Teaching Genetic Algorithms with a Graphical User Interface

Gregory J. Toussaint, Daniel J. Pack, and Randy L. Haupt
United States Air Force Academy

Abstract - *Over the past several years, genetic algorithms have emerged as a powerful tool for solving optimization problems in engineering. Genetic algorithms model biological evolution on the computer using the principles of natural selection, mating and mutation. Although the subject has been predominantly studied at the graduate level, undergraduate students can easily master the concepts. We have developed MATLAB-based software with a graphical user interface to teach the fundamentals of genetic algorithms. The program allows a user to adjust several different parameter values associated with genetic algorithms including the optimization function, the population size, the crossover rate, and the mutation rate. A user can graphically monitor how these parameters affect the evolutionary path the algorithm takes to find an optimal solution. This approach of teaching students to experiment with genetic algorithms increases their level of understanding and allows them to quickly grasp the essential properties of the method. We will demonstrate how users can execute the genetic algorithm software to solve some sample engineering optimization problems.*

1 Introduction

Recently, many researchers around the globe have given a considerable amount of attention to a number of methods that use non-traditional system models to solve complex problems. Among these methods, neural networks, fuzzy logic, and genetic algorithms are most prominent. All three methods simulate the dynamics of a system without relying on rigorous mathematical models. To those who are not familiar with the new methods, such approaches may seem inadequate without mathematical rigor, but the techniques are superior to conventional methods in a variety of advanced problems: inverted pendulum control problem using neural networks[1], Sendai train control using fuzzy logic control[2], and antenna array pattern optimization problem using genetic algorithms[3]. In most cases, however, these new methods have been taught and studied in the graduate schools and have not been easily accessible to wider audiences.

To help undergraduate students learn one of these new-proven techniques early in their academic careers, we developed simple and effective graphical user interface software for genetic algorithms. We started this work because we could not easily find a good educational tool to present genetic algorithms to undergraduate students. Our MATLAB-based software allows students to easily explore how genetic algorithms use the principles of natural selection to solve optimization problems. The software allows students to control most of the algorithm's parameters and observe how they influence the problem solving process.

The paper is organized in the following manner. First we start off with a brief description of genetic algorithms. Section three presents the graphical user interface software followed by two sample problem solving sessions in section four. These sessions demonstrate how the software can be used by an undergraduate student to explore the subject. The last section concludes with discussion on future modifications to the software.

2 Genetic Algorithms

Biologists have observed genetic algorithms in their original forms in nature for over a century. The basic principle is that only those who can best adapt to the current environment survive, reproduce, and pass on their traits to the next generation. The less fit individuals cannot successfully compete with the more fit individuals and die. Over the past two decades this principle has caught the attention of researchers in the engineering community who have subsequently developed computer models of the natural selection process. They have applied these models to a variety of applications. Holland initiated the significant work in this field and introduced genetic algorithms to the engineering community[5]. Goldberg popularized the method[6], and DeJong[7] applied the method to several applications. In the remainder of this section, we describe the ins and outs of the genetic algorithms.

A genetic algorithm "population" is a set of binary strings. Each string is a binary encoding of the optimization variables. The bits in each string represent an individual's traits. Fitness, in this case, is the function value corresponding to a set of variables. The genetic algorithm selects individuals with high fitness values for mating, reproduction, and mutation. The overall goal is to create an individual string which represents the optimum solution to a particular search problem.

Genetic algorithms use the population of random binary strings to start search problems from multiple initial locations. Starting from many different locations allows the algorithms to explore a larger portion of the search space than traditional search techniques, such as gradient-based methods, that start from a single point. In addition, as the set of solutions moves toward an optimum, the number of possible solutions stays the same. In this respect, genetic algorithms resemble a parallel process while classical techniques can be viewed as a serial approach.

For genetic algorithms, mating and reproduction involve selecting individuals based on their fitness and then swapping genetic material (bits) to produce new individuals. The rational for this procedure is that if individuals with better fitness are favored during reproduction, the average fitness for the entire population in the next generation will tend to improve. The offspring represent new initial solutions to be used in the next iteration. The process continues until the population converges to an optimum solution. The number of iterations required to find an optimal solution depends on the complexity of the problem and the parameter settings for the genetic algorithm. From our experience, the convergence occurred within a few iterations for a simple optimization problem and approximately 30 to 50 iterations for modest size optimization problems.

After a new population is generated, random mutations may occur. These random mutations result in changing the value of a single bit from one to zero or vice versa. Mutations occur with a small probability, typically in much less than one percent of the total genetic material. The random changes allow the algorithm to continue exploring the search space after the solutions converge near the optimal values. Without mutation, the algorithm could prematurely converge on a local optimum and have no way to move to the global solution.

In addition to searching from multiple locations, genetic algorithms offer other advantages over classical search techniques. The genetic algorithm must determine a fitness for each solution, but does not require any auxiliary information about the evaluation function. Classical search techniques often rely on information about the derivative of a function to determine the next step. The derivative of a function increases the computational complexity and may not always be available. Another advantage for genetic algorithms is that they have the ability to identify more than one optimal solution. If there are two equally good solutions in the search space, a genetic algorithm should have portions of the population converge on both points. A traditional search technique will ignore one of the potential solutions.

