

The Usefulness of Genetic Algorithms in Optimizing Ill-behaved Objective Functions

Scott R. Guirlinger

Since they were first extensively studied in the sixties, genetic algorithms have demonstrated their innate ability to optimize highly discontinuous, noisy, and non-linear objective functions, particularly those previously found to be unsolvable by traditional means. Through mimicking the processes that biological evolution induces upon chromosomes, genetic algorithms can evolve vast sets of potential solutions in parallel, randomly selecting, mutating, and recombining their data to practically always produce at least one very good solution to the problem at hand. Even though the initial solution set of a genetic algorithm is typically generated randomly, the end result of the simulation process after many iterations, or generations, is distinctly non-random (better than random). While genetic algorithms will never replace traditional optimization methods, it can be used very effectively as both an alternative to and in conjunction with gradient-based means.

Keywords: Genetic algorithm, generation, fitness function, fitness-proportionate selection, rank-based selection, roulette wheel selection, tournament selection, mutation, mutation rate, recombination, crossover, crossover rate

1 Introduction

Traditional optimization algorithms are gradient-based, designed to work with continuous variables only. When optimization of structured systems involves many parameters that interact in highly discontinuous, noisy, or non-linear ways, the gradient-based approach typically will not suffice, as the algorithm tends to become stuck on a sub-optimal solution. Specifically, objective functions that are characterized by numerous local optima, vast planes in multi-dimensional space, or points at which gradients are undefined all pose problems for traditional optimizers [1]. A common example of such an ill-behaved solution space is seen in the optimization of manufacturing or procurement functions where discrete variables such as stock or

inventory levels cause countless discontinuities in the governing functions themselves.

Genetic algorithms can combat these issues, however, by being one of the few optimization algorithms that can work directly with discrete design variables. They inherently solve optimization problems by using natural selection, the process that drives biological evolution, repeatedly modifying a population of individual solutions. At each step, a genetic algorithm selects individuals at random from the current population to be parents and uses them produce the children for the next generation, "evolving" toward an optimal solution. Compared to traditional optimizers, genetic optimizers are actually more likely to find the best overall (globally optimal) design. As will be shown in this

paper, they have become the optimization procedure of choice in today's world of technology for solving complex systems over noisy and discrete design spaces.

2 Brief History

Genetic algorithms first began to appear in the late 1950's and early 1960's with biologists seeking to programmatically model the process of natural evolution. Shortly thereafter, researchers began applying "evolution-inspired algorithms" to function optimization and machine learning problems. The first development worthy of note was Ingo Rechenberg's evolution strategy technique, in which one individual mutated to produce another and the better of the two was kept for the next iteration [2]. The use of a population or group of individuals in genetic algorithms was introduced a few years later.

John Holland began his work with genetic algorithms in the early 1960's and is credited with first proposing the idea of a recombination or crossover operator. His landmark achievement came in 1975 with the publication of his book entitled *Adaptation in Natural and Artificial Systems*. This book finally detailed how the evolutionary processes of selection, mutation, and crossover could be combined to form a highly effective problem-solving strategy [3].

Also in 1975, Kenneth De Jong completed his dissertation. In his research, he was able to illustrate how genetic algorithms could be used to solve a vast assortment of functions, particularly noisy, discontinuous, and non-linear functions [2], providing much of the motivation for the use and development of genetic algorithms for years to come.

Over the last few decades, the increase in computing power and efficiency has allowed the use of genetic algorithms to become more widespread. They have been

applied in numerous areas to subjects as diverse as structural optimization, pattern recognition and classification, stock market prediction and portfolio planning, and scheduling at airports and assembly lines.

3 Definition of Genetic Algorithm

Genetic algorithms use Darwin's concept of natural selection or "survival of the fittest" along with other evolutionary processes to continually produce iterations, commonly called *generations*, of solutions that are better than their predecessors [4]. A common use for genetic algorithms is in optimization problems, where one seeks to either maximize or minimize a parameter or set of parameters. Indeed, the phrase "survival of the fittest" inherently implies a maximization procedure [1]. (Because minimizing a function, say $f(x)$, is the same as maximizing its inverse, $-f(x)$, only the maximization process will be referred to throughout this paper.)

