10^{-6}	-28.1092	-20.0956
x_1	-0.364835	0.455781	1.665537	6.698525×10^{-6}	-20.1040	15.5895
x_2	1.148302	0.868567	1.897576	3.462192×10^{-6}	-9.9574	6.7342
x_3	1.154115	-0.890394	1.915014	1.581266×10^{-6}	-9.2596	6.5939

no higher order saddle points exist for the current function since $f : \mathbb{R}^2 \mapsto \mathbb{R}$.

3.2 Fitness Evaluation

Since the ultimate goal is to optimize first order saddle points on a potential energy surface, a fitness function was designed to isolate the desired features of such stationary points. As noted previously, first order saddle points can be characterized by a zero gradient and one negative eigenvalue in the Hessian matrix. With this in mind, the following fitness function was developed,

$$f(\vec{x}) = \frac{1}{\|\vec{g}\| + \epsilon} \cdot \frac{1}{(n-1) + \epsilon} \tag{3.2}$$

where $\vec{x}$ is a vector of the problem variables, $\|\vec{g}\|$ is the l_2-norm of the gradient vector divided by $\sqrt{k}$ where k is the number of problem variables (or *gradient length*), n is the number of negative eigenvalues in the Hessian matrix, and ϵ is a parameter chosen small[2] and used to prevent division by zero as the algorithm converges. Note that individuals that are close to a first order saddle point ($\|\vec{g}\| \approx 0, n = 1$) will have higher fitness values than those further away since their gradient norm and number of negative eigenvalues will result in small denominators in the fitness function. Hence, the genetic algorithm should be constructed to positively bias those individuals with

[2]The parameter ϵ is chosen to be on the order of the desired accuracy in the gradient length of the converged solution. A value of 1×10^{-6} is used since this is somewhat of a standard for convergence in chemical structure optimization.

the highest fitness values. With this fitness function, an "optimum" solution corresponds to a first order saddle point. Note that this fitness function does not bias one saddle point over another, therefore the algorithm could theoretically find all or any of the first order saddle points on the surface. For the optimization of minima the $(n-1)$ term in Equation 3.2 was changed to n such that individuals with all positive eigenvalues would be favoured. Various runs of the genetic algorithm, performed with different parameter values, are discussed in the remainder of this chapter to illustrate the behaviour of the method, and its dependence on these parameters.

3.3 The Defining Parameters

Several parameters characterize and control the operations of the genetic algorithm and affect the behaviour of the evolution towards an optimum. These parameters are listed in Table 3.2 below. Although many of these parameters were discussed briefly in Chapter 2, a few require further explanation. An initial population of individuals is generated by taking small random perturbations about an initial guess at the optimum. These perturbations are restricted by the value of M_{init} since it is assumed that the initial guess is a good one. Furthermore, since the problem to be solved typically involves data for a physical system, the individuals created throughout the evolution are restricted to within M_{sub} of the initial guess to avoid infeasible values. Any individuals that fall out of this region are forced back by changing the invalid variable values to small random perturbations from the initial

Table 3.2: Parameters in the current genetic algorithm implementation.

Notation	Description
x_0, y_0	Initial guess at the optimum.
μ	Population size.
G_{max}	Maximum number of generations.
p_c	Probability of crossover.
p_m	Probability of mutation.
n_b	Number of bits per variable.
M_{init}	Maximum amount to perturb the initial guess to form the initial population.
M_{sub}	Maximum amount any subsequent individuals can deviate from the initial guess.
S	Method used for selecting parents. Possible values are 't' for tournament selection and 'r' for roulette-wheel selection. For tournament selection, parameters include the size of the tournament, t_{size}, and the probability of selecting the most fit individual, t_{prob}.
E	Type of encoding scheme used. Possible values are 'm' for multiplicative and 'i' for interval (or range).
B	Binary representation used. Possible values are 'g' for Gray and 'b' for standard binary.
R	Type of replacement strategy used to form the population for the next generation. Possible values are 'a' for above-average and 'o' for all-offspring.
λ_{tol}	The value below which negative eigenvalues must lie to be counted as negative.

guess, as before. Following the completion of each generation, forming the surviving population for the next generation can be done in two ways. First, the population can be formed from only the offspring created in the current generation. Second, the population can consist of random selections from among the parents and offspring that have above average fitness values. In the latter case, the best individual from the previous generation is always copied back into the new population. The parameter R specifies which of these methods to use. Finally, λ_{tol} is a tolerance that is placed

on the value of the negative eigenvalues, below which they must lie to be counted as negative. This parameter is required because of finite machine precision which can cause values that are essentially zero to be very small, and maybe negative. Points with such negative eigenvalues are of no interest, but nevertheless would be considered favourable by the fitness function. For the Chong-Zak objective function this can cause problems at points in the flat region of the surface (outside the interesting regime) where the gradients are small and one of the eigenvalues is small and negative. In this case $\lambda_{tol} = -5.0$ has proved to be adequate to concentrate the search in the interesting region of the surface.

3.4 Results for First Order Saddle Points

The following subsections investigate the effect of each of the parameters in the algorithm. Since genetic algorithms are stochastic in nature, each run represents only one sample out of the total ensemble. Thus, to obtain statistically significant results data was taken over 25 runs for each parameter set and averaged. For each of these runs a different random seed was used and recorded. Subsequent data sets were generated using these same 25 seeds for consistency. For the current analysis the default parameter values are as follows: $x_0 = y_0 = 0.0$, $\mu = 100$, $G_{max} = 200$, $p_c = 0.75$, $p_m = 0.03$, $n_b = 24$, $M_{init} = M_{sub} = 2.0$, $S =$ 't,' $t_{size} = 6$, $t_{prob} = 0.75$, $E =$ 'm,' $B =$ 'g,' $R =$ 'a,' $\lambda_{tol} = -5.0$, with any differences noted in the captions of the respective figures.

3.4.1 Location of the Initial Guess

Since the initial population of individuals is created by perturbing an initial guess at the optimal variable values, the exact location of this initial guess can have a profound effect on the behaviour of the algorithm. Figure 3.3 shows the effect on the average and best fitness values resulting from changing the initial guess from $(0, 0)$ to $(-1, -1)$, with $M_{init} = M_{sub} = 3.0$ such that both cases encompass all of the interesting region of the surface. There is little difference in the overall behaviour

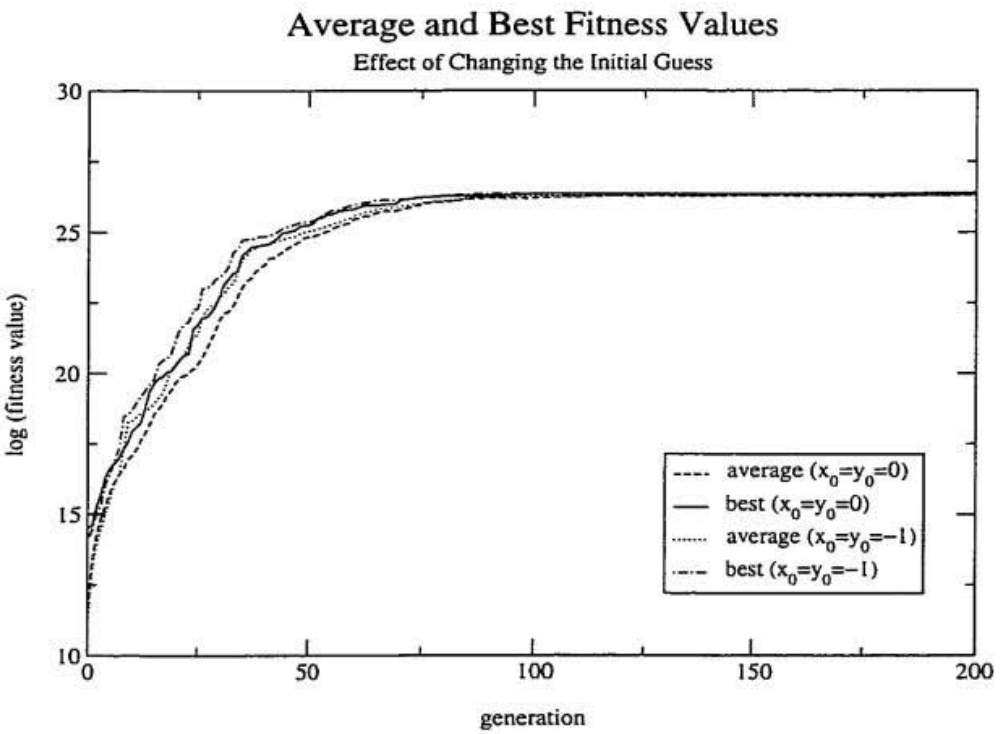


Figure 3.3: Plot of average and best fitness values for different initial guesses. The case with $(x_0, y_0) = (0, 0)$ results in slightly better fitness values.

of the algorithm due to changing the initial guess, since the average and best fitness

Table 3.3: Results obtained by varying the initial guess, where G_{avg} is the average number of generations required to find the best individual found in the evolution.

Initial Guess	Region	(x, y)	$\|\vec{g}\|$	n	G_{avg}
$(0, 0)$	x_2	$(1.148302, 0.868567)$	3.462192×10^{-6}	1	84
$(0, 0)$	x_3	$(1.154115, -0.890394)$	1.581266×10^{-6}	1	62
$(-1, -1)$	x_2	$(1.148302, 0.868567)$	3.462192×10^{-6}	1	60

values follow similar trends. However, the case with initial guess $(-1, -1)$ gave higher fitness values slightly earlier. Furthermore, the two cases report optima in different regions of the surface as shown in the results in Table 3.3. Initial guess $(0, 0)$ resulted in a nearly equal number of optima near x_2 and x_3 (see Figure 3.2). In addition, one of the runs for $(x_0, y_0) = (0, 0)$ resulted in premature convergence to a suboptimal solution. Initial guess $(-1, -1)$ resulted in the majority of optima near x_3, with a significant number of optima also found near x_3, in an average generation of 63. In addition to the difference in time required to cluster around a point, the initial guess can have a direct influence on the location of the optimum found, as is to be expected.

3.4.2 Effect of Perturbation and Validation Parameters

For the current genetic algorithm design, the parameters M_{init} and M_{sub} directly affect the behaviour of the algorithm and are closely linked with the location of the initial guess. A plot of the best fitness value each generation for initial guess $(-1, -1)$ and different M_{init} and M_{sub} values is shown in Figure 3.4. Again, the overall behaviour is similar, but with $M_{init} = M_{sub} = 3.0$ resulting in a slightly

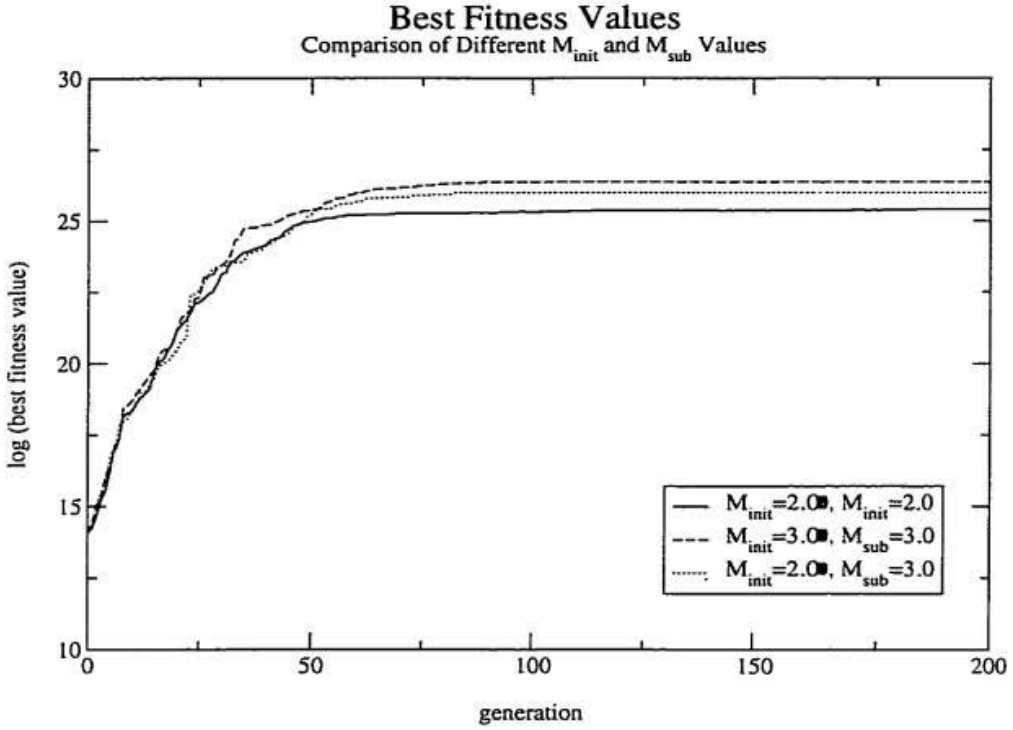


Figure 3.4: Plot of best fitness values for different values of M_{init} and M_{sub}. The case with $M_{init} = M_{sub} = 3.0$ gives slightly better fitness values.

better fitness value. However, the three sets of runs again result in different optima. Parameter values $M_{init} = M_{sub} = 2.0$ resulted in the majority off optima near x_1 in an average generation of 63 but with 4 runs converging prematur-ely. The set of runs with $M_{init} = 2.0, M_{sub} = 3.0$ resulted in the majority of optima near x_1 as well, but in an average generation of 55 and no runs converging prematurely. The set of runs with $M_{init} = M_{sub} = 3.0$ resulted in the majority of optima near x_2 in an average generation of 60, but did not report any optima near x_1. It is important to note that with $(x_0, y_0) = (-1, -1)$ and $M_{init} = M_{sub} = 2.0$, the only saddle point within

the region defined by the M_{init} and M_{sub} constraints is x_1. Clearly, the optimum obtained is highly dependent on the value of M_{init} and M_{sub} since smaller values of these parameters result in a more local search.