We have briefly outlined the fundamental aspects of genetic algorithms and have identified some of their advantages over traditional search techniques. In the next section, we describe our graphical user interface and some of the unique features of the underlying genetic algorithm.

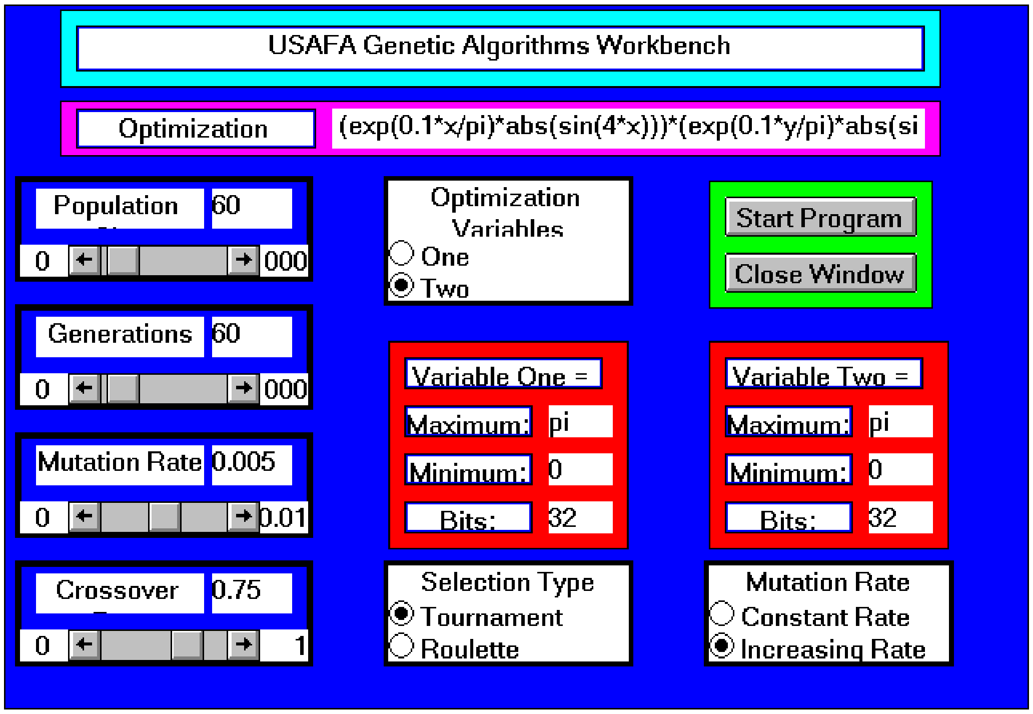
3 Graphical User Interface for Genetic Algorithms

We designed our graphical user interface to help students learn genetic algorithms using the MATLAB numeric computation software. MATLAB is well know for its numerical processing capabilities, but it also has built-in functions to create standard graphical tools such as windows, sliders, and push buttons. We used these utilities to allow a user to efficiently execute the genetic algorithm program. For each program execution, the user can easily vary the relevant parameter values without having to write or modify code. In addition to the easy access to the parameter values, the software graphically displays the solutions which allows users to observe the evolutionary process. We believe a simple graphical user interface that varies the algorithm's parameter values appeals to individuals more than modifying computer code.

We developed the software on an IBM compatible (486 / 33 MHz) personal computer using the MS-Windows Student Edition of MATLAB (Version 4). The program runs on any platform that supports MATLAB, but the graphical displays differ slightly among MS-Windows, X Windows, and Macintosh operating systems. We successfully executed the code on a Sun Sparc station 2, but did not verify its operation on a Macintosh. Any changes between the platforms are cosmetic.

To initiate the graphical user interface, the user first invokes the MATLAB software then executes the file containing the interface program. The program displays a new window with all of the required parameter values to execute the genetic algorithm routine, as shown in Fig. 1. The individual types in the function to be optimized using standard MATLAB expressions and notation. There are a few restrictions on the optimization function. Currently the program only allows for one or two variables in the expression to be optimized. If there is one variable, it must be "x" and if there are two variables they must be "x" and "y". For this program, optimum means maximum. To minimize a function, the entire expression just needs to be negated.

The user moves sliders to adjust the population size (number of individuals), the number of generations, the mutation rate and the crossover rate for the algorithm. These four parameters are also changed by typing their values directly into a text field. Next, the user clicks on a radio button to choose either tournament selection or roulette selection for the mating procedures. The radio button allows only one of two choices to be selected at a time. Although there are a variety of selection strategies, we limited the first version of our software to these two.