Because of the complex nature of today's business and engineering problems, genetic algorithms are practically always designed to work on populations of solutions rather than single solutions. The benefit of a population of solutions being solved in parallel is discussed in more detail later. The initial candidate solutions for a genetic algorithm may be selected intelligently with some prior knowledge of the system at hand, but most often the input to the algorithm is a pool of randomly selected candidates [2].

Solutions are typically represented as binary strings, arrays of numbers, or strings of alphanumeric characters [2]. These types of representations allow for multiple pieces of information concerning specific aspects of each solution to be stored all in one place. When these solutions are operated upon using one of the methods discussed later, each number or letter can be individually toggled or changed in an effort to produce a better solution. These strings or arrays

representing candidate solutions do not necessarily have to be of fixed length, although the majority of work with genetic algorithms is focused on fixed-length character strings [5]. The decision to fix or vary their length must be made based on the problem at hand.

To solve any optimization problem, an objective function must be defined. The objective function combines all relevant population characteristics or parameters into one function that is then maximized (or minimized). In the case of genetic algorithms, this objective function is often referred to as the *fitness function* since it is used continually to evaluate the ability of potential solutions to survive. During the development of a new generation, each individual or potential solution will be assigned a fitness score that can be used to determine if that individual survives or carries on to the new generation and/or if that individual is selected for recombination. So when a fitness function is being decided upon, care must be taken to ensure that higher fitness scores are given to individuals that are closer to solving the problem at hand [6]. Various methods of selection based on the fitness function are discussed further in the following section.

Once a system's fitness function has been defined, a genetic algorithm for the system is ready to begin. Most often the algorithm is implemented in the cyclical manner illustrated in Figure 1. The first generation consists of a population of randomly generated individuals. The fitness of the population is evaluated and if this generation does not satisfy the optimization criteria, the individuals in the population are selected for survival based on their fitness. A new population is then created by performing operations such as recombination and mutation on these selected individuals. The old population is discarded and another iteration of the

genetic algorithm is performed using the new population [5]. The various methods of selection, operation, and reinsertion are discussed at greater length in the sections that follow.

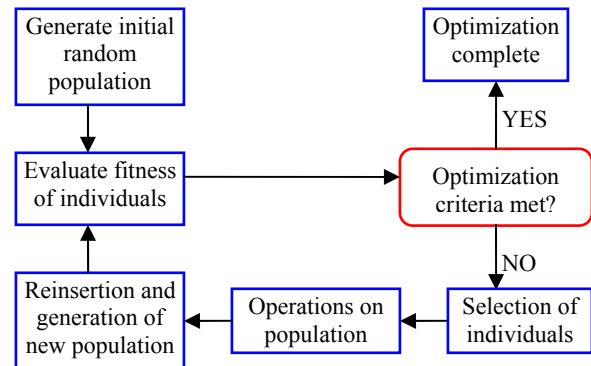


Figure 1: General structure of a single population genetic algorithm.

4 Methods of Selection

The first task in populating a new generation is to select which individuals will be operated upon to create this new generation. In all methods of selection researched here, the fitness of each individual determined its likelihood of selection. For the purpose of this paper, the various methods of selection will be divided into two general categories: *fitness-proportionate selection* and *rank-based selection* [4].

4.1 Fitness-proportionate Selection

In fitness-proportionate selection, each individual's fitness is divided by the overall total fitness of the entire population to determine that individual's probability of selection. In its purest form, this selection technique works in such a way that more fit individuals are more likely to be chosen, but not guaranteed. There are several derivatives of this technique and it should be noted that in practice more than one technique is often used to prevent the algorithm from biasing its selection and

thereby losing some of the diversity of the population.

Elitist selection is one variation of fitness-proportionate selection where the most fit members of a generation are indeed guaranteed to be selected [2]. This technique is often used in conjunction with others to ensure that the best solutions from each generation are carried on to the next generation.

Another fitness-proportionate selection model is called *thresholding*, in which individuals with fitness scores above some threshold are selected to reproduce. In a rare case when no individuals meet this threshold, the population can go extinct. For this reason, thresholding is also often only used in conjunction with other techniques.

One of the most commonly used selection methods is *roulette wheel selection* where a point is conceptually selected at random on a roulette wheel with each potential solution being given a slice of the wheel proportionate to its fitness score [2]. If the selected point is in a given individual's slice of the wheel, then that individual is selected and moves on.