3.4.3 Effect of Population Size

To demonstrate the effect of population size on the outcome of a genetic algorithm, two cases with $\mu = 100$ and $\mu = 200$ are displayed in Figure 3.5. Results obtained

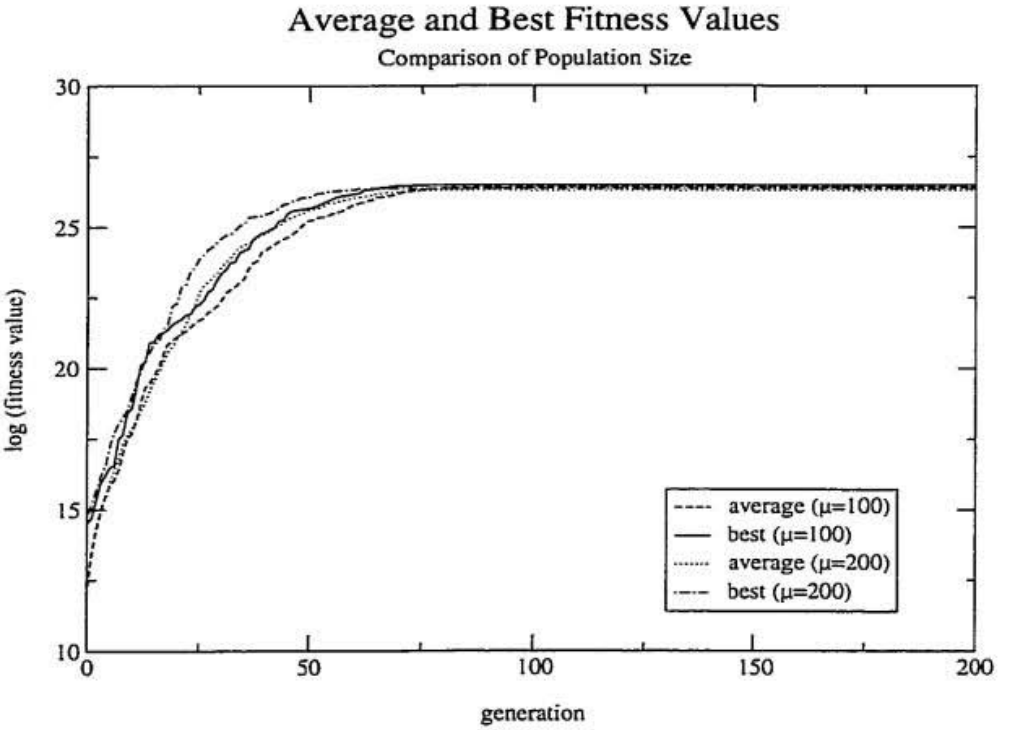


Figure 3.5: The effect of population size on the behaviour of the genetic algorithm. Doubling the population size has little effect, but does result in slightly earlier convergence.

for these sets of runs are shown in Table 3.4, where the percentage of runs resulting

Table 3.4: Results obtained for different population sizes, where $\mu = 200$ found an optimum slightly earlier on average than $\mu = 100$.

Region	% of runs		G_{avg}	
	100	200	100	200
x_1	8	4	49	33
x_2	28	56	57	46
x_3	64	40	57	49

in optima near each saddle point is shown. Note that both sets of runs resulted in optima near all three saddle points.

Doubling the population size makes very little difference, with the exception that the larger population gives earlier convergence on average. This is likely because the sampling of the surface is more thorough with the larger population, and it is therefore more likely to have individuals with high fitness values early in the evolution.

One very important factor in considering the usefulness of doubling the population size is the increase in computational overhead associated with more individuals. It would seem reasonable to assume that the improvement in behaviour with the doubled population would not be worth the expense unless the optimum was found in half the number of generations. For the current example this is not the case so doubling the population size is not likely to be worthwhile.

3.4.4 Effect of Crossover Rate

The effect of changing the crossover rate is shown in Figure 3.6 where the average and best fitness values are displayed for $p_c = 0.60$, 0.75, and 0.90. Results obtained

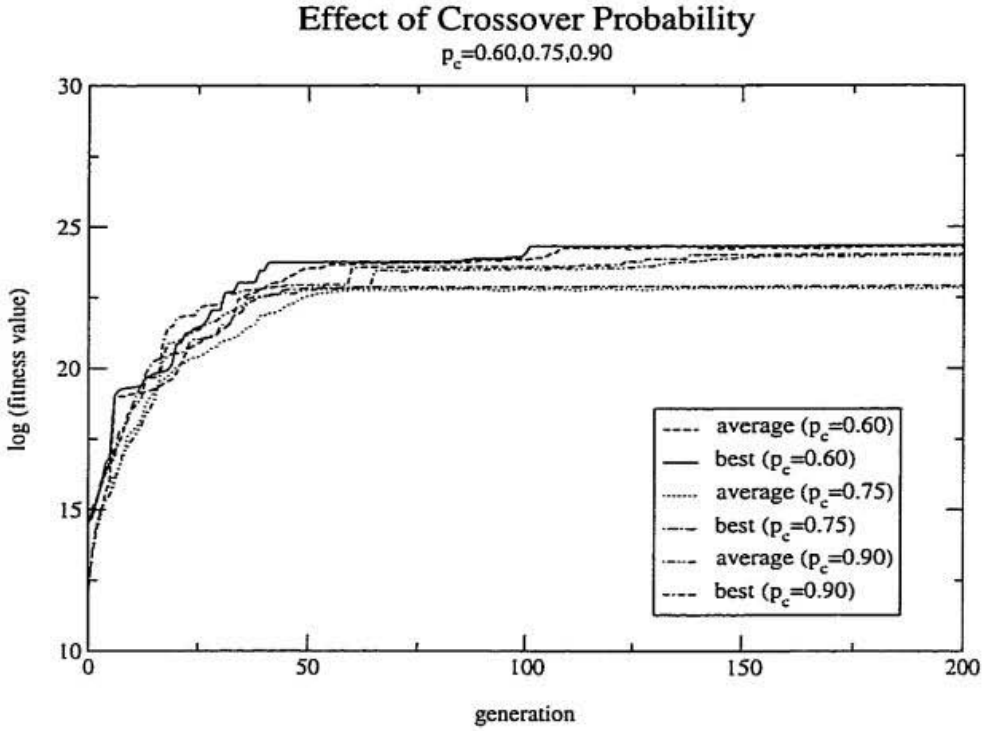


Figure 3.6: The effect of changing the crossover probability p_c with standard binary encoding. A crossover probability of 0.60 results in higher fitness values but gives slower convergence.

for these sets of runs are shown in Table 3.5. Note that all three values of p_c resulted in optima near all three saddle points, but somewhat faster convergence is achieved with $p_c = 0.75$.

In addition to the runs performed above one sample run of the algorithm with $p_c = 0.60$ resulted in premature convergence. A scatter plot of the individuals present every 40 generations during this run is shown in Figure 3.7 where the algorithm forms two clusters near M_2, ultimately favouring individuals in the cluster near x_3, yet never actually reaching the optimum.

Table 3.5: Results obtained for different crossover probabilities, showing $p_c = 0.75$ reporting an optimum slightly earlier.

Region	% of runs			G_{avg}		
	0.60	0.75	0.90	0.60	0.75	0.90
x_1	4	4	8	52	52	114
x_2	52	56	56	68	60	74
x_3	44	40	36	109	100	78

For the Chong-Zak function we conclude that $p_c = 0.75$ appears to be the best crossover probability due to faster convergence on average.

3.4.5 Effect of Mutation Rate

Thus far, most runs of the genetic algorithm reported a majority of optima in the region of the surface near saddle points x_2 and x_3, however there are three saddle points present. Once a good individual is found, all subsequent individuals seem to move in that direction and cluster within small regions[3]. In cases where the initial guess is a good one this is a desirable behaviour, since the clustering will most likely occur around the optimum sought. This effect can be seen in Figure 3.8 in which the individuals are displayed every four generations for the first twenty generations of a sample run. To prevent this rapid clustering, in the hope of finding the other saddle point, the mutation rate can be increased to ensure more thorough sampling of the surface, Figure 3.9 is a scatter plot generated by increasing the mutation rate

[3]There is a possibility that such clustering can actually lead to problems due to premature convergence. Later work by the author addressed this concern, and a discussion can be found in Section 4.4.

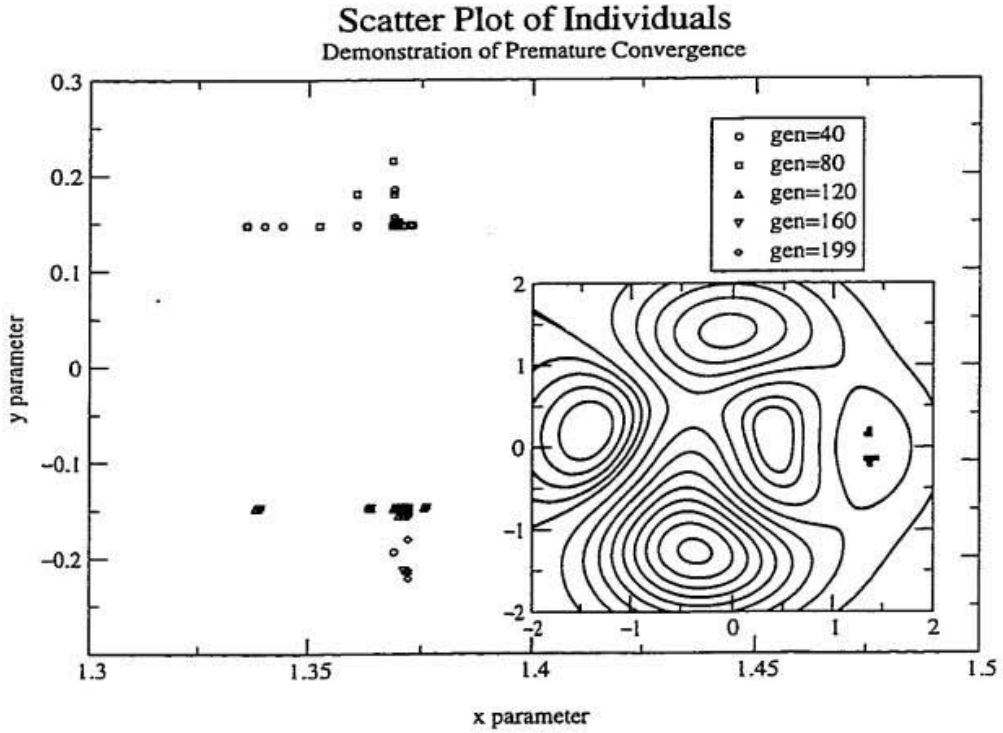


Figure 3.7: Scatter plot of individuals every 40 generations for $p_c = 0.60$, where each point represents a single individual in the population. This run demonstrates premature convergence. The location of the two clusters relative to the stationary points on the surface is shown on the super-imposed contour plot.

from 0.03 to 0.08. Note that there is distinct clustering around both saddle points x_2 and x_3 (regions A and B of Figure 3.9 respectively), but with much more dispersion than with $p_m = 0.03$. However, the best individual was still found near x_3, with $(x, y) = (1.148302, 0.868567)$, $\|\vec{g}\| = 3.462192 \times 10^{-6}$ and $n = 1$.