USAFA Genetic Algorithms Workbench

Optimization (exp(0.1*x/pi)*abs(sin(4*x)))*(exp(0.1*y/pi)*abs(si

Population Size 60 (0 – 000)

Generations 60 (0 – 000)

Mutation Rate 0.005 (0 – 0.01)

Crossover Rate 0.75 (0 – 1)

Optimization Variables: One [✓ Two]

Variable One = Maximum: pi Minimum: 0 Bits: 32

Variable Two = Maximum: pi Minimum: 0 Bits: 32

Selection Type: [✓ Tournament] Roulette

Mutation Rate: Constant Rate [✓ Increasing Rate]

Figure 1: *MATLAB graphical user interface used to set parameter values and initiate the genetic algorithm.*

The tournament selection scheme randomly chooses two individuals from the entire population. The program then compares the fitness of the individuals and 75 percent of the time the more fit individual is selected to be included in the mating pool. One quarter of the time, the individual with the lower fitness proceeds to the mating pool. This method is easy to implement and produces reasonable results. In the roulette selection approach the program first determines the total fitness of the population. Then the software computes each individual's percentage of the total fitness and uses that value to create a cumulative probability distribution. The program generates a random number between zero and one and searches the population to find the individual whose cumulative probability is greater than the random number. For more details, the reader is referred to Goldberg[6]. Roulette table selection requires more computation than tournament selection and will increase the execution time of the genetic algorithm.

The user clicks another radio button to select the number of variables in the optimization function. The number of variables selected must match the number used in the optimization function for the algorithm to execute successfully. For each variable, the user enters the maximum and minimum values it can have during the search and the number of bits used to represent the solution. The number of bits determine the binary string length of each individual as well as the resolution of the search. Increasing the number of bits improves the resolution of the search, but also increases the computational complexity of the algorithm.

Finally, the user clicks another radio button to determine if the mutation rate will remain constant throughout the search or if it will increase based on the number of generations completed. Increasing the mutation rate as a function of the generation allows the algorithm to gradually expand its search at a time when the population would normally be converging toward the optimal solution. To illustrate the value of this technique, consider the case where the mutation rate is constant. The number of mutations would be approximately the same for each generation. As the program executes, the most fit individual will begin to dominate the population and will be favored during the mating process. With its mating prowess, the optimum solution will take over a higher percentage of the population and the algorithm will no longer be searching for new solutions effectively. When the mutation rate is zero, the entire population equals the optimum solution. Increasing the number of mutations as a function of the generation prevents the algorithm from stagnating as it progresses. In the later generations, an aggressive mutation rate helps explore a larger percentage of the search space and possibly escape from a local minimum.

As the genetic algorithm executes, the program displays a small window with a summary of the solution process. The window contains numerical information about the generation completed, the best fitness found so far, the average fitness for that generation, and the standard deviation of the fitness for the population. The summary data allows the user to track the algorithm's progress toward a solution.

When the program starts, the software generates a random population and calculates the fitness of each individual. It then sorts the population from lowest to highest fitness. The best solution found up to the current iteration always survives to the next generation. This technique is well known as the elitist approach, and it ensures we do not lose any valuable information found during the search. The program then randomly chooses individuals for mating using one of the selection approaches discussed above. Even though the best solution is protected, it is still included as a member of the mating pool, so it can pass its beneficial traits on to future generations. After selecting and mating individuals, we perform random mutations on the entire population. Since we are using the elitist approach and maintaining the best individual, mutation never destroys our optimum solution.

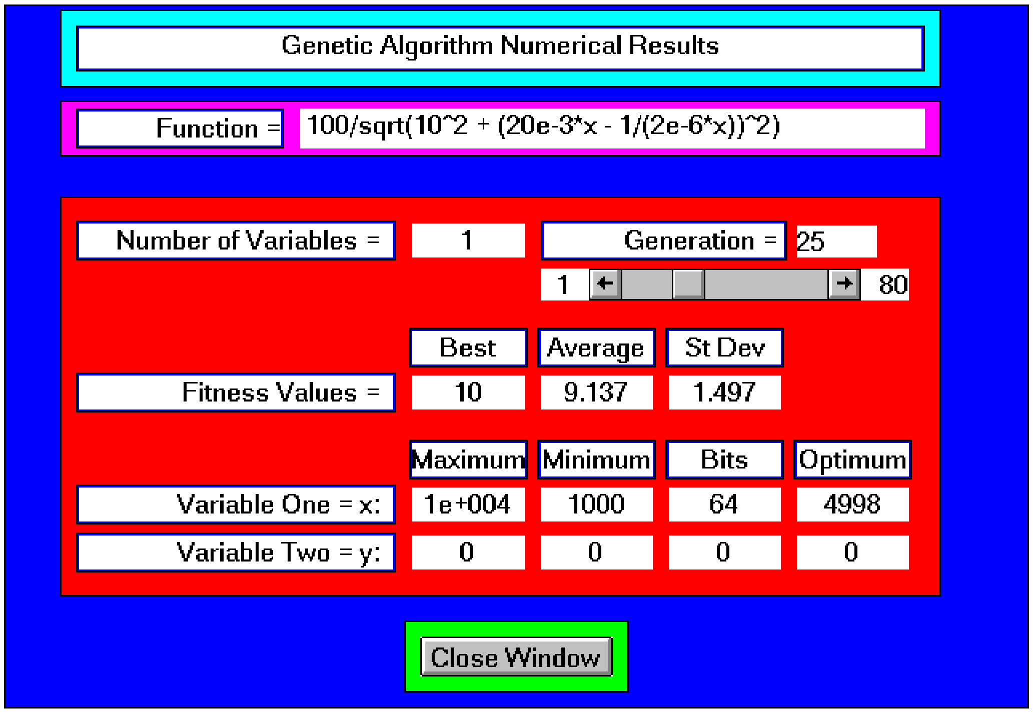
As the program executes, the fitness statistics are recorded, and we maintain a list of the best solutions for each iteration to display at the end of the program. Maintaining this list allows the user to visualize how the solution set improves after each iteration. Once the algorithm completes the requested number of generations, the software can present the results in numerical or graphical form. The numerical summary contains some of the initial parameter values and fitness data for each generation, as shown in Fig. 2. The user controls the generation number displayed with a slider bar or can directly edit the generation number in the text field. The statistics include the best fitness, average fitness, and standard deviation of the fitness for the designated generation. In addition, the optimum parameter values for the variables are displayed for each generation. The numerical results are instructive, but the graphical results provide a more meaningful representation of how the algorithm performed and converged on a solution.