As potential solutions are improved with each generation, the disparity between them often lessens; they begin to converge to one or more optimal solutions. Selections made later in the genetic algorithm must be done on a different scale than they were earlier. Indeed, a *scaling selection* technique is sometimes used to counteract this phenomenon. In general, scaling selection makes the fitness function become more discriminating as the average fitness of the population increases [2].

4.2 Rank-based Selection

Unlike fitness-proportionate selection, rank-based selection does not use probabilities. Instead, all individuals in a population are sorted according to their fitness. In such a way, rank-based selection can overcome the

potential scaling problems of fitness-proportionate selection [4].

It should be noted that some of the variations on fitness-proportionate selection could still carry over to rank-based selection, however. An elitist selection technique can be used to ensure that all of the highly ranked individuals are selected or a roulette wheel selection can be conducted where the size of the wheel slices is determined by an individual's rank, not their proportional fitness.

Another common selection technique that falls more under the realm of rank-based selection is *tournament selection*. In tournament selection, individuals are compared pair wise or in smaller subgroups and the most fit (highest ranked) individual from each pair or subgroup is selected for operation [2]. This method can help to encapsulate some of the natural effects of locality and neighborhood. There are also other less commonly used methods of selection that are more specifically tied to this premise of location but will not be discussed here.

5 Common Operators

Once individuals or potential solutions have been selected as parents, they must produce offspring to populate a new generation for the genetic algorithm. (In this section, individuals are sometimes referred to as chromosomes because of the biological influences on the operators.) The two operators that are almost always used in every genetic algorithm are recombination and mutation.

Other natural processes such as migration and diffusion that have a significant impact on evolution can also be implemented into genetic algorithms [4]. These processes are not discussed within the scope of this paper, however.

5.1 Recombination

Recombination is an evolutionary process in which two parent chromosomes combine to form one offspring chromosome containing attributes of each parent. This process is sometimes referred to as *crossover* in the context of genetic algorithms “because of the way genetic material crosses over from one chromosome to another” [5].

Recombination is analogous with sexual reproduction, since it requires two parent chromosomes to produce one new offspring chromosome and each characteristic of the offspring comes from one of the two parents [1].

Two common forms of recombination are single-point crossover and uniform crossover. In single-point crossover a common point of exchange between the two parent chromosomes is set at a randomly selected location. The offspring chromosome adopts all coding before the common point from one parent and all coding after the common point from the other [2]. An example of single-point crossover is illustrated in Figure 2.

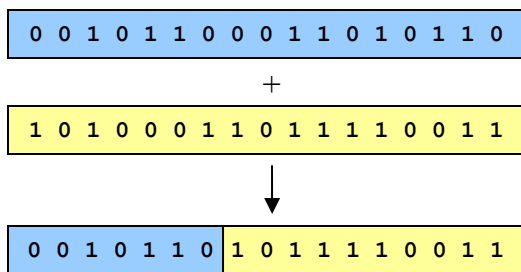


Figure 2: Example of binary single-point crossover.

In uniform crossover, the coding at any given location is chosen with 50/50 probability from either parent’s coding at that same location [2]. An example of uniform coding is shown in Figure 3.

Since not all chromosomes are selected for recombination, there must be some parameter that at least loosely controls how many chromosomes are selected in each

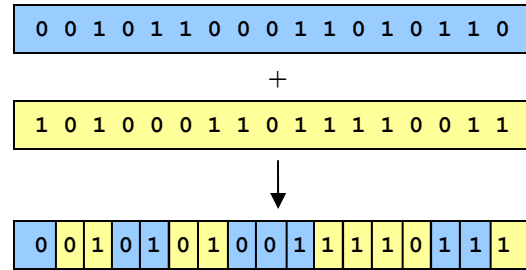


Figure 3: Example of binary uniform crossover.

population. This parameter is typically referred to as the *crossover rate* and is taken as an input to the overall genetic algorithm. While the rate may vary programmatically throughout the algorithm, it is often more effective to simply fix the rate at the onset of evolution. A common value for the crossover rate is around 0.7 [6], meaning that 70% of all individuals in a given population will be selected for recombination, a reasonable approximation of reality. As will be seen shortly, recombination is the driving operator behind any genetic algorithm, able to “build upon the success of the past, yet still explore new areas of the search space” [1].