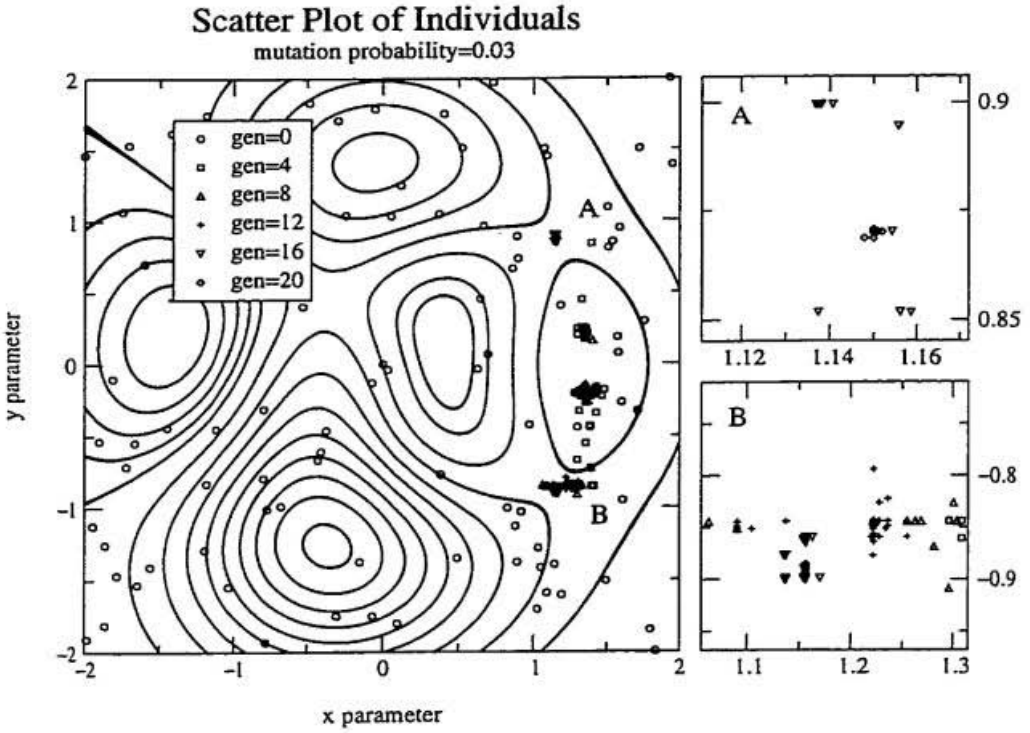


Figure 3.8: Scatter plot of individuals every 4 generations for the first 20 generations, with $p_m = 0.03$. The population quickly clusters around saddle point x_3 in region B. The optimum reported was $x = 1.154115, y = -0.890394$, where $\|\vec{g}\| = 1.581266 \times 10^{-6}$ and $n = 1$. Enlarged views of areas A and B are shown in the graphs on the right.

3.4.6 Effect of the Selection Method

To compare the various available strategies for choosing parents for reproduction, the two implemented methods are compared in Figure 3.10. Two cases using tournament selection are shown with different selection pressures[4] imposed by changing the number of individuals taking part in each tournament. Although there is little differ-

[4]see Section 2.2.4

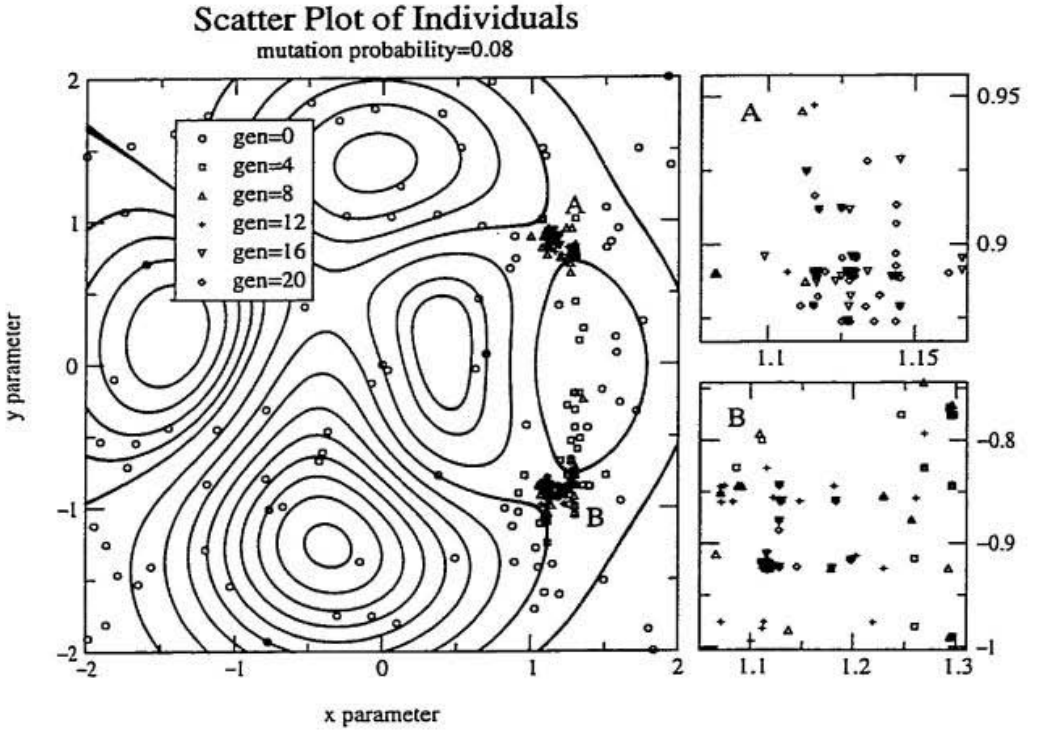


Figure 3.9: Scatter plot of individuals every 4 generations for the first 20 generations with $p_m = 0.08$. Although the optimum reported was in the region of x_3 (B), the sampling continues to cluster around x_2 (A) as well. Enlarged views of regions A and B are shown in the graphs on the right.

ence between the three plots, roulette-wheel and tournament selection with $t_{size} = 6$ behave very similarly, while tournament selection with $t_{size} = 2$ does not perform as well. Examination of the optima obtained reveals the same conclusion, as shown in Table 3.6. Note that neither $S =$ 't', $t_{size} = 2$ nor $S =$ 'r' found x_1, and that $S =$ 't', $t_{size} = 6$ gave results earlier than either of the other two methods. Hence, $S =$ 't', $t_{size} = 6$ appears to be the best selection method of the three tested.

However, one must also consider the slight increase in overhead associated with

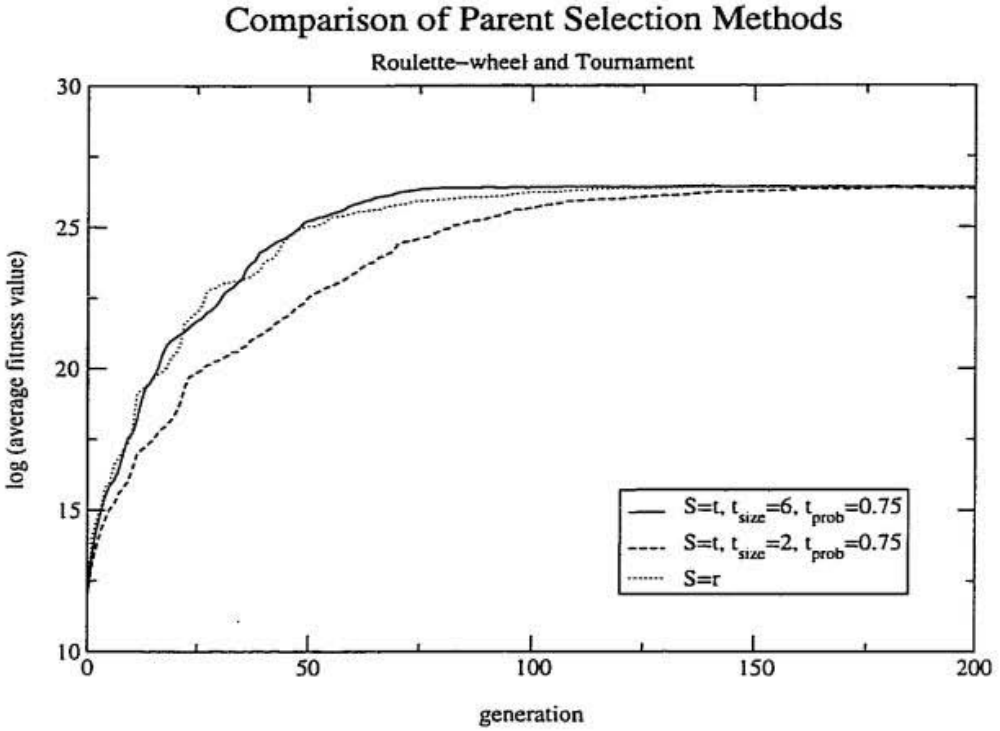


Figure 3.10: A comparison of roulette-wheel and tournament selection. Two runs of tournament selection are shown with different selection pressure, changed by using 2 individuals in each tournament instead of 6.

increasing t_{size} from 2 to 6 since there are 3 times more calls to choose random individuals from the population, as well as 3 times more comparisons of fitness values to determine the most fit individual[5]. Regardless, it is thought that the improvement in evolution speed is worth the small expense.

[5]see Section A.9.1

Table 3.6: Results obtained for different parent selection methods. Note that $S =$ 't' with $t_{size} = 6$ gave earlier results and is the only method that finds saddle point x_1.

S	Region	% of runs	G_{avg}
t, $t_{size} = 6$	x_1	8	48.5
t, $t_{size} = 6$	x_2	28	56.4
t, $t_{size} = 6$	x_3	64	56.8
t, $t_{size} = 2$	x_2	52	98.9
t, $t_{size} = 2$	x_3	48	97.6
r	x_2	48	65.5
r	x_3	52	71.5

3.4.7 Effect of the Encoding Method

The effect of changing the encoding method from multiplicative encoding to interval (or *range*) encoding is shown in Figure 3.11 which plots the average and best fitness values for both types of encoding. Interval encoding was used with $n_b = 22$ for the intervals $x, y \in [-2, 2]$ since a precision of 0.000001 is desired (see Equation 2.1). Therefore, to ensure a fair comparison, multiplicative encoding was used with $n_b = 21$ since the largest number to be encoded is 2.0, requiring a representation of at most 2×10^6. For multiplicative encoding only x_2 was found, with gradient lengths on the order of 10^{-6} in an average generation of 50.1. Whereas for interval encoding all three saddle points were found with gradient lengths on the order of 10^{-6} but slightly less than multiplicative, in an average of ≈ 55 generations. As a result, interval encoding is considered superior to multiplicative encoding since a slightly more accurate result was found with minimal increase in the number of generations and the problem space was better sampled, since all three saddle points were found.

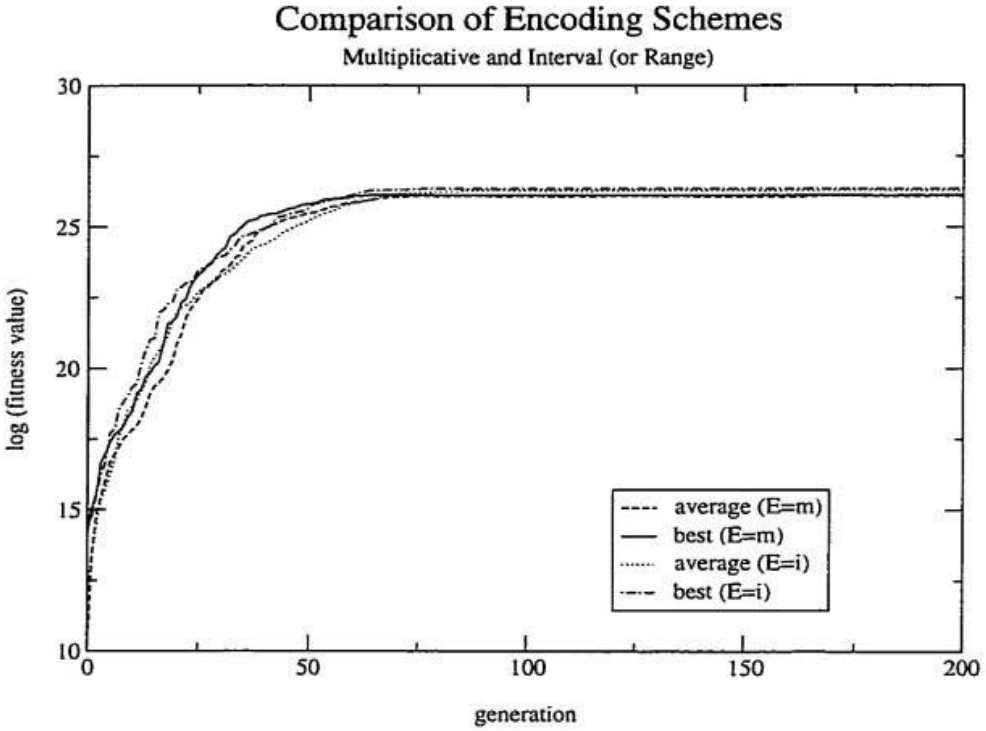


Figure 3.11: A comparison of multiplicative and interval encoding. The interval encoding method outperforms the multiplicative method since a slightly higher fitness value is attained, and all three saddle points are found as opposed to just x_2 for multiplicative encoding.