After the completion of the program, the user also selects from three graphical display options. The first option plots the variables and fitness for each generation. Again, the user controls the generation number of the information displayed. For one variable problems, we use a two dimensional plot of the variable versus fitness. For two variable problems, we plot one variable on each axis and encode the fitness information with color. MATLAB has several colormaps for graphical representation and we chose the "Hot" colormap which uses lighter colors for larger values and darker colors for smaller values.

In addition to the two dimensional representations, the user can look at how the solutions progress with a three dimensional graph. The three dimensional graph plots the solutions for each generation against the generation number and their fitness values. The three dimensional perspective vividly shows how the population moves toward the optimum solution and how an increased mutation rate influences the search in the later generations if the user selects that option.

The final graphical representation summarizes the information in the numerical results on a single two dimensional plot. The program displays best fitness, average fitness, and fitness standard deviation against the generation number. The graph shows the effect of the elitist strategy, since the best solution never decreases.

It also shows how the solutions converge toward an optimum as the average fitness increases and the fitness standard deviation decreases. If the mutation rate is set to increase as a function of the generation, the average fitness will increase but will not approach the best fitness as it does when the mutation rate is constant. Similarly, the fitness standard deviation does not converge if the mutation rate increases.


Genetic Algorithm Numerical Results

Function = 100/sqrt(10^2 + (20e-3*x - 1/(2e-6*x))^2)

Number of Variables = 1

Generation = 25

	Best	Average	St Dev
Fitness Values =	10	9.137	1.497

	Maximum	Minimum	Bits	Optimum
Variable One = x:	1e+004	1000	64	4998
Variable Two = y:	0	0	0	0

Figure 2: *Numerical results displayed after the completion of the genetic algorithm.*

After reviewing all of the numerical and graphical results, the user can return to the algorithm interface window to adjust parameter values and execute the program again. The interface easily allows the user to make small changes to the algorithm and view the results in a variety of formats. In the following section, we present two sample sessions which illustrate how students could apply the software to solve engineering problems.

4 Experimental Results

To illustrate the effectiveness of the software, we will present an electrical engineering problem that has a known solution. Suppose a student wants to determine the resonant frequency of a series RLC circuit. The circuit contains an AC voltage source with a magnitude of 100 V, a resistor with R = 10 Ω, an inductor with L = 20 mH, and a capacitor with C = 2 µF. Resonance occurs at a frequency when the circuit appears completely resistive to the source and the current in the circuit is a maximum. Using analytical techniques, we can determine the circuit's resonant frequency, $\omega_o = 1/\sqrt{LC}$. The student can use the genetic algorithm to find this solution by maximizing the current in the circuit. The expression for the current in terms of the component values and the radian frequency is shown in Eq. 1.

$$I = \frac{|V|}{\sqrt{R^2 + \left(\omega L - \frac{1}{\omega C}\right)^2}} \tag{1}$$

In terms of the given parameter values, the expression for current becomes:

$$I = \frac{100}{\sqrt{10^2 + \left(0.02\omega - \frac{1}{2 \times 10^6 \omega}\right)^2}} \qquad (2)$$

The student enters the current expression and sets the parameter values for the genetic algorithm shown in Table 1.

Parameter	Value
Population Size	80
Generations	80
Mutation Rate	0.005
Crossover Rate	0.75
Selection Type	Roulette
Mutation Type	Constant
Number of Variables	1

Variable	Max	Min	Bits
x	10000	1000	64

Table 1: *Parameter values for the resonant circuit example.*

The student can now execute the genetic algorithm and observe the fitness statistic, which represents the magnitude of the current, as the algorithm gradually converges on the solution: $x = \omega_o = 5000$ rad/s and $I = 10$ A. Table 2 shows the actual results from the algorithm, and Fig. 3 shows a plot of the variable values versus fitness for the population of solutions at the first generation. Fig. 4 is the same plot for a later generation and clearly demonstrates how the solutions converge on the optimum value.

Solution	
Best Fitness =	10.0000
Variable x =	5000.1221

Table 2: *The genetic algorithm's solution for the resonant circuit example.*

In addition to the two dimensional graphs, we can view the results in three dimensions. The three dimensional graph in Fig. 5 plots individual solutions versus the generation number versus the fitness. After the initial search stages, the graph clearly levels off showing how the population has converged on the optimum solution.