5.2 Mutation

In contrast to recombination, *mutation* is the evolutionary process in which the genetic information of one parent is altered in some way to produce one offspring. Because only one parent is involved in this procedure, mutation can be likened to an asexual reproduction process [1].

Mutation specifically alters single points in a chromosomes coding. While multiple points could potentially be changed in a single process, the likelihood of this occurring is quite remote. Altering multiple points also tends to be inefficient and hinder the genetic algorithm in its search for an optimal solution because it changes too much of what is usually known to be a

favorable solution. A simple example of the mutation process is given in Figure 4.

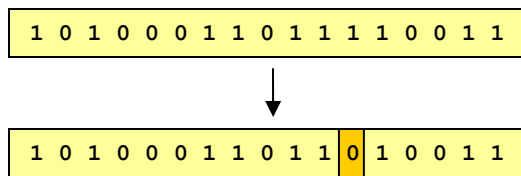


Figure 2: Example of binary mutation.

Much like the crossover rate for recombination, mutation must also be controlled by some parameter that determines how many chromosomes are selected for its operation. This parameter, called the *mutation rate*, is also taken as an input to the algorithm. The mutation rate may also be allowed to vary as the algorithm progresses, but most often is set to a value around 0.001 [6]. This value is set to be far lower than the crossover rate in order to mimic reality in which mutation only plays a relatively minor role in a given population. Indeed, the purpose of mutation is typically not to drive the development of the population but more often “to restore the genetic diversity lost during the application of [selection] and crossover” [1].

6 Methods of Reinsertion

After the individuals in a selected population have undergone operations such as recombination and mutation, they must be reinserted back into the previously evaluated population. After reinsertion, the newly created population replaces the previous population for the next iteration of the genetic algorithm. The two methods most commonly used for reinsertion are generational reinsertion and steady-state reinsertion.

It is important to note that any of the methods of selection discussed earlier could also be used here as methods of reinsertion. This is because reinsertion is simply the

selection of individuals to join the new population. (Selection was defined earlier more specifically as the selection of individuals for operation.) One should also note that several other reinsertion techniques such as global reinsertion and local reinsertion do exist beyond those mentioned here [4].

6.1 Generational Reinsertion

Often for simplicity and/or storage space, it is desirable to discard the previous iteration’s population and replace it entirely with the newly created population. This method is called *generational reinsertion* and permits no individuals to be retained between generations [2].

6.2 Steady-State Reinsertion

Because there is certainly a chance that any parent may be better than its offspring, it may be more desirable to retain some individuals between generations. In *steady-state reinsertion*, the offspring from each generation go back into the pre-existing population, “replacing some of the less fit members of the previous generation” [2]. In this way, the best members of both the pre-existing and newly created generations are retained to form the current population. Because the number of members in the population remains relatively stable in this method, it is deemed to be “steady-state”.

7 Benefits of Genetic Algorithms

So what advantages to genetic algorithms provide over more traditional optimization methods? First, where most other algorithms are serial in nature, genetic algorithms intrinsically work with many solutions in parallel. They are able to explore the solution space in multiple directions at one time [2], and therefore stand a better chance at finding the true global maximum of the system on the first try. When traditional algorithms begin to

diverge away from the optimal solution, there is often no way to correct their path. They are extremely reliant upon their initial conditions, as different starting points may lead to different end solutions. Genetic algorithms on the other hand are far less sensitive to the initial conditions imposed upon them [1], as they will eventually discard any solution that does not show enough promise. This feature helps to provide for more flexibility, robustness, and simplicity of design.

Due to their parallel nature, genetic algorithms are also much more efficient at navigating vast search spaces than traditional algorithms. For example, engineers at General Electric and Rensselaer Polytechnic Institute developed a genetic algorithm that produced a high-performance jet engine turbine design after navigating a solution space of more than 10^{387} possibilities. In comparison to a conventional design with less than half the performance that took several years and over \$2 billion to develop, the genetic algorithm arrived at its solution after only two days on a typical engineering workstation [2].

As was mentioned earlier, genetic algorithms have truly found their niche in their ability to find effective solutions even over complex solution spaces – ones with very noisy, discontinuous, and non-linear fitness functions. Traditional gradient-based algorithms are limited to finding local optima, whereas genetic algorithms are less likely to become stuck on local optima and can often find solutions much closer to the global optima. It should be noted here that there is usually no way to tell if an optima is the one true global optima, but that genetic algorithms will still almost always produce a relatively very good solution to the problem at hand [2].