3.4.8 Gray versus binary

The encoding scheme is also defined by a choice between standard binary or Gray encoding. A comparison of the use of Gray encoding versus standard binary encoding is shown in Figure 3.12. Note that both sets of data follow the expected trend of the average fitness approaching the best fitness as the algorithm evolves. The Gray encoding performs much better, reaching an optimum after an average of ≈ 54 generations. The standard binary encoding however, levels off at a much lower fitness

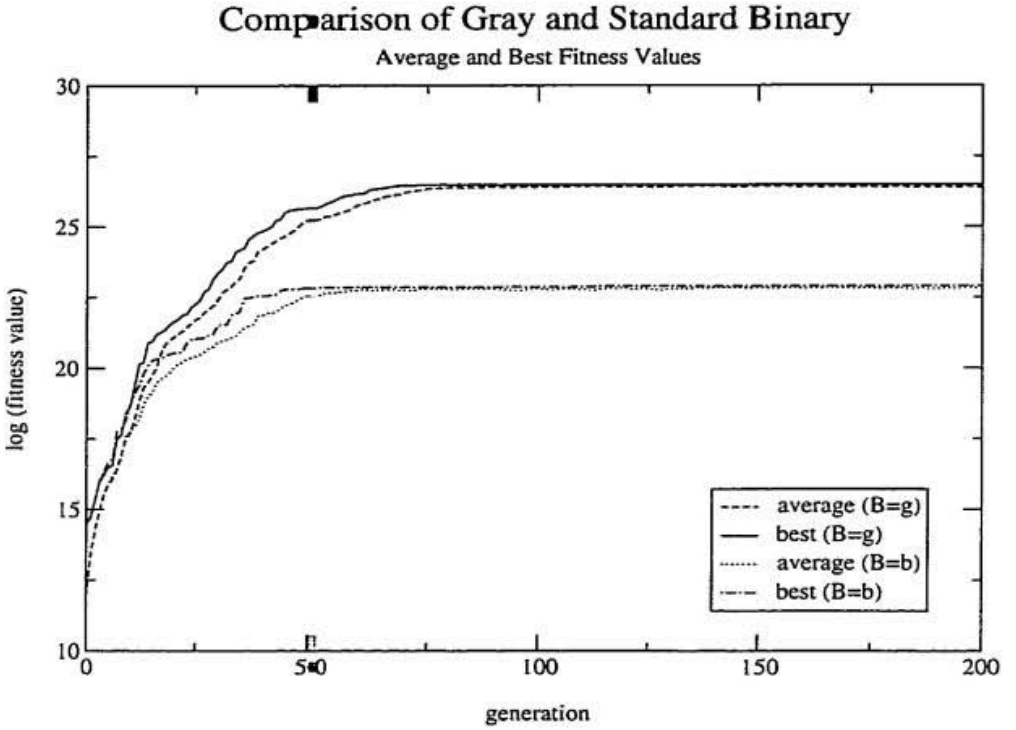


Figure 3.12: Plots of average and best fitness values for Gray and standard binary encoding. Gray encoding far outperforms standard binary encoding.

value. In addition, while the optima found in both cases contained one negative eigenvalue in the Hessian, the Gray encoded algorithm resulted in optima all with gradient lengths on the order of 10^{-6}. Whereas, the standard binary encoding only resulted in one run (out of 25) with an optimum with such accuracy. Clearly, Gray encoding proved to be the best encoding scheme for this example. This result was somewhat expected since Gray encoding is less sensitive to mutation effects, resulting in a more gradual, and smooth evolution[6].

[6]see Section 2.2.6

3.4.9 Replacement Strategies

The final parameter to be investigated is the method used to replace the individuals in the population from one generation to the next. Using only the offspring to form the population for the next generation can sometimes cause problems if many of the offspring are less fit than the parents. A plot of the average and best fitness values resulting from using the all-offspring replacement and the above-average replacement is shown in Figure 3.13. Note that for $R =$ 'o' the average fitness values remain noisy

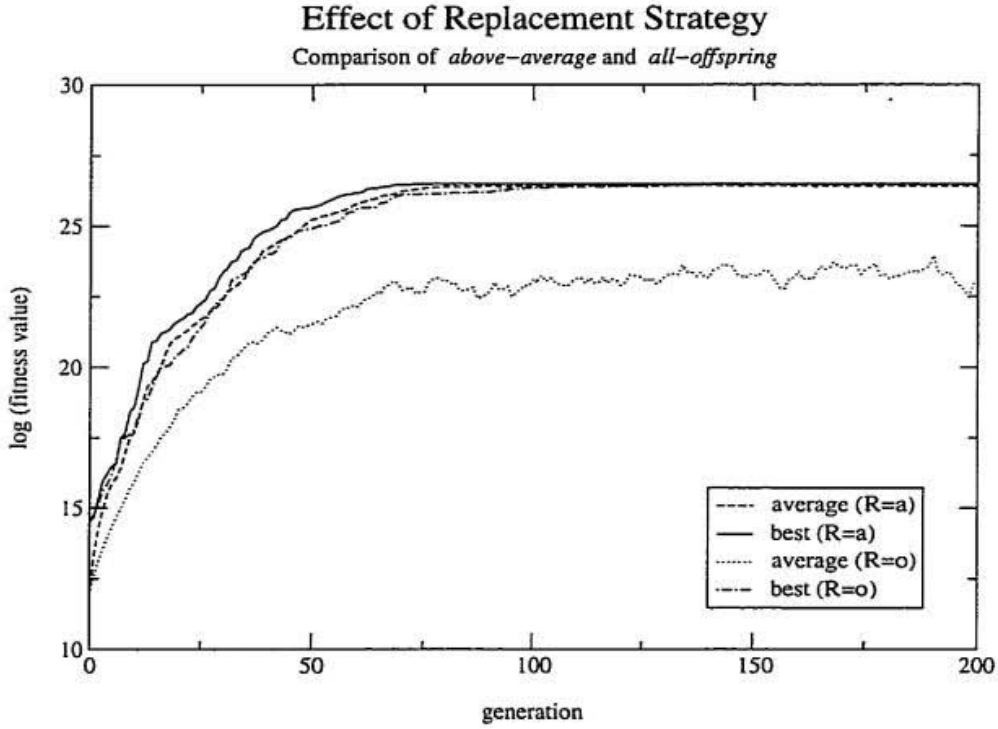


Figure 3.13: Plots of average and best fitness values for *above-average*($R =$ 'a') and *all-offspring* ($R =$ 'o') replacement strategies. The *all-offspring* strategy results in the average fitness values never approaching the best fitness values.

throughout the evolution, never approaching the best fitness as is expected. This is due to the stochastic nature of the offspring creation and the resulting lack of bias when using them as the new population. Examination of the individuals present in the population show that many individuals far removed from any first order saddle point remain in the population throughout the evolution, weighing down the average. This can be seen in Figure 3.14 where the contours of the function are superimposed

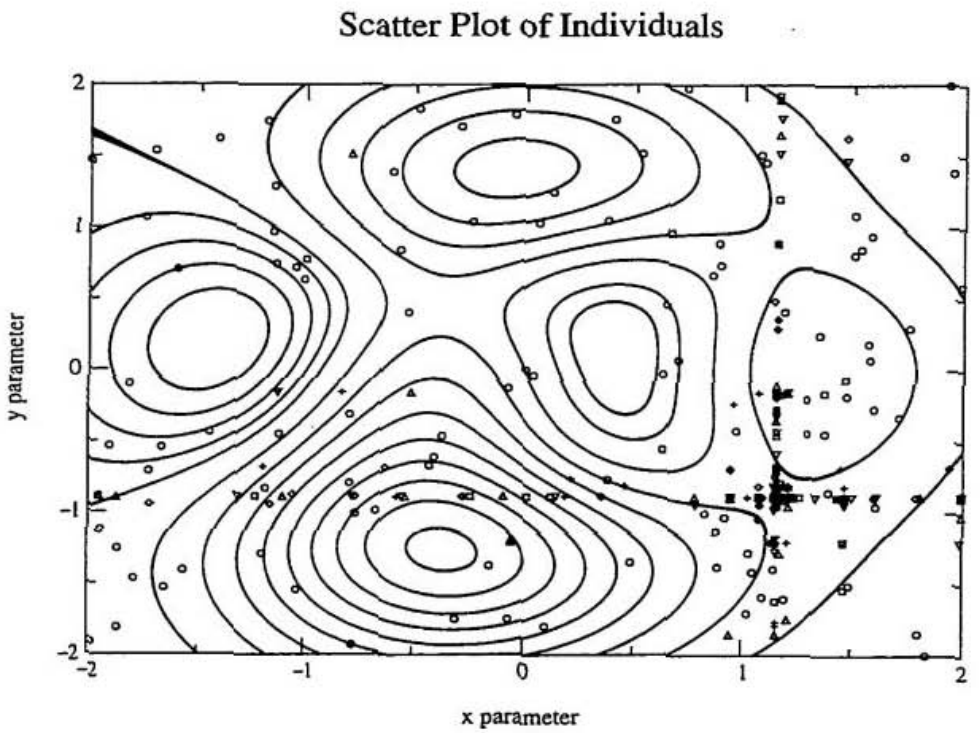


Figure 3.14: Scatter plot of individuals for *all-offspring* ($R =$ 'o'), displaying clustering around saddle point x_3.

on the graph.

An interesting feature to note is that many of the individuals have the same

x value, and many have the same y value. This phenomenon is due to single-point crossover, since this form of crossover changes only the variable in which the crossover point occurs, leaving all others the same. Hence, as the algorithm evolves and favours a given stationary point, most of the offspring formed will contain changes in only one of the problem variables, hence creating vertical and horizontal lines on a scatter plot (as seen in Figure 3.17) intersecting near the optimum. This effect would be diminished by increasing the mutation rate, causing many, or all of the variables to be modified in many of the offspring produced.

3.5 Objective Function Geometry Considerations

To explore how each of the runs for a particular set of parameter values contribute to the overall average obtained, a plot of the average fitness value of each generation for each run was plotted along with the overall average. This plot is shown in Figure 3.15 where the dots are the average fitness values for different runs and the solid line is the overall average of these average fitness values. Note that three distinct bands of points occur. Each of these bands corresponds to populations sampling the areas around different saddle points on the surface. In other words, sampling the region surrounding different saddle points can result in a different average fitness value for that sample. This phenomenon is a consequence of the encoding scheme used and the local geometry of the surface around the stationary points. The use of a particular encoding scheme is equivalent to defining a grid of discrete values on

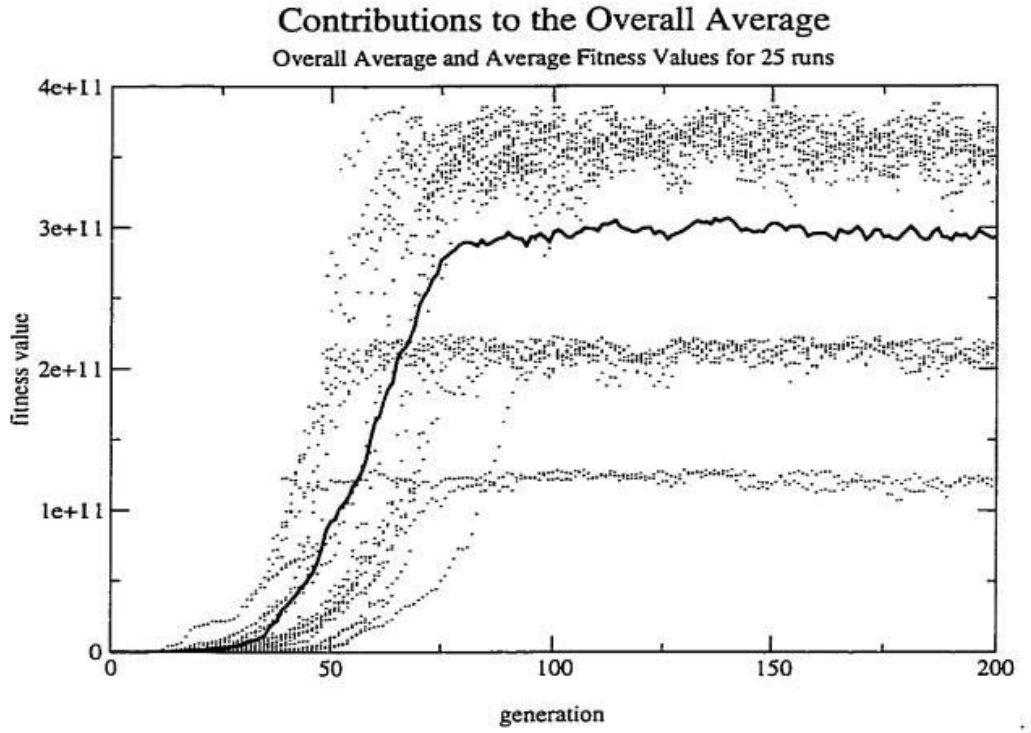


Figure 3.15: Plot of average fitness values each generation for each of 25 runs with the overall average of these runs. Three bands of points correspond to the three saddle points on the surface.

the surface of the objective function from which the variable values can be chosen to form individuals. To illustrate, consider a surface whose contours are shown in Figure 3.16. This surface displays two maxima of the same height, with maximum A laying atop a gently sloping hill and maximum B laying atop a steep slope. The same grid size, representing possible discrete values of a sample, is superimposed on both of these stationary points. Note that the objective function values under the grid at A do not change nearly as much as the objective function values under the grid at B. The fitness values will follow this same trend. Thus, a point on the grid a given

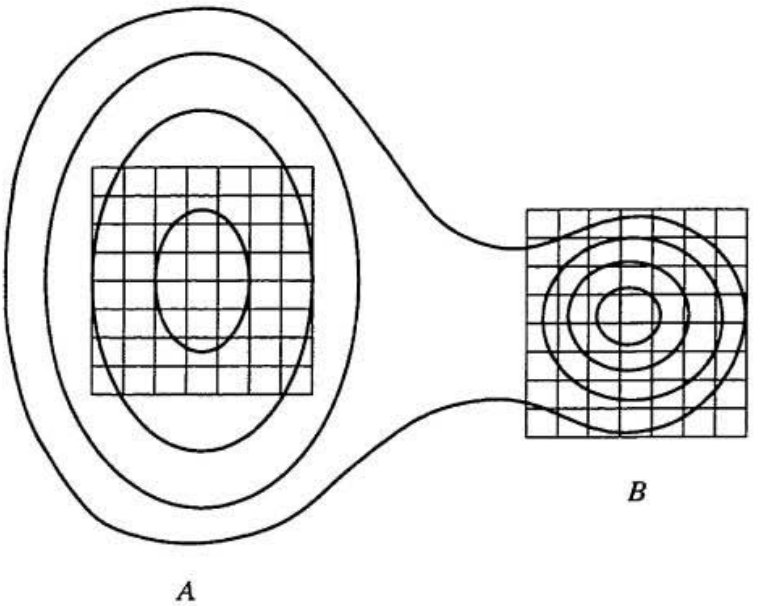


Figure 3.16: Contours of a sample surface to demonstrate the effect of the local geometric features of the objective function. The surface around maximum A has a more gentle slope than the surface around B which is a steep maximum.

small distance from the maximum at A will have a lower gradient than a point the same distance from the maximum at B. For the fitness function used, this will result in samples taken near A having a higher fitness value on average than samples taken near B, provided each has the same number of negative eigenvalues. This is reflected in Figure 3.15. However, taking the log of the fitness values (as was done throughout this chapter) results in the three bands of points collapsing together around the overall average line. Consequently, any difference in fitness values seen on a log plot are significant, and not due to landing on different stationary points.