For a two variable example, we created an artificial problem that would challenge most of the traditional gradient search techniques. The function we want to optimize is a product of two exponentially increasing sinusoids, shown in Eq. 3.

$$\left[e^{\frac{0.1x}{\pi}} \times \left|\sin(4x)\right|\right] \times \left[e^{\frac{0.1y}{\pi}} \times \left|\sin(3y)\right|\right] \qquad (3)$$

MATLAB separately generated the three dimensional plot of the fitness function shown in Fig. 6. Obviously, if we can plot the function in three dimensions, there is no need to search for the optimum solution, but we did this to illustrate the nature of the search space the algorithm will have to deal with. The user can initiate the genetic algorithm with parameter values such as the ones shown in Table 3 and find the solution described in Table 4.

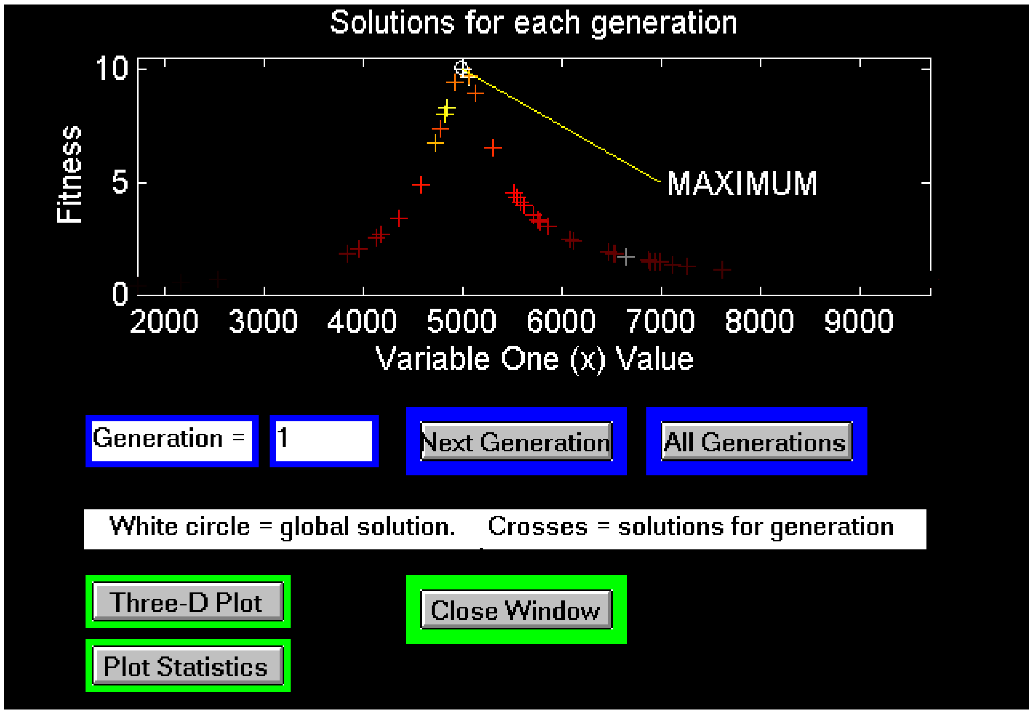


Figure 3: *Plot of the optimization variable versus fitness at the first generation. The global optimum is the white circle on the graph.*

Parameter	Value
Population Size	60
Generations	60
Mutation Rate	0.005
Crossover Rate	0.75
Selection Type	Tournament
Mutation Type	Increasing
Number of Variables	2