To this point it may seem that genetic algorithms are nothing more than a randomly scattered initial population where

each individual finds its own local optimum and the best of these local optima is kept as the algorithm's solution. This is not the case, however, due to the important effects of recombination. Indeed, recombination allows for "the transfer of information between successful candidates" [2]. Some information can be inferred about the as yet unexplored solution space and a global optimum can be honed in upon quicker. So were it not for the evolutionary process of recombination, genetic algorithms would be no better than pure random guesses. As it stands, though, a genetic algorithm uses stochastic processes and begins with a randomly generated population, but its result is "distinctly non-random (better than random)" [5]. In fact, by creating an initial population and running no iterations, a genetic algorithm is simply a Monte Carlo simulation.

Perhaps the greatest feature of a genetic algorithm, however, is that it does not need to know anything about the problem at hand. Further, it does not need to know how to even solve the problem, simply to recognize a good solution when it finds one [1]. In this way, genetic algorithms are much more "open-minded" than humans or many other traditional optimization methods. They rarely are reliant upon their initial conditions as discussed above, and do not necessarily make adjustments with the idea of constant improvement. Genetic algorithms possess a unique ability to make poor short-term decisions that result in better long-term gains. For example, a random recombination or mutation may not create offspring that aid the fitness of the current population but some part of that offspring may survive to later generations and prove to be beneficial.

In essence, genetic algorithms rely only upon their fitness function to determine if the changes that are made within a given iteration produced improvement or not.

They are “blind” to all the other influences affecting the system [2]. Because genetic algorithms can consider almost all possible pathways to an optimal solution, they often arrive at solutions that work better than those designed by humans or other systems but whose internal operation cannot be explained. Some select examples of such counterintuitive solutions discovered by genetic algorithms are presented later.

8 Risks of Genetic Algorithms

The benefits of genetic algorithms do not come without some inherent risks and limitations. The most evident risk seen in genetic algorithms as well as in nature is the risk of extinction. If the number of individuals in a population drops too low, the mutation rate is too high, or the crossover rate is too low, a population will most certainly be in danger of going extinct. For this reason, great care must be taken when determining mutation and crossover rates at the onset of the algorithm. Of course, the algorithm developer may desire to have the possibility of extinction in their simulation for the purpose of identifying if no feasible solution can be found.

Another risk when running a genetic algorithm is that an individual with a significantly higher fitness score than all other individuals, a “super-subject” [3], will appear in early generations and reproduce far too rapidly, driving down the diversity of the population too quickly. This well-documented problem is known as *premature convergence* [2]. A problem also exists that has the opposite effect of premature convergence. If too many individuals have similar fitness scores, the algorithm will not be able to determine which individuals to discard and which to keep and may stop progressing altogether. This problem is essentially caused by the “watering-down” of the population. However, because these problems are so

well known, several methods to avoid their occurrence have been developed over the past few decades. As discussed earlier, rank-based selection is one of these methods, used to reduce the strength of the stronger individuals. Other methods are now presented.

Windowing reduces the fitness of each individual by the fitness of the worst individual [3]. This strengthens the stronger members of the population, creating more disparity.

Exponential transformation takes the square root of each individual’s fitness plus one [3]. The influence of the stronger individuals is reduced, allowing for more diversity.

Linear transformation also reduces the influence of the stronger individuals, but instead transforms each individual’s fitness linearly ($y = mx + b$) [3].

Finally, while genetic algorithms possess the ability to find several potential solutions at one time, they are typically only able to find estimates of these optimal solutions, not the exact optima themselves [1]. But if an exact optimum is truly desired, one can typically use the solutions provided by the genetic algorithm as initial conditions to a more traditional, gradient-based optimization algorithm. In such a way, genetic algorithms still prove to be highly useful and efficient. Of course, it should also be noted that in the case of analytically solvable problems, traditional methods are advised for use over genetic algorithms because their more direct approach can result in both higher efficiency and higher accuracy. Still, for problems or solution spaces that are complex and cannot be solved by analytical means, genetic algorithms will almost always find at least one very good solution to the design problem at hand.