3.6 Results for Minima

To illustrate the robustness of genetic algorithms for optimization problems, the fitness function was modified, while maintaining the same basic structure, to seek the minima of the Chong-Zak function. The behaviour exhibited was similar to that seen for saddle points. A scatter plot of individuals for a given run is shown in Figure 3.17, where the first minimum, m_1 is found at $(x, y) = (-1.431359, 0.206945), \|\vec{g}\| = 3.246533 \times 10^{-6}, n = 0$ in generation 99. The all-offspring replacement method was used here since it allows one to easily observe the convergence.

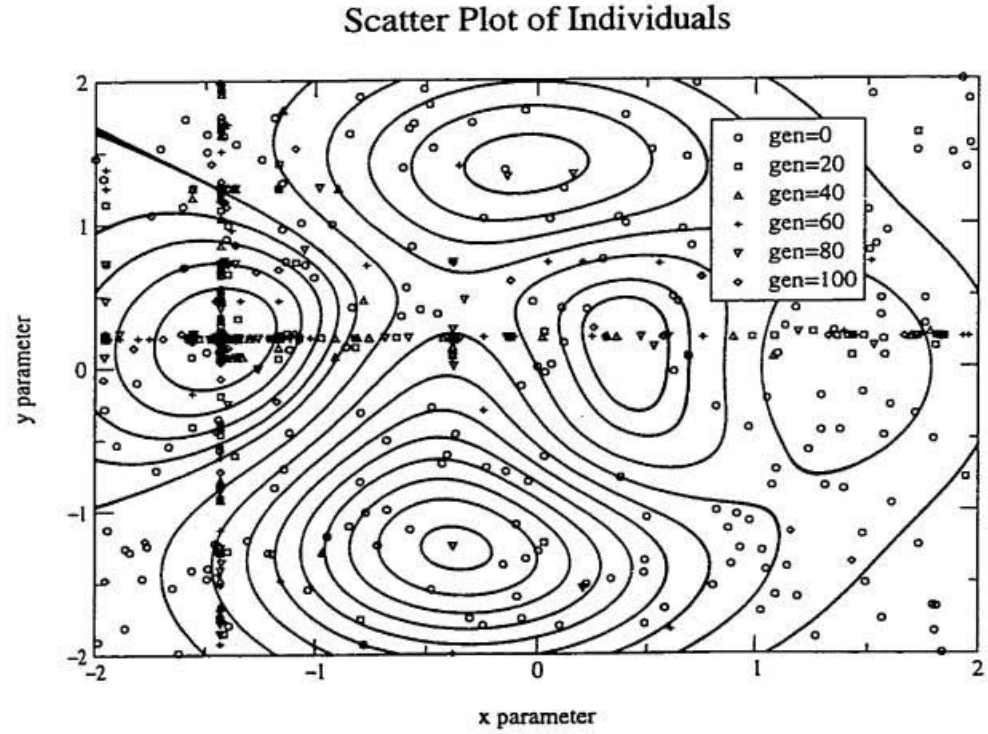


Figure 3.17: Scatter plot of individuals for $\mu = 200$, $B =$ 'b,' $R =$ 'o,' and $\lambda_{tol} = 0.0$, with the fitness function chosen to seek a minimum.

In another plot where the population size was decreased to 100, with B = 'g' and R = 'a' m_2 happened to be obtained. The optimum found was at $(x, y) = (0.404936, 0.166523)$, with $f(x, y) = -0.930778$, $\|\vec{g}\| = 1.703883 \times 10^{-6}$ in generation 59.

3.7 Conclusions and Recommendations

As a result of the above testing performed on the Chong-Zak function, the recommended parameter values for efficient use of this implementation of a genetic algorithm are: $\mu = 100$, $G_{max} = 200$, $p_c = 0.75$, $p_m = 0.03$, $n_b = 24$, S = 't,' $t_{size} = 6$, $t_{prob} = 0.75$, E = 'i,' B = 'g,' R = 'a.' It is concluded that these values gave the best results in general and provide a good starting point for the optimization of transition state structures[7]. However, given the large increase in computational complexity in moving from 2D mathematical functions to chemical structures, it is likely that $G_{max} = 200$ will result in long wait times. This is not likely to cause problems since most runs of the algorithm with B = 'g' showed little improvement after $G_{max} = 100$, hence 100 generations will be used as the starting point. Also, M_{init}, M_{sub}, and λ_{tol} are problem dependent and would therefore be very different for chemical structures. Note that a convergence criteria should be introduced since the possibility of obtaining a more accurate solution in subsequent generations should be balanced with the

[7]Note that these parameters are not completely independent, and some coupling exist between many or all of the parameters. Hence, a more thorough investigation of the effect of the parameter values on the behaviour of the algorithm would include many more combinations of possible values.

expense of continuing the computation. Thus, for chemical structures the convergence criteria, $||\vec{g}|| < \delta$ is introduced, where δ is user defined. We will choose $\delta \approx 5 \times 10^{-5}$ as the default value.

Chapter 4

Results From Chemical Structures

"No amount of experimentation can ever prove me right; a single experiment can prove me wrong."

-Albert Einstein

The transition from the optimization of mathematical functions to the optimization of chemical structures requires the consideration of several additional concepts. One consideration is the choice of coordinate system to use. Internal coordinates in the form of a Z-matrix was used as it provides the most intuitive structure description. In Z-matrix coordinates, one represents the molecule by the bond lengths between adjacent atoms, and the angles between adjacent bonds. Other common coordinate systems include Cartesian where each atom is simply assigned its position in xyz-space, and natural internal coordinates which represents a structure as combinations of bond lengths, bond angles, and torsions. In GA's, reducing the coupling between variables is preferred since this allows them to evolve independently and leads to a more efficient algorithm. Based on the amount of coupling between variables, Z-

matrix coordinates are a good choice. In comparison to Cartesian coordinates and natural internal coordinates, Z-matrix coordinates have less coupling than Cartesian coordinates, but natural internal coordinates provide nearly completely decoupled variables. Thus, future work on this project could possibly benefit from the use of natural internal coordinates.

Second, one must decide on the method of energy calculation used. The energies, first derivatives, and numerical second derivatives were calculated using an *ab initio* approach at the Hartree-Fock level, with the 3-21G basis set.

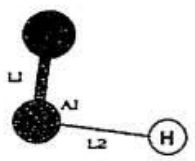
To demonstrate the viability of the genetic algorithm code, various transition state structures were selected for optimization. These choices were taken from a range of chemical reactions so as to sample some of the different chemical characteristics that can arise. The test cases shown in Table 4.1 were taken from the list provided by Baker and Chan [25] where a large variety of chemical reactions used in recent literature for testing transition state structure optimization methods are presented.

4.1 Physical Aspects of Transition State Structures

Recall that the defining characteristics of transition state structures, and thus the features that should be isolated in a genetic algorithm are; transition state structures are first order saddle points on a potential energy surface whose second derivative matrix has one and only one negative eigenvalue, the gradient vector at these points has a norm of zero. Additionally, the eigenvector corresponding to the negative

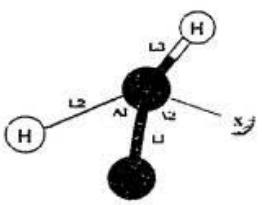
Table 4.1: Test cases used for transition state structure optimization (Bond lengths given in angstroms and bond angles in degrees). The starting geometries used for the current optimization are as shown in the form of a Z-matrix, and are the same as those given in [25].

1. HCN $\leftrightarrow$ HNC



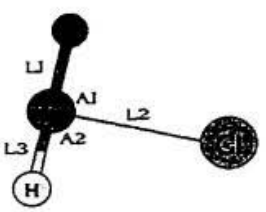
C1	L1	1.14838
N2 C1 L1	L2	1.58536
H3 C2 L2 C1 A1	A1	90.0

2. HCCH $\leftrightarrow$ CCH_2



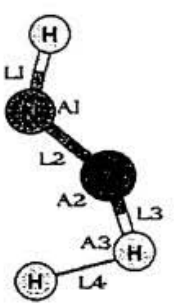
C1	L1	1.24054
C2 C1 L1	L2	1.65694
X3 C1 1.0 C2 90.0	L3	1.06318
H4 C1 L2 C2 A1 X3 180.0	A1	60.3568
H5 C1 L3 X3 A2 C2 180.0	A2	60.3568

3. HOCl $\leftrightarrow$ HCl + CO



O1	L1	1.17
C2 O1 L1	L2	2.335
Cl3 C2 L2 O1 A1	L3	1.127
H4 C2 L3 Cl3 A2 O1 180.0	A1	90.0
	A2	90.0

4. HNC + H_2 $\leftrightarrow$ H_2CNH



H1	L1	1.0
N2 H1 L1	L2	1.2
C3 N2 L2 H1 A1	L3	1.0
H4 C3 L3 N2 A2 H1 D1	L4	1.2
H5 H4 L4 C3 A3 N2 D2	A1	120.0
	A2	150.0
	A3	90.0
	D1	170.0
	D2	10.0

eigenvalue must be such that the deformation of the molecular structure in that direction along the surface connects the transition state structure to reactants and products.

4.2 Unique Features of the Genetic Algorithm

Unlike other transition state optimization techniques, the genetic algorithm approach is not sensitive to the structure of the initial Hessian. The algorithm promotes the production of individuals with one negative eigenvalue, favouring the correct Hessian eigenvalue structure.

Furthermore, genetic algorithms are known for their ability to efficiently sample a search space to locate a global optimum. Although this is not the intention in the current implementation it is worth noting that transition state structure optimization is less of a local search than the optimization of minima; for transition states. it is very unlikely that an initial guess can be made as close to the desired optima as is possible for minima.

4.3 Results

For the optimization of chemical structures, the following parameter values were used: $\mu = 100$, $G_{max} = 100$, $p_c = 0.75$, $p_m = 0.05$, $n_b = 31$, $S =$ 't', $E =$ 'i'. $B =$ 'g', $R =$ 'a', $\lambda_{tol} = 0.0$. Some of these values differ from the starting values proposed in Chapter 3, which reflects the problem dependency of genetic algorithms.

The mutation probability was increased from 0.03 to 0.05 since the former did not sufficiently sample the search space. The number of bits used for each variable, n_b was increased to 31[1] since, for interval encoding, the accuracy increases with more bits. Since the negative eigenvalues for chemical structures are usually very small compared to those for mathematical functions, λ_{tol} was changed to 0.0. Thus any negative eigenvalues, regardless of the size, are counted as negative.

Furthermore, the generation of the initial population was modified for interval encoding. Individuals are generated randomly within the intervals specified for the variables, with the initial guess added to the population without modification. This eliminates the need for the M_{init} and M_{sub} variables since all n_b-bit integers can be mapped into their corresponding intervals, and thus any subsequent individuals are guaranteed to be within the respective intervals.

The structures given in Table 4.1 were optimized with the VA method discussed in Chapter 1, and the results are compared to those obtained from optimizing with the genetic algorithm. Reaction 1 (see Table 4.1) is an HCN $\leftrightarrow$ HNC rearrangement and the results obtained are shown in Table 4.2. The bond lengths are reported in angstroms and the bond angles in degrees. The optimized structures obtained from both methods are very similar and both have a Hessian matrix with one negative eigenvalue and a gradient length on the order of 10^{-5}. The total energy, E_t is reported

[1]This is the maximum number of bits that can be used for each variable since a single unsigned integer is used, corresponding to 32 bits, and a single bit is reserved for the sign bit, as in multiplicative encoding, despite the fact that it is not used in interval encoding.

Table 4.2: Results obtained for the HCN $\leftrightarrow$ HNC rearrangement showing the initial geometry and the optimized geometries from the GA and VA methods. The Hessian has one negative eigenvalue (n) for all three structures, but the genetic algorithm structure has a slightly lower gradient length ($\|\vec{g}\|$).

Variable	Initial	GA	VA
L1	1.14838	1.18265	1.18269
L2	1.58536	1.40780	1.40741
A1	90.0	55.0267	55.0541
E_t	-92.20273	-92.24604	-92.24604
$\|\vec{g}\|$	7.53×10^{-2}	5.72×10^{-5}	7.30×10^{-5}
n	1	1	1

Table 4.3: Results obtained for the HCCH $\leftrightarrow$ CCH_2 rearrangement showing the initial geometry and the optimized geometries from the GA and VA methods. The Hessian has one negative eigenvalue (n) for all three structures, but the genetic algorithm structure has a slightly lower gradient length ($\|\vec{g}\|$).