Variable	Max	Min	Bits
x	π	0	64
y	π	0	32

Table 3: *Parameter values for the two variable example.*

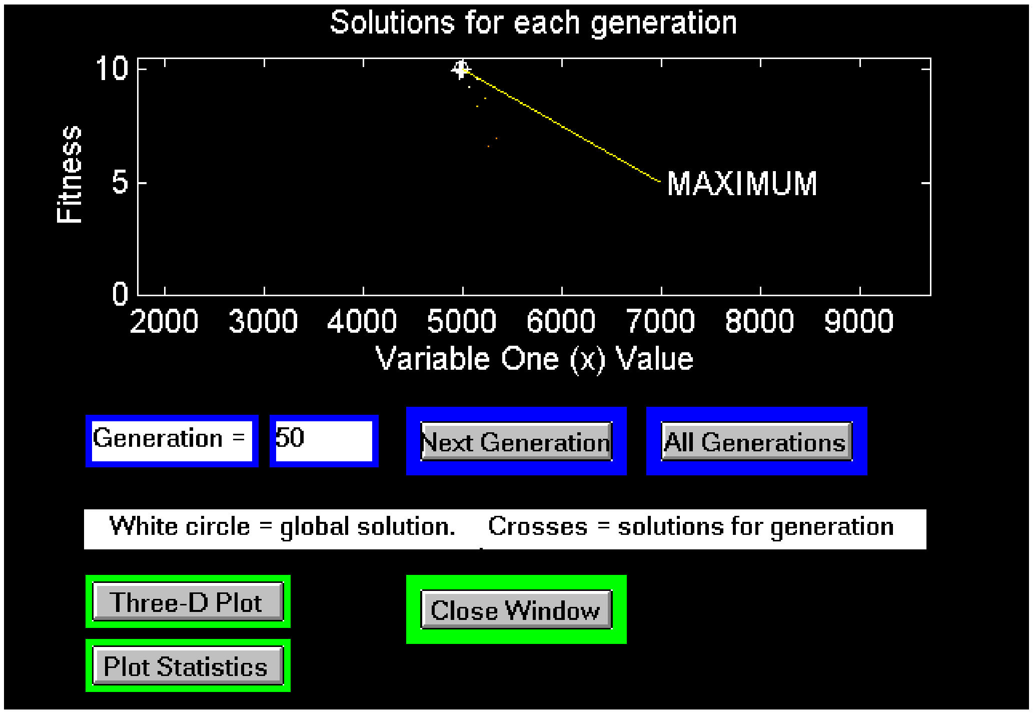


Figure 4: *Plot of the optimization variable versus fitness after the individual solutions have converged to the optimum value.*

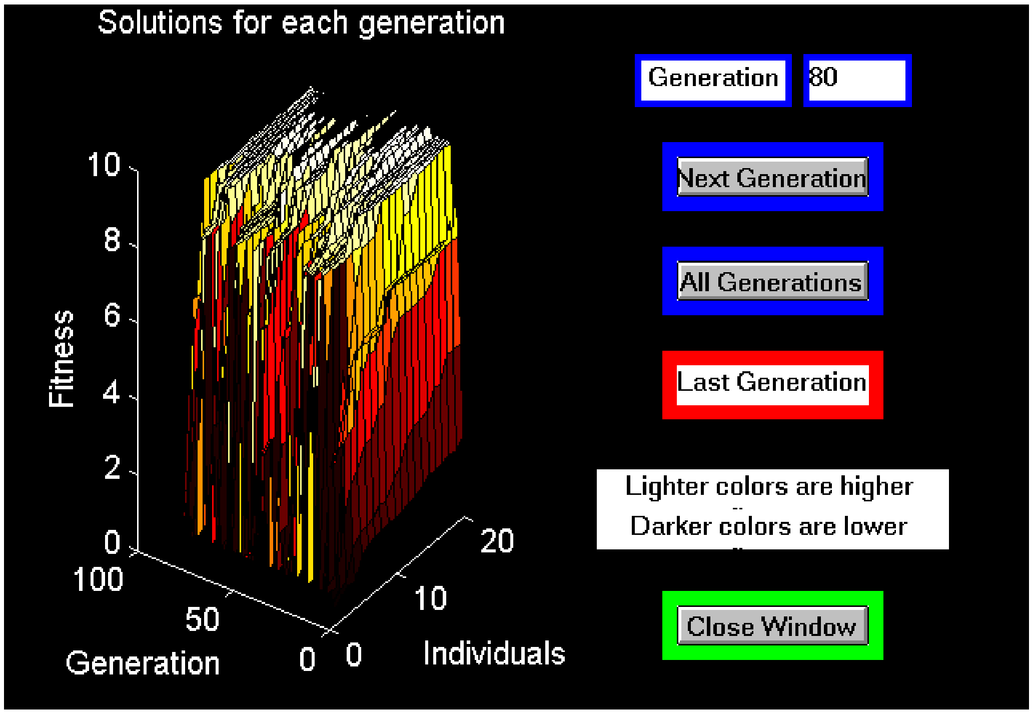


Figure 5: *Three dimensional plot of the genetic algorithm's solution to the resonant circuit problem. This graph displays the fitness of 20 solutions for each generation.*

Solution	
Best Fitness =	1.1856
Variable x =	2.7467
Variable y =	2.6292

Table 4: *The genetic algorithm's solution for the two variable example.*

The numerical results from running the program have the same appearance as the one variable case, but one of the graphs is significantly different and worth examining. Fig. 7 shows a plot of the two function variables versus fitness. The fitness information is represented in color and does not reproduce adequately in black and white. The results in Fig. 7 closely correspond to a top view of the three dimensional graph of the search space shown in Fig. 6.