9 Recent Applications

Problems involving timetabling, scheduling, engineering, and optimization appear to be particularly appropriate for the use of genetic algorithms [7]. Several recent applications of genetic algorithms have developed solutions that have proven to be very effective and yet are most unusual in nature. The research and work behind a few of these more interesting applications is summarized below.

9.1 Geometry of Wire Antenna

Traditionally, the design of a wire antenna's shape is a random "guess and test" process based on pre-specified antenna properties and previous experience. For obvious reasons, this design process was a good candidate for improvement and optimization. Edward Altshuler and Derek Linden designed a genetic algorithm that determined an optimal wire antenna shape after evolving potential solutions through a search space [2].

These two men specifically used their algorithm to design a circularly polarized seven-segment antenna with hemispherical coverage. Each potential solution in the genetic algorithm kept track of the three-dimensional coordinates of each end of the seven antenna segments, the algorithm iterating and evolving them until it arrived at the shape shown in Figure 5. In spite of its atypical geometry, the antenna has shown in both other external simulations and in experimentation to exhibit a "nearly uniform radiation pattern with high bandwidth" [2].

9.2 Military Battle Plans

Tactical military battle planning can be very complex even for very simple mission plans. There are often many trade studies that must either be run or accounted for before engaging in a conflict or operation, and human officers often do not have the time

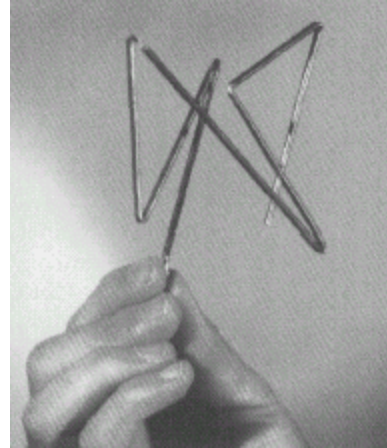


Figure 3: A crooked-wire genetic antenna [8].

and resources available to them to make qualified decisions all the time.

So for both training and analysis purposes, Robert Kewley and Mark Embrechts developed a genetic algorithm that automated the process of military battle planning. This simulation accounted for such factors as the landscape, troop movement speed, and firing accuracy while evolving an optimal battle plan based off a commander's desired outcome. To add another layer of reality, the developers allowed for the enemy's battle plans to evolve genetically at the same rate as the "friendly" system's plans. In this way, the evolving battle plan was forced to correct any inherent weaknesses before the enemy could find and exploit them [2].

When the results of this genetically developed plan were compared with the plans drawn up by trained military experts, the solution designed by the genetic algorithm showed "significantly higher mean performance than those generated by the experienced military experts" [9]. According to the researchers, the algorithm's plan made "good tactical sense" [2] while also providing insights into specific areas of mission planning that were overlooked at the onset of the task. The work conducted by these two men once

again demonstrates that genetic algorithms can indeed provide *useful* and *feasible* solutions to real-world problems.

9.3 University Exam Scheduling

Scheduling and timetable problems are generally known to be NP-complete, which in mathematical complexity theory means that they are the most difficult to solve in NP (“Non-deterministic Polynomial time”) [10]. In other words, no guaranteed-optimal solution can be found with the methods known to and proposed by humans today. E.K. Burke and J.P. Newall decided to push as hard as possible against this inevitable downfall and developed a genetic algorithm for university exam scheduling in hopes of besting the software systems currently in place at some universities.

The developer’s algorithm used rank-based selection while working against both hard constraints (two exams cannot physically be in the same location at the same time) and soft constraints (students should not be assigned consecutive exams) [2]. They also chose to incorporate a gradient-based algorithm for use in further optimizing the genetic algorithm’s solutions once it had completed its simulation run. In the end, the genetic algorithm produced exam schedules with 40% reduction in conflicts over the well-established algorithm previously used at some universities [2].

9.4 Software Tools

Over the last decade, the rise of computer power in the technology field has allowed for the emergence of several genetic optimization software packages, two of which are discussed below. These plug-ins and toolboxes can be very useful to a systems designer or engineer properly trained in their functionality.

DARWIN, an advanced optimization code written by Advanced Design and Optimization Technologies (ADOPTTECH)

utilizes a genetic algorithm that is particularly beneficial when optimizing large system objective functions involving many discrete design variables. The code is also capable of handling continuous design variables simultaneously, providing the end user with the ability to perform optimizations on variable sets of mixed types [11]. DARWIN is currently available as a plug-in or add-on with several modeling and simulation software packages.