Variable	Initial	GA	VA
L1	1.24054	1.24658	1.24645
L2	1.65694	1.42802	1.42920
L3	1.06318	1.05552	1.05565
A1	60.3568	54.1655	54.1117
A2	60.3568	86.6367	86.6471
E_t	-76.265417	-76.29343	-76.29343
$\|\vec{g}\|$	2.68×10^{-1}	1.50×10^{-4}	2.64×10^{-4}
n	1	1	1

in Hartrees and the initial and final values match those reported in [25]. This reaction provides a good visual example since it only has three variables so a cluster plot can therefore be produced. An example of such a cluster plot of individuals every 40 generations is shown in Figure 4.1. Note that the individuals cluster in a smaller and smaller region as the population evolves to the optimum.

Results for Reaction 2 are shown in Table 4.3. The two optimized geometries are

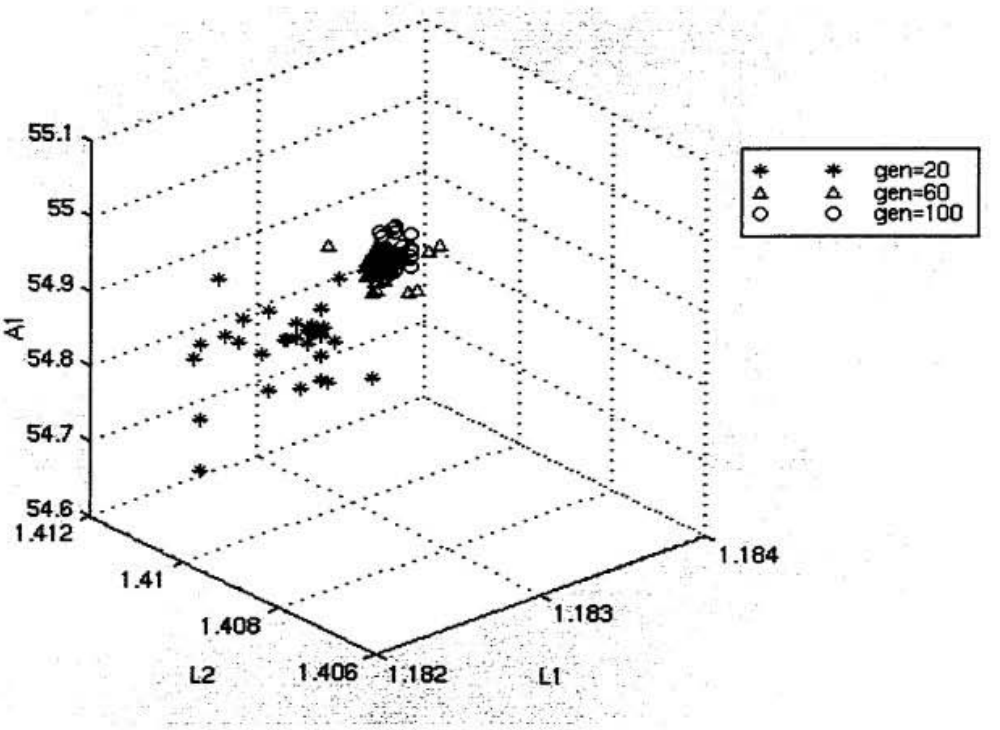


Figure 4.1: Cluster plot of individuals every 40 generations for the HCN $\leftrightarrow$ HNC reaction. The clustering of individuals gets tighter as the population evolves.

Table 4.4: Results obtained for the HOCl $\leftrightarrow$ HCl + CO reaction showing the initial geometry and the optimized geometries from GA and VA. The two optimized geometries are similar with comparable gradient lengths, with the VA gradient length slightly lower.

Variable	Initial	GA	VA
L1	1.17	1.11608	1.11607
L2	2.335	2.55176	2.55180
L3	1.127	1.10407	1.10418
A1	90.0	126.6372	126.6381
A2	90.0	47.6255	47.5872
E_t	-569.87865	-569.89752	-569.89752
$\|\|\vec{g}\|\|$	4.58×10^{-1}	4.73×10^{-5}	6.67×10^{-5}
n	1	1	1

again very similar, both having gradient lengths on the order of 10^{-4}. Again, the energies match those reported by Baker and Chan. Results obtained for Reaction 3 are shown in Table 4.4. There are only slight differences between the two optimized geometries and both converged with gradient lengths on the order of 10^{-5}. All three geometries have a single negative eigenvalue, and the energies listed agree with those in the original paper.

Results obtained for Reaction 4 are shown in Table 4.5. Again, both optimized geometries are similar. However, note that the initial geometry has two negative eigenvalues but both optimized geometries have just one. The energy of the optimized structures differ in the fifth decimal place, with the VA energy matching that reported by Baker and Chan.

Table 4.5: Results obtained for the HNC + $H_2 \leftrightarrow H_2CNH$ reaction showing the initial geometry and the optimized geometries from GA and VA.

Variable	Initial	GA	VA
L1	1.0	1.01305	1.01184
L2	1.2	1.21603	1.21292
L3	1.0	1.11342	1.11201
L4	1.2	1.14220	1.15952
A1	120.0	117.6791	118.7790
A2	150.0	152.2407	152.7408
A3	90.0	93.1678	93.9809
D1	170.0	181.1704	180.0548
D2	10.0	-1.2450	-0.0829
E	-93.30097	-93.31110	-93.31114
$\|\|\vec{g}\|\|$	2.97×10^{-1}	5.21×10^{-4}	2.55×10^{-4}
n	2	1	1

4.4 Further Modifications

The current version of the genetic algorithm code requires an extremely long run time when compared to traditional methods such as VA. Many possibilities exist for improving the computational expense and will be discussed in Chapter 5.

Also, for problems other than the optimization of first order saddle points, the above-average replacement strategy will likely require modification. In the current implementation, the pool from which individuals are chosen is formed from all of the parents and offspring whose fitness values are above the average fitness value of the offspring population. In the case where this elite pool comes from a small percentage of the population, as when a few exceptionally high fitness values bias the average fitness upward, this can lead to very fast convergence. However, care must be taken to avoid cases of premature convergence. Additionally, in the cases where this elite pool

number less than the size required to form a new population, selecting individuals only from this pool will cause duplication, limiting the gene pool and hence the effective sample size. In more practical terms, this duplication also causes wasted CPU time to be used on re-evaluating the energies and derivatives of duplicate individuals. A proposed modification for the case of very small elite pool sizes involves placing all of the above average individuals in the population and filling in the remainder (up to the population size μ) with individuals created by mutating elite-pool individuals. The case where the pool is sufficiently large to form a new population is not modified. Although the rapid clustering suits our problem, problems such as conformational searches will probably require a modification such as that proposed above to prevent clustering to a single small region of the surface.

Chapter 5

Conclusions and Future Work

"It is good to have an end to journey towards; but it is the journey that matters in the end."

-Ursula K. LeGuin

5.1 Genetic Algorithms in a Nutshell

Genetic algorithms manipulate a collection of potential solutions to a problem in parallel, rather than successively improving a single estimate of the optimum as is done in traditional methods. The algorithm works with the encoded form of these potential solutions rather than the solution values themselves, and operates on these encoded values with stochastic operators. Implementation of a genetic algorithm is problem dependent and each piece of software is sufficiently detailed to restrict it to solving only the type of problems for which it was written. Such algorithms which have been highly adapted for a specific problem are often more efficient at solving that problem, at the expense of generality.

A growing number of texts have been written in the area of genetic algorithms. many of an introductory nature with various applications. Some such texts used by the author include [26, 27, 28, 29, 30].

5.2 Genetic Algorithms and Transition State Structures

The work presented in this thesis has laid the foundation for ongoing research in the area of chemical structure optimization using genetic algorithms. Optimization of the Chong-Zak function with varying parameters demonstrated the behaviour of the method, providing a good testing medium, as well as a basis for optimizing chemical structures. The code written was able to find all three saddle points of the function, which illustrates the effectiveness of the fitness function used. Furthermore, the two minima of the Chong-Zak function were also found, which demonstrates the robustness of the genetic algorithm technique.

Applying the implementation to the optimization of chemical structures proved that it was able to efficiently sample the regions given and effectively find a transition state structure. Transition state structures for several chemical reactions were determined, in agreement with other optimization techniques.

5.3 Ideas For Future Work

Throughout this research, various ideas for decreasing the run time of the genetic algorithm as well as improving its convergence were discussed. Some of these ideas

that are not yet implemented are discussed in the following subsections.

5.3.1 Real Valued Encoding

It has been proposed that the best encoding scheme for a genetic algorithm is the representation that most closely reflects the normal representation of the data in solution space. For the current problem this would be the floating point representation of the real-valued variables. The direct use of real values as opposed to encoding via standard binary has been discussed by several authors. If the real-valued representations are used several components of the algorithm would have to be modified, and alternate forms of the genetic operators would have to be developed, for which several possibilities already exist. In the real-valued encoding scheme, the operators no longer rely on direct manipulation of bits, but rather on such things as random perturbations and variable swapping. Despite the research conducted in real-valued encoding, the genetic algorithm community has yet to warm up to the idea, and the majority of implementations still rely on some form of binary encoding. Although it is not clear that real-valued encoding would improve the performance of the current implementation, further inquiry may be a worthwhile venture.

5.3.2 *Ab initio* versus Molecular Mechanics Energies

The majority of the computational time required to run the genetic algorithm is taken up by the *ab initio* routines used to compute the energy, gradients, and

Hessians for each geometry. Since the use of molecular mechanics force fields is much less computationally intensive than *ab initio* methods, it is believed that using this method for energy, and derivative calculations would be beneficial. However, if molecular mechanics were to be used, the fitness evaluation of an individual would essentially be reduced to just a few simple function evaluations. One difficulty in this approach is the lack of available force fields for transition state structures compared to those available for minima.

5.3.3 Elimination of Expensive Derivatives

One way to eliminate expensive derivatives is to use molecular mechanics instead of *ab initio* calculations, as mentioned above. In molecular mechanics, derivatives are just simple function evaluations and are therefore computationally cheap. Another approach involves a type of interpolation strategy to avoid full calculation of the derivatives for a portion of the population. In this case, first and second derivatives would be calculated for a number of individuals and interpolation would be used to assign derivative values to those individuals lying near the individuals whose derivatives have been determined. Although this is an approximation strategy, it would likely provide sufficient bias toward the correct region of the potential energy surface. However, after a certain point, all derivatives must be calculated to complete the evolution to the optimum.

5.3.4 Hybrid Genetic Algorithms

Genetic algorithms were designed as global optimization techniques, that is, given a surface, find any (or all) of the optima, regardless of the features of the surface. This cannot be achieved with traditional methods since a single approximation of the optimum is used and such methods will often miss, or get stuck, in a local optimum. More often than not, this behaviour is not desired. Furthermore, most traditional methods will require an initial guess that is in the region near the optimum (with the correct Hessian structure) in order to converge. Satisfying this requirement is often very difficult. Thus, genetic algorithms provide benefits in these areas.

However, for the problem examined in this thesis, global search behaviour can cause problems with long run times. If one is able to provide a very good starting geometry or if the genetic algorithm can generate a good geometry within a reasonable number of generations, traditional methods may be able to help. Thus, it is proposed that a hybrid genetic algorithm would be a feasible approach. By first evolving a population for a number of generations such that the gradient length of the best individual is sufficiently decreased and the Hessian has one negative eigenvalue, one can use this best individual as the starting geometry for one of the traditional methods, such as Newton-Raphson. Since the initial guess is then in the region near the optimum, traditional methods should be able to converge to a transition state structure in a reasonable number of iterations.

Table 5.1: CPU time (in hours, minutes and seconds) required to optimize the chemical structures shown in Chapter 4 on a 600MHz Pentium III. These run times do not compare to the time required to optimize the same structures with the VA method, for which the run times were less than 5 minutes.

Reaction	# of Variables	# of Generations	Time (hh.mm.ss)
$HCN \leftrightarrow HNC$	3	100	19.04.05
$HCCH \leftrightarrow CCH_2$	5	100	30.57.33
$HOCl \leftrightarrow HCl + CO$	5	73	88.55.04
$HNC + H_2 \leftrightarrow H_2CNH$	9	100	75.09.45

5.3.5 Parallel Implementation

The genetic algorithm code lends itself well to parallel implementation, as is the case with most genetic algorithms. In the case of chemical structures, parallelizing the fitness evaluation would significantly decrease the wait time required for optimization, since the fitness evaluation is the most expensive component of the algorithm for this problem. Parallelization, in the present case, could be accomplished with minimal effort and would make good use of modern computing architecture.

5.4 Final Words

One important consideration in computational algorithms is the run time required. Typical examples of such times for the chemical structures optimized using the genetic algorithm are shown in Table 5.1. Although the genetic algorithm code written requires very long run times when compared to traditional optimization techniques, optimizing the code and using some of the ideas presented in this chapter, will help

to decrease this run time. Furthermore, the genetic algorithm was implemented with the idea of optimizing transition state structures that proved difficult (or impossible) to optimize with traditional methods, as well as to allow the flexibility of providing an initial guess far removed from the saddle point and still achieve convergence. The results of this thesis indicate that this is indeed possible.