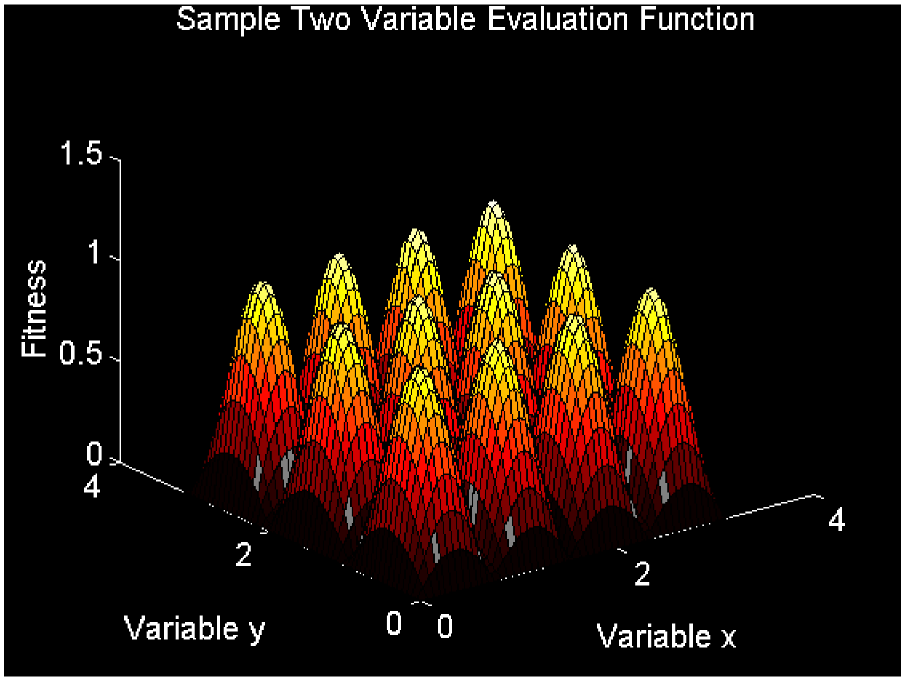


Figure 6: E*valuation function for the two variable example.*

The only other graph available to the user is a plot of the fitness statistics versus generation, as shown in Fig. 8. This graph shows the best fitness, average fitness, and fitness standard deviation versus generation and provides some insight into how the algorithm converged. In this case, program increased the mutation rate as a function of the generation which prevented the average fitness from approaching the best fitness as discuss in Section 3.

These simple examples illustrate the basic operation of the graphical user interface and the genetic algorithm it controls. The software is very flexible in the types of problems it can handle and is only limited by MATLAB's extensive range of commands and the user's imagination.

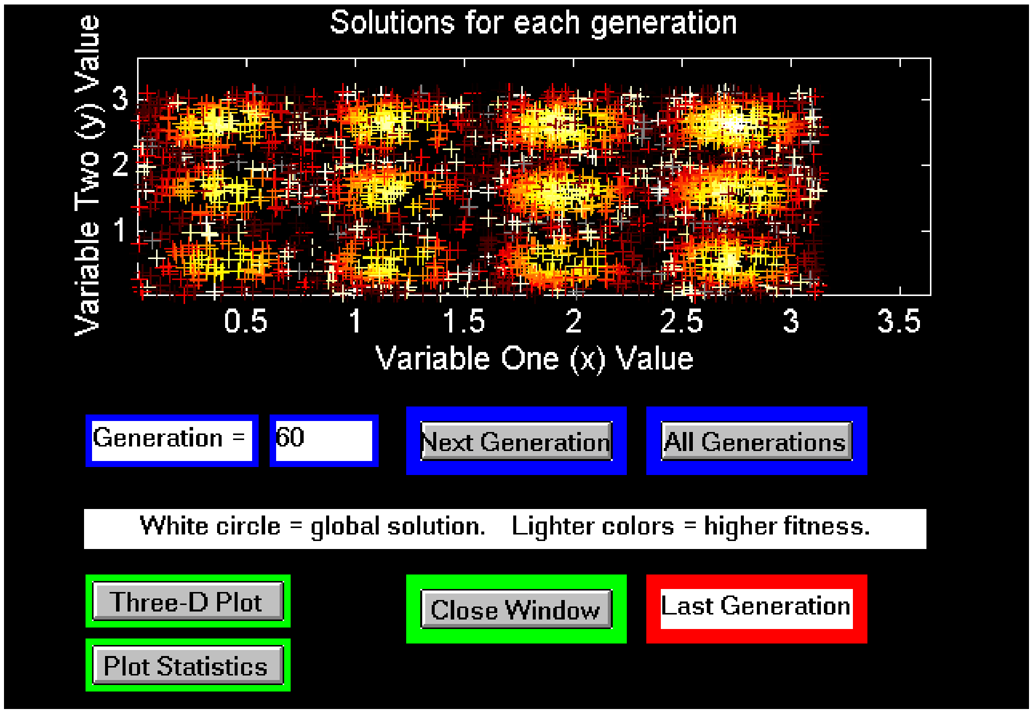


Figure 7: *Plot of two variable function versus fitness. Fitness is encoded in color with lighter colors representing higher fitness.*

We plan to improve the graphical user interface and the underlying genetic algorithm program to enhance its value as a teaching tool and to improve its effectiveness for research. One of the first improvements will be to optimize the MATLAB code for efficiency and speed. We can always move to a more powerful computational platform, but we would like the program to be as efficient and elegant as possible.

A simple improvement would be to add a significant amount of error checking to the basic program. Currently the program requires a knowledgeable user to make it operate successfully. A variety of error checking for the optimization function and other values the user is allowed to enter will prevent execution errors that cause the program to halt.

Another additional feature would be to archive the data generated from a set of simulations into a single database for further analysis. For example, instead of optimizing a function with just one set of initial conditions, we would allow one or more parameters to vary over a range of values and collect data on how the variations influence convergence. These types of changes would require extensive changes to the graphical user interface and how the results are displayed, but the code for the algorithm would need only minor adjustments.

We can also improve the program by increasing the flexibility of the optimization function. We could allow more than two variables in the function, allow the user to specify the variable names to naturally match a particular application, or include external constraints on the optimization function. These enhancements would require significant changes to both the interface and the genetic algorithm.