For MATLAB, one of the most widely used engineering programs today, there exists both an optionally built-in genetic and evolutionary algorithm toolbox and several similar toolboxes available via third-party developers. The reasoning behind the relatively widespread use of MATLAB for genetic and evolutionary algorithm applications is that “MATLAB’s integrated graphics and matrix-based processing capabilities make it ideal for GA [genetic algorithm] work” [1]. Indeed, genetic algorithm code written in MATLAB tends to be relatively concise and comprehensible.

10 Conclusions

Through these applications, it can be seen that genetic algorithms are opening up the realm of possibility for effective solutions in many areas. Solutions developed via genetic algorithms that at first seem to border on the bizarre are being found to be more effective than their more conservative predecessors. Still, genetic algorithms should only be viewed “as alternatives to more traditional methods, not as replacements” [1]. Traditional optimizers certainly still hold their previous place in solving relatively well-behaved objective functions, but need the assistance of genetic algorithms to better estimate optima for larger, noisier functions.

As mentioned earlier, one of the greatest strengths of a genetic algorithm is its inherent ability to “adapt” to the specific

task at hand without ever needing to know any background information about the objective function it is solving. Its operators of recombination and mutation allow it to determine its own path without regard to external influences, building upon its previous successes, yet still able to explore more new regions of the solution space [1]. By searching with many individuals in parallel and keeping intercommunication strong between these individuals via recombination (crossover), a genetic algorithm will almost always return at least one, if not several, very good solutions among those possible.

11 References

- [1] Baker, Richard (1998, July). "Genetic Algorithms in Search and Optimization." *Financial Engineering News*. Retrieved December 4, 2004 from the World Wide Web: <http://www.fenews.com/fen5/ga.html>
- [2] Marczyk, Adam (2004). "Genetic Algorithms and Evolutionary Computation." *The Talk.Origins Archive*. Retrieved December 4, 2004 from the World Wide Web: <http://www.talkorigins.org/faqs/genalg/genalg.html>
- [3] Rennard, Jean-Philippe (2000). "Introduction to Genetic Algorithms." *Rennard.org*. Retrieved December 4, 2004 from the World Wide Web: <http://www.rennard.org/alife/english/gavintrgb.html>
- [4] Pohlheim, Hartmut (2004). "Evolutionary Algorithms 2 Overview." *GEATbx: Genetic and Evolutionary Algorithm Toolbox for use with MATLAB Documentation*. Retrieved December 4, 2004 from the World Wide Web: <http://www.geatbx.com/docu/index.html>
- [5] Kantrowitz, Mark (1997). "What's a Genetic Algorithm (GA)?" *Carnegie Mellon University Artificial Intelligence Repository*. Retrieved from the World Wide Web: <http://www-2.cs.cmu.edu/Groups/AI/html/faqs/ai/genetic/part2/faq-doc-2.html>
- [6] Buckland, Mat (2002). "Genetic Algorithms in Plain English." *AI-Junkie.com*. Retrieved December 4, 2004 from the World Wide Web: <http://www.ai-junkie.com/ga/intro/gat1.html>
- [7] *Wikipedia: the free encyclopedia* (2004). "Genetic Algorithm." Retrieved November 8, 2004 from the World Wide Web: http://en.wikipedia.org/wiki/Genetic_algorithm
- [8] Altshuler, Edward & Linden, Derek (1997, July). "Design of a wire antenna using a genetic algorithm." *Journal of Electronic Defense*, 20(7), 50-52.
- [9] Kewley, Robert & Embrechts, Mark (2002, May). "Computational military tactical planning system." *IEEE Transactions on Systems, Man and Cybernetics, Part C - Applications and Reviews*, 32(2), 161-171.
- [10] *Wikipedia: the free encyclopedia* (2004). "NP-complete" Retrieved November 8, 2004 from the World Wide Web: <http://en.wikipedia.org/wiki/NP-complete>
- [11] *Advanced Design and Optimization Technologies* (2002). "Darwin: a general purpose genetic algorithm." Retrieved November 8, 2004 from the World Wide Web: <http://www.adoptech.com/software/Darwin.htm>