Bibliography

[1] B. Murtagh and R. Sargent, Comput. J. **13**, 185 (1972).

[2] *Nonlinear Programming*, edited by K. Ritter (Academic Press, New York. 1970). pp. 31–65.

[3] W. Davidon, AEC Res. and Dev. Report ANL-5990 (revised) (1959).

[4] R. Fletcher and M. Powell, Comp. J. **6**, 163 (1963).

[5] C. Broyden, J. Inst. Maths. Appls. **6**, 222 (1970).

[6] R. Fletcher, Comp. J. **13**, 317 (1970).

[7] D. Goldfarb, Math. Comp. **24**, 23 (1970).

[8] D. Shanno, Math. Comp. **24**, 647 (1970).

[9] W. Davidon, Math. Prog. **9**, 1 (1975).

[10] R. Fletcher, *Practical Methods of Optimization, Volume 1* (John Wiley & Sons, Ltd., New York, 1980).

[11] J. Baker, J. Comp. Chem. **7**, 385 (1986).

[12] A. Banerjee, N. Adams, and J. Simmons, J. Phys. Chem. **89**, 52 (1985).

[13] P. Csaszar and P. Pulay, J. Mol. Struct. **114**, 31 (1984).

[14] C. Cerjan and W. Miller, J. Chem. Phys **75**, 2800 (1981).

[15] T. Helgaker, Chem. Phys. Lett. **182**, 503 (1991).

[16] J. Bofill, J. Comp. Chem. **15**, 1 (1994).

[17] J. M. Anglada and J. M. Bofill, J. Comp. Chem. **19**, 349 (1998).

[18] R. Poirier, Y. Wang, and C. Pye, Chemistry Dept., Memorial University of Newfoundland, St. John's, NF (1996).

[19] M. Powell, AERE Subroutine Library, Harwell, Didcot, Berkshire, UK .

[20] R. Judson, Reviews in Computational Chemistry **10**, 1 (1997).

[21] J. Mestres and G. E. Scuseria, J. Comp. Chem. **16**, 729 (1995).

[22] J. Holland, *Adaptation In Natural and Artificial Systems: An Introductory Analysis With Applications to Biology, Control, and Artificial Intelligence* (University of Michigan Press, Ann Arbor, Michigan, 1975).

[23] D. E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning* (Addison-Wesley, Reading, Massachusetts, 1989).

[24] E. Chong and S. Zak, *An Introduction To Optimization* (John Wiley & Sons, Ltd., New York, 1996).

[25] J. Baker and F. Chan, J. Comp. Chem. **17**, 888 (1996).

[26] D. A. Coley, *An Introduction to Genetic Algorithms for Scientists and Engineers* (World Scientific Publishing Co. Pte. Ltd., Singapore, 1999).

[27] A. M. S. Zalzala and P. J. Fleming, *Genetic Algorithms in Engineering Systems* (The Institution of Electrical Engineers, London, UK, 1997).

[28] K. F. Man, K. S. Tang, and S. Kwong, *Genetic Algorithms* (Springer-Verlag London Limited, Great Britain, 1999).

[29] *Genetic Algorithms and Simulated Annealing*, edited by L. Davis (Morgan Kaufmann Publishers, London, 1987).

[30] *Foundations of Genetic Algorithms*, edited by G. J. E. Rawlins (Morgan Kaufmann Publishers, California, 1991).

[31] O. Lendl, http://random.mat.sbg.ac.at (1997).

Appendix A

Code Documentation

The following is a discussion of the programming details for the genetic algorithm code written to optimize first order saddle points. A list of source code files is given, as well as a description of the various functions with some sample code.

A.1 Source Code Files

The source code files for the implementation of the genetic algorithm are as follows:

Table A.1: Source code files

File name	Description
mun_ga_globals.h	Contains all global variables.
mun_ga_data_structs.h	Contains the structure definition for an Individual, and various pointers to functions.
mun_ga_params.h	Contains the static parameters used.
mun_ga.c (.h)	Main file which controls the flow of the algorithm.
mun_ga_IO.c (.h)	Utilities for input and output.

mun_ga_breed.c (.h)	Contains the main reproduction function, functions for crossover and mutation, as well as functions for finding the best Individual in a group and validating Individuals.
mun_ga_encode.c (.h)	Contains functions for encoding and decoding real valued data using multiplicative or interval encoding. Also, functions for Gray encoding and decoding.
mun_ga_fitness.c (.h)	Contains functions for computing the fitness values of the initial population and the offspring produced, as well as utility functions to calculate the norm of a vector and to normalize the fitness values of a group of Individuals.
mun_ga_generate.c (.h)	Contains functions for the generation of the initial population, and for choosing Individuals to form the population for the next generation.
mun_ga_memory.c (.h)	Contains functions for allocating and deallocating memory.
mun_ga_select.c (.h)	Contains functions for roulette-wheel and tournament selection.
mun_ga_setup.c (.h)	Contains functions for initializing various function pointers, and functions for parsing the input file containing parameter values.
mun_ga_utils.c (.h)	Contains a function for copying all of the contents of an individual, as well as a function to flip a biased coin with a specified probability.

A.2 Data Structures

An `Individual`, representing a possible set of variable values is stored as a structure with definition,

```
struct individual{
```

```
    Chromosome *chrom;   /* bit string of encoded variables */
    double fitness;  /* fitness value */
    double normfit;  /* normalized fitness value */
    double *grad;    /* gradient vector */
    double func_value;  /* function value */
    double grdlth;   /* norm of the gradient vector */
    int negeig;      /* number of negative eigenvalues */
    int generation;  /* used only for the "best" individual */
} Individual;
```

located in mun_ga_data_structs.h, where a Chromosome is an unsigned integer. The `chrom` array stores the bit string, `fitness` is the value assigned to the `Individual` by the defined fitness function, and `normfit` is the value after the `fitness` has been normalized with respect to the current population. The gradient, `grad`, the function value, `func_value`, the norm of the gradient vector, `grdlth`, and the number of negative eigenvalues in the Hessian, `negeig`, associated with the given variable values are used to calculate the fitness value. The `generation` variable is used to monitor when the most fit `Individual` was formed.

A population of `Individuals` is stored as an array of `Individual` structures. In addition, an instance of the `Individual` structure is used to store the most fit `Individual` found in the evolution, this instance is referred to as `best`.

A.3 Input/Output

The parameter values used in the routine are read in from a file called **gadat**. This file has a specific arrangement assumed by the functions used to parse it. This assumed format is shown in Table A.2.

Table A.2: Input file format.

Variable	Type	Description
#	N/A	a comment
point_type	char	Type of point sought, m – minimum, s – first order saddle point
popsize	int	Number of Individuals
maxgen	int	Maximum number of generations
crossprob	double	Probability of crossover
mutprob	double	Probability of mutation
bits_per_var	int	Number of bits used per variable
paramscale	double	Maximum perturbation of an Individual in the initial population from the initial guess.
valid	double	Maximum perturbation of an offspring from the initial guess.
select_method	char	Which selection method to use: r – roulette wheel, t – tournament. If tournament selection is used, the following two additional parameters are needed: toursize (int) – number of Individuals in each tournament, and tourprob (double) – probability of selecting the most fit Individual.
encode_method	char	Which encoding scheme to use: m – multiplicative encoding, i – interval encoding. If interval encoding is used, the following additional parameters are needed: $\mathtt{lb}_i$ (double) – lower bound of interval, and $\mathtt{ub}_i$ (double) – upper bound of interval, for $i = 0 \ldots \mathtt{numvar}$ where numvar is the number of variables.
bin_rep	char	Binary representation used, g – Gray encoding, b – standard binary.
choose_newpop	char	Method used to choose Individuals for the next generation, a – selection from above average, o – all offspring.
tol	double	Value below which negative eigenvalues must lie to be counted as negative.
epsilon	double	Parameter used in fitness function.
gen_inc	int	Interval for writing **xy_gen_*** data files.

The functions used for parsing the input file are:

```
void file_format()
void corrupt_file(char *filename, int line, char *str)
int read_parameters(char *filename)
```

which are located in mun_ga_setup.c and are used to print the file format to standard out, interrupt execution in the event that file `filename` has the incorrect format (reporting the line `line`, where the error occurred, and the string `str` that was read), and read in the parameters from `filename`, respectively.

The output is written to files **stats**, and **history**. The former contains the average, maximum, minimum, and best-so-far fitness values. The latter contains information that allows one to trace through the execution history of the code, including `Individual` fitness information for each generation. The **history** file is printed only if `SAVEHIST` is defined. In addition, data files of variable values for each `Individual` are written out periodically. The file name format of these files is **xy_gen_***, where * is the generation number at the time of printing. The values for * include 0, `maxgen`, and all multiplies of `gen_inc` in between. Functions used for writing out various data include,

```
void print_stats()
void print_ind(Individual ind)
void print_chrom(FILE *fp,Chromosome *chrom,int chromlength)
void print_params(FILE* fp)
```

which are located in mun_ga_IO.c. These functions print the components of the **stats** file, print the values of the components in the `Individual ind` to the **history** file, print the chromosome, `chrom` (with length `chromlength`) bitwise to `FILE *fp`,

and print the values of the variables for each `Individual` in oldpop to `FILE *fp`, respectively.

A.4 Random Numbers

Genetic algorithm code relies extensively on the generation of random numbers. Software which generates pseudo-random numbers, **prng-2.2**, developed by Otmar Lendl [31] was used. This program provides various choices for the algorithm used to generate the random numbers, each of which has several parameters. The *explicit inversion congruential generator* was chosen for the current project. In addition, a function,

```
int flip(double prob)
```

located in mun_ga_utils.c, was implemented to flip a biased coin, returning 1 with a probability `prob`, otherwise returning 0.

A.5 Memory Allocation

Memory allocation is done in the following functions:

```
unsigned int* alloc_uintarr(int num)
double* alloc_doublearr(int num)
double** alloc_2Ddoublearr(int n, int m)
struct individual* alloc_pop(int num)
void alloc_memory()
void free_memory()
```

which are located in mun_ga_memory.c. The first two functions allocate arrays of num unsigned integers and doubles respectively. The third allocates a two dimensional, n × m array of doubles, and the fourth allocates an array of num Individual's to form a population. Finally, alloc_memory controls all memory allocation, calling the previous functions in turn. All memory is returned to the operating system with the free_memory function.

A.6 Initial Population

The initial population of possible solutions are created by either randomly perturbing the initial guess, parset (for multiplicative encoding), or by choosing random numbers within the intervals specified (for interval encoding). The following function, located in mun_ga_generate.c, creates the initial population and places the Individuals in the oldpop array.

```
void generate_population(double *parset){
   int i,j;
   double perturbation;
   double *temp;

   temp = (double*)calloc(numvar,sizeof(double));

   /* copy parset to first individual in oldpop */
   encode(parset,oldpop[0].chrom);

   /* initialize all popsize individuals */
   if(encode_method=='m'){
      for(i=1;i<popsize;i++){
         for(j=0;j<numvar;j++){
         /* get random number by which to scale initial parameters
          * to create new individual restrict random number between
```

```
            * -paramscale and paramscale */
              perturbation=paramscale*(2*prng_get_next(g)-1.0);
              temp[j] = parset[j]+perturbation;
           }
           /* encode individual created by random perturbations */
           encode(temp,oldpop[i].chrom);
        }
     else{
        for (i=1;i<popsize;i++) {
           for (j=0;j<numvar;j++) {
              oldpop[i].chrom[j]=(unsigned)(prng_get_next(g)*
                                  ((1<<bits_per_var)-1));
           }
        }
     }
     free(temp);
     return;
  }
```

The size of the perturbation is scaled to between -paramscale and paramscale since the initial guess is usually close to the optimum. Following the addition of a small perturbation to the initial parameters, the resulting Individual is encoded as discussed in the following section.

A.7 Encoding Scheme

Various possibilities for encoding schemes are available, and can be found in mun_ga_encode.c. For representation of real valued data the encoding functions include,

```
void mult_encode(double *real, unsigned *int_rep)
void interval_encode(double *real, unsigned *int_rep)
```

with corresponding decoding functions for restoration of the real valued data, `real` from the integer representation, `int_rep`.

For multiplicative encoding a predetermined required precision of 1.0×10^{-6} was set in mun_ga_params.h,

```
#define SHIFTAMT (1.0E6)
```

and is used to encode each `Individual` by multiplying each variable by this amount, giving an unsigned integer. This scale factor is removed by decoding during fitness evaluation and before reporting results. The number of bits required per variable is user defined by the `bits_per_var` parameter. Negative numbers are accounted for by setting the $(\texttt{bits\_per\_var})^{\text{th}}$ bit to 1. Each variable is represented by a single unsigned integer, of which the lower `bits_per_var` data bits + 1 sign bit are used. The following function encodes an `Individual` using multiplicative encoding.

```
void mult_encode(double *real, unsigned *int_rep){
   int index;
   unsigned temp;

   for(index=0;index<numvar;index++){
      temp = (unsigned)(fabs(real[index])*SHIFTAMT);
      /* if the value is negative, set the bits_per_var bit to 1 */
      if(real[index]<0){
         temp = temp | (1<<bits_per_var);
      }
      int_rep[index] = temp;
   }
   return;
}
```

Decoding of each `Individual` for the purposes of fitness evaluation reverses the above process, converting the previously encoded integers back to the original floating point

numbers by determining the integer value of the extracted lower bits_per_var bits, dividing by SHIFTAMT, and adding a negative sign if the (bits_per_var)th bit is set.