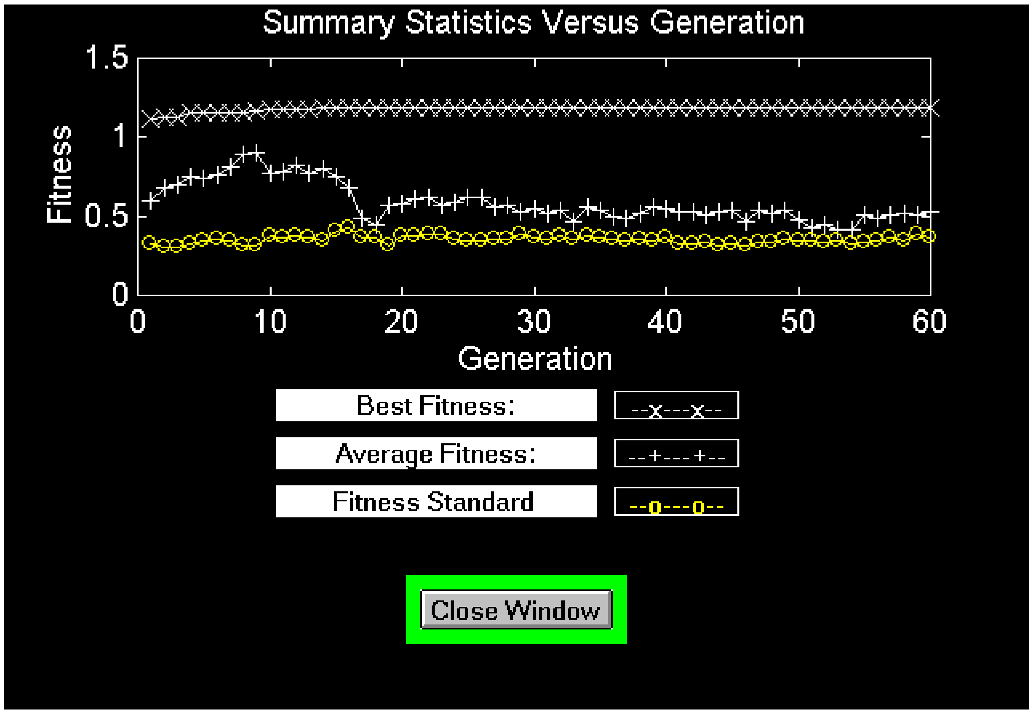


Figure 8: *Summary statistics for the two variable example. Starting from the top of the screen, we show best fitness, average fitness, and fitness standard deviation plotted against the generation number.*

5 Conclusion

Genetic algorithms are becoming an accepted and valuable optimization technique in the engineering community because they can elegantly find solutions to complex problems. To help proliferate this tool, we developed graphical user interface software to allow novice users to investigate the world of genetic algorithms. By making the interface and the underlying algorithm simple, users can easily set up, execute, and observe the results of the program as it solves meaningful problems. In addition, users are free to modify any part of the code if they want to explore the program in more detail. We developed the software using the widely available MATLAB program and plan to introduce it as an optional part of our electrical engineering curriculum. We will continue to refine the program as we experiment with it and receive constructive criticism from various users.

Bibliography

[1] Daniel J. Pack, Min Meng, and A. C. Kak, “Comparative Study of Motion Control Methods for a Nonlinear System,” *Proceedings for 19th International Conference of the IEEE Industrial Electronics Society*, pp. 1413-1418, Maui Hawaii, November 1993.

[2] Daniel Schwartz, “Fuzzy Logic flowers in Japan,” *IEEE Spectrum*, July 1992.

[3] Randy Haupt, “An Introduction to Genetic Algorithms for Electromagnetics,” *IEEE Antennas and Propagation Magazine*, vol 37, no 2, April 1995.

[4] Rick L. Riolo, “Survival of the Fittest Bits,” *Scientific American*, July 1992.

[5] John Holland, “Genetic Algorithms,” *Scientific American*, July 1992.

[6] David Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley Publishing Company, Inc, Reading Massachusetts, 1989.

[7] K.A. DeJong, “On Using Genetic Algorithms to Search Problem Spaces: Genetic Algorithms and Their Applications,” *Proceedings of the Second International Conference on Genetic Algorithms*, pp. 210-216.

Biographies

GREGORY J. TOUSSAINT is an Electrical Engineering instructor at the USAF Academy, CO. He received his M.S. in Systems Engineering from the Air Force Institute of Technology, Wright-Patterson AFB, OH in 1992 and the B.S. in Electrical Engineering from Cornell University, Ithaca, NY in 1989.

DANIEL J. PACK is an Assistant Professor of Electrical Engineering at the USAF Academy, CO. He received the Bachelor of Science in Electrical Engineering in 1988, the Master of Science in Engineering Sciences in 1990, and the Ph. D in Electrical Engineering in 1995 from Arizona State University, Harvard University, and Purdue University, respectively.

RANDY L. HAUPT received his Ph.D. in Elect. Engr. from University of Michigan, Ann Arbor, MI in 1987, the M.S. in Elect. Engr. from Northeastern University, Boston, MA in 1983, the M.S. in Engr. Management from Western New England College, Springfield, MA in 1981, and the B.S. in Elect. Engr. from USAF Academy, CO in 1978. He is currently a Lt. Col. and a Professor of Elect. Engr. at the USAF Academy, CO.