Interval encoding uses a simple mapping between real values in the corresponding domain and integer values.

```
void interval_encode(double *real, unsigned *int_rep){
   int i;
   for(i=0;i<numvar;i++){
      /* add 0.5 to temp to ensure rounding up */
      int_rep[i] = (unsigned)((real[i]-domains[2*i])*
         ((1<<bits_per_var)-1)/(domains[2*i+1]-domains[2*i])+0.5);
   }
   return;
}
```

Note that special consideration for negative numbers is not required for the interval encoding scheme.

In addition to the integer representation of real numbers, Gray encoding is provided as an alternative to standard binary. For this case, the real numbers are encoded using one of the above functions, after which they are converted to Gray code using the following function which was derived from [26].

```
void gray_encode(Chromosome *bin){
   int i,j;
   unsigned mask;
   unsigned *gray;
   gray = (unsigned*)calloc(numvar,sizeof(unsigned));

   for(i=0;i<numvar;i++){
      mask = (1<<(bits_per_var-1));
      gray[i] = bin[i] & mask;
      for(j=0;j<bits_per_var;j++){
         if(((bin[i] & mask)>>1) == (bin[i] & (mask>>1))){
            mask = mask >> 1;  /* leave gray bit as 0 */
```

```
            }
            else{
               mask = mask >> 1;
               gray[i] |= mask;  /* set gray bit to 1 */
            }
         }
         /* set the sign bit */
         if (bin[i]&(1<<bits_per_var))
            gray[i]=gray[i]|(1<<bits_per_var);
      }
      /* copy the contents of gray to the individual passed in */
      for(i=0;i<numvar;i++){
         bin[i] = gray[i];
      }
      free(gray);
      return;
   }
```

with a corresponding decoding function.

The overall encoding and decoding processes are controlled by the functions.

```
void encode(double *data, Chromosome* ind)
void decode(Chromosome *ind, double *decoded)
```

which call the correct encoding and decoding functions using the function pointers `to_int` and `to_float` respectively which are set in the function,

```
void set_encode()
```

located in mun_ga_setup.c, based on the input parameter `encode_method`. Furthermore, the main encode and decode functions call the Gray encode and decode functions if the input parameter `bin_rep` is set to 'g.'

A.8 Fitness Evaluation

Following the generation and encoding of the `Individuals` in the initial population a numerical fitness value is calculated for each `Individual`. The fitness function used depends on the type of critical point sought and is set using the function,

```
void set_fitness()
```

located in mun_ga_setup.c, which uses the value of the `point_type` input parameter to set a pointer to one of the following,

```
void ts_evaluate(Individual *ind,int numneg)
void min_evaluate(Individual *ind,int numneg).
```

which are located in mun_ga_fitness.c. Each `Individual ind`, in the population is then decoded, the function value, gradient, and number of negative eigenvalues (numneg) is calculated, and a fitness value is assigned. This procedure is done in the function,

```
void init_fitness()
```

which calls the appropriate evaluation function. The gradient length is calculated using the function,

```
double calc_norm(double *vec, int n)
```

where `n` is the dimension of the vector `vec` for which the length is to be calculated. Following reproduction, the fitness of the offspring created is calculated in the function,

```
void offspring_fit()
```

which behaves similar to the fitness evaluation for the initial population. Finally, a function,

```
void normalize(Individual *pop)
```

calculates the maximum, minimum, and average fitness values in a population pop, as well as normalizes the fitness values between 0 and 1 if roulette-wheel selection is used, since this is the only function that makes use of normalized values.

A.9 Reproduction

The reproduction operators are controlled by the function,

```
void breed(double* initial)
```

which calls the selection, crossover, and mutation functions to reproduce the population, as well as the validate function (in the case of multiplicative encoding),

```
void validate(Individual* test, double* initial)
   int i=0,j=0;
   int changed=0;
   double perturbation;
   double *temp;
   temp = (double*)calloc(numvar,sizeof(double));

   for(i=0;i<popsize;i++){
      decode(test[i].chrom,temp);
      for(j=0;j<numvar;j++){
         if(fabs(temp[j]-initial[j])>valid){
            perturbation=paramscale*(2.0*prng_get_next(g)-1.0);
            temp[j] = initial[j] + perturbation;
            changed=1;
         }
      }
      /* re-encode individuals that had parameters changed */
```

```
        if(changed){
            encode(temp,test[i].chrom);
            changed=0;
        }
    }
    free(temp);
    return;
}
```

to ensure that the offspring produced (`test`), is within `valid` of the initial guess. `initial`. The functions that perform the reproduction operations are discussed in the following sections.

A.9.1 Selection

The two possible methods for selecting parents include roulette-wheel and tournament selection. The method used is user defined by the `select_method` input parameter, and the function,

```
void set_select()
```

sets the function pointer to the correct selection function. The function that performs roulette wheel selection is,

```
int roulette_select(){
    int i;
    double r;
    double sum;
    double totalfit=0.0;

    /* sum up the total fitness...*/
    for(i=0;i<popsize;i++){
        totalfit += oldpop[i].normfit;
    }
```

```
    r = prng_get_next(g);
    sum = 0;
    if(totalfit != 0){
       for(i=0;i<popsize;i++){
          sum += oldpop[i].normfit / totalfit;
          if(sum > r) break;
       }
    }
    else{  /* just pick a random parent if totalfit=0 */
       i = (int)(prng_get_next_int(g) % popsize);
    }
    return i;
}
```

which, as noted before, makes use of the normalized fitness values. The function that performs tournament selection is,

```
int tour_select(){
    int i;
    int *parent;
    int tour_best;
    int winner;
    double r;
    parent = (int*)calloc(toursize,sizeof(int));

    for(i=0;i<toursize;i++){
       parent[i] = (int)(prng_get_next_int(g) % popsize);
    }
    /* find best individual in current tournament */
    tour_best = parent[0];
    for(i=1;i<toursize;i++){
       if(oldpop[parent[i]].fitness > oldpop[tour_best].fitness){
          tour_best = parent[i];
       }
    }
    /* get random number for comparison to tourprob */
    r = prng_get_next(g);
    if(tourprob > r){
       /* choose the best individual in the tournament */
       winner = tour_best;
    }
```

```
    else{ /* choose a random individual in the tournament */
       winner = parent[(int)(prng_get_next_int(g) % toursize)];
    }
    return winner;
}
```

which uses user defined values for toursize and tourprob to bias the selection process. After two parents are selected for reproduction, crossover is performed.

A.9.2 Crossover

The crossover operation is performed by the following function,

```
void crossover(Chromosome *parent1, Chromosome *parent2,
               Chromosome *offspring1, Chromosome *offspring2){
/* This function is derived from SGA-C:  A C-language
 * Implementation of a Simple Genetic Algorithm, Robert
 * E. Smith, David E. Goldberg, and Jeff A. Earickson,
 * TCGA Report No. 91002, 1994. */

   int i,j;
   unsigned mask;
   unsigned temp=1;
   int crosspoint;

   /* determine whether or not to perform crossover */
   if(flip(crossprob)){
      crosspoint = (int)(prng_get_next_int(g) % chromlength);
      for(i=0;i<numvar;i++){
         if(crosspoint >= ((i+1)*UNSIGNEDSIZE)){
            /* crosspoint not reached yet, so swap these ints */
            offspring1[i] = parent1[i];
            offspring2[i] = parent2[i];
         }
         else if((crosspoint<((i+1)*UNSIGNEDSIZE)) &&
                 (crosspoint>(i*UNSIGNEDSIZE))){
            mask = 1;
            for(j=0;j<(crosspoint+1-i*UNSIGNEDSIZE);j++){
```

```
                mask = mask <<1;
                mask = mask | temp;
            }
            offspring1[i] = (parent1[i] & mask) |
                            (parent2[i] & (~mask));
            offspring2[i] = (parent1[i] & (~mask)) |
                            (parent2[i] & mask);
        }
        else{
            offspring1[i] = parent2[i];
            offspring2[i] = parent1[i];
        }
    }
}
else{  /* do not crossover, just copy parents to offspring */
    for(i=0;i<numvar;i++){
        offspring1[i] = parent1[i];
        offspring2[i] = parent2[i];
    }
}
}
```

which crosses over the bit strings of `parent1` and `parent2`, at `crosspoint` to produce `offspring1` and `offspring2`, with user input probability `crossprob`. Note the use of the UNSIGNEDSIZE parameter, set in mun_ga_params.h as,

```
#define UNSIGNEDSIZE (8*sizeof(unsigned int))
```

which is equal to the number of bits in an unsigned integer. Following crossover. the offspring produced are subjected to mutation.

A.9.3 Mutation

The chromosome string of an `Individual` is mutated using the following function,

```
void mutate(Chromosome *c){
```

```
    int i,j;
    unsigned mask=0;
    unsigned temp = 1;

    for(i=0;i<numvar;i++){
       mask = 0;
       for(j=0;j<bits_per_var;j++){
          if(flip(mutprob)){
             mask = mask | (temp<<j);
          }
       }
       c[i] = c[i]^mask;
    }
    return;
}
```

which uses the XOR logical operator to flip the state of a bit with a user input probability `mutprob`.

A.10 Tracking the Optimum

At the start of the algorithm, the `best Individual` is set to the initial guess for comparison purposes, and after the generation of the initial population, the following function is called to determine the most fit `Individual`.

```
void find_best(Individual *current_pop){
   int i;
   int bfi=-1; /* index of individual with best fit */
   double tmpfit;

   tmpfit=best.fitness;
   for(i=0;i<popsize;i++){
      if(current_pop[i].fitness > tmpfit) {
         bfi=i;
         tmpfit=current_pop[bfi].fitness;
      }
```

```
    }
    if (bfi==-1) {
        /* no better fitness was found in this population! */
        return;
    }
    /* update best individual structure */
    copy_ind(&(current_pop[bfi]),&best);
    best.generation = generation;
    return;
}
```

The Individual with the highest fitness is copied into best using the function,

```
void copy_ind(Individual *ind_source, Individual *ind_dest)
```

located in mun_ga_utils.c, which copies each component of Individual, ind_source to the corresponding component of ind_dest.

If an Individual with a higher fitness than that found previously is not present then the best Individual remains unchanged. Therefore, best always contains the Individual with the highest fitness throughout the evolution and is reported as the optimum when the algorithm is complete.

A.11 Replacement of the Population

Upon completion of reproduction, a portion of the parents and offspring can be chosen for the next generation in one of two ways. The first method is implemented in the following function,

```
void all_offspring(){
   int i;
   for(i=0;i<popsize;i++){
      copy_ind(&(offspring[i]),&(newpop[i]));
```

```
    }
    return;
}
```

located in mun_ga_generate.c, which copies all of the offspring created in the current generation into the population for the next generation, none of the parents continue on. The second method is shown in the function,

```
void above_average(){
    int i,r;
    int numold=0;
    int numnew=0;
    int *subpop;
    subpop = (int*)calloc(2*popsize,sizeof(int));

    /* create pool of individuals from oldpop and offspring
     * consisting of those individuals whose fitness values are
     * greater than the average from the previous generation */
    for(i=0;i<popsize;i++){
        if(oldpop[i].fitness > avg){
            subpop[numold] = i;
            numold++;
        }
    }
    for(i=0;i<popsize;i++){
        if(offspring[i].fitness > avg){
            subpop[numold+numnew] = i;
            numnew++;
        }
    }
    /* subpop complete...add best individual from previous
     * generation to new population */
    copy_ind(&(best),&(newpop[0]));

    /* choose random individuals from subpop to occur in the
     * next generation, start at 1 since best individual is in 0 */
    for(i=1;i<popsize;i++){
        r = (int)(prng_get_next_int(g) % (numold+numnew));
        if(r < numold){
            /* chosen individual came from previous generation */
```

```
            copy_ind(&(oldpop[subpop[r]]),&(newpop[i]));
        }
        if(r >=numold){
            copy_ind(&(offspring[subpop[r]]),&(newpop[i]));
        }
    }
    free(subpop);
    return;
}
```

also located in mun_ga_generate.c, which fills the surviving population with a random selection from among those parents and offspring with fitness values above the average fitness of the offspring. Which of the above functions is used is dependent on the user defined `select_newpop` parameter which is used in,

```
void set_choose_newpop()
```

located in mun_ga_setup.c, to set a function pointer to the correct population replacement function.

A.12 Central GA Control

The flow of the genetic algorithm is controlled by a main function,

```
void ga_(double parset[ ], double pargrd[ ], int noptpr,
         double grdlth, double func_value)
```

which is called from a Fortran subroutine within Mungauss. The core Mungauss operations are used to calculate the objective function value, gradient, and Hessian for the `Individuals`. The genetic algorithm code calls the appropriate Fortran functions within Mungauss to obtain this information when it is required.

A.13 Code Availability

Information about the above source code can be obtained by contacting one of the following:

Dr. R. A. Poirier - rpoirier@mun.ca
Sharene Bungay - sharene@math.mun.ca