A Genetic Algorithm for Function Optimization: A Matlab Implementation

Christopher R. Houck
North Carolina State University
and
Jeffery A. Joines
North Carolina State University
and
Michael G. Kay
North Carolina State University

A genetic algorithm implemented in Matlab is presented. Matlab is used for the following reasons: it provides many built in auxiliary functions useful for function optimization; it is completely portable; and it is efficient for numerical computations. The genetic algorithm toolbox developed is tested on a series of non-linear, multi-modal, non-convex test problems and compared with results using simulated annealing. The genetic algorithm using a float representation is found to be superior to both a binary genetic algorithm and simulated annealing in terms of efficiency and quality of solution. The use of genetic algorithm toolbox as well as the code is introduced in the paper.

Categories and Subject Descriptors: G.1 [**Numerical Analysis**]: Optimization— *Unconstrained Optimization, nonlinear programming, gradient methods*

General Terms: Optimization, Algorithms

Additional Key Words and Phrases: genetic algorithms, multimodal nonconvex functions, Matlab

1. INTRODUCTION

Algorithms for function optimization are generally limited to convex regular functions. However, many functions are multi-modal, discontinuous, and nondifferen-

Name: Christopher R. Houck
Address: North Carolina State University, Box 7906, Raleigh, NC, 27695-7906,USA,(919) 515-5188,(919) 515-1543,chouck@eos.ncsu.edu
Affiliation: North Carolina State University
Name: Jeffery A. Joines
Address: North Carolina State University, Box 7906, Raleigh, NC, 27695-7906,USA,(919) 515-5188,(919) 515-1543,jjoine@eos.ncsu.edu
Affiliation: North Carolina State University
Name: Michael G. Kay
Address: North Carolina State University, Box 7906, Raleigh, NC, 27695-7906,USA,(919) 515-2008,(919) 515-1543,kay@eos.ncsu.edu
Affiliation: North Carolina State University
Sponsor: This research was funded in part by the National Science Foundation under grant number DMI-9322834.

tiable. Stochastic sampling methods have been used to optimize these functions. Whereas traditional search techniques use characteristics of the problem to determine the next sampling point (e.g., gradients, Hessians, linearity, and continuity), stochastic search techniques make no such assumptions. Instead, the next sampled point(s) is(are) determined based on stochastic sampling/decision rules rather than a set of deterministic decision rules.

Genetic algorithms have been used to solve difficult problems with objective functions that do not possess "nice" properties such as continuity, differentiability, satisfaction of the Lipschitz Condition, etc.[Davis 1991; Goldberg 1989; Holland 1975; Michalewicz 1994]. These algorithms maintain and manipulate a family, or population, of solutions and implement a "survival of the fittest" strategy in their search for better solutions. This provides an implicit as well as explicit parallelism that allows for the exploitation of several promising areas of the solution space at the same time. The implicit parallelism is due to the schema theory developed by Holland, while the explicit parallelism arises from the manipulation of a population of points—the evaluation of the fitness of these points is easy to accomplish in parallel.

Section 2 presents the basic genetic algorithm, and in Section 3 the GA is tested on several multi-modal functions and shown to be an efficient optimization tool. Finally, Section 4 briefly describes the code and presents the list of parameters of the Matlab implementation.

2. GENETIC ALGORITHMS

Genetic algorithms search the solution space of a function through the use of simulated evolution, i.e., the survival of the fittest strategy. In general, the fittest individuals of any population tend to reproduce and survive to the next generation, thus improving successive generations. However, inferior individuals can, by chance, survive and also reproduce. Genetic algorithms have been shown to solve linear and nonlinear problems by exploring all regions of the state space and exponentially exploiting promising areas through mutation, crossover, and selection operations applied to individuals in the population [Michalewicz 1994]. A more complete discussion of genetic algorithms, including extensions and related topics, can be found in the books by Davis [Davis 1991], Goldberg [Goldberg 1989], Holland[Holland 1975], and Michalewicz [Michalewicz 1994]. A genetic algorithm (GA) is summarized in Fig. 1, and each of the major components is discussed in detail below.

(1) Supply a population P_0 of N individuals and respective function values.
(2) $i \leftarrow 1$
(3) $P_i' \leftarrow selection\_function(P_i - 1)$
(4) $P_i \leftarrow reproduction\_function(P_i')$
(5) $evaluate(P_i)$
(6) $i \leftarrow i + 1$
(7) Repeat step 3 until termination
(8) Print out best solution found

Fig. 1. A Simple Genetic Algorithm

The use of a genetic algorithm requires the determination of six fundamental issues: chromosome representation, selection function, the genetic operators making up the reproduction function, the creation of the initial population, termination criteria, and the evaluation function. The rest of this section describes each of these issues.

2.1 Solution Representation

For any GA, a chromosome representation is needed to describe each individual in the population of interest. The representation scheme determines how the problem is structured in the GA and also determines the genetic operators that are used. Each individual or chromosome is made up of a sequence of genes from a certain alphabet. An alphabet could consist of binary digits (0 and 1), floating point numbers, integers, symbols (i.e., A, B, C, D), matrices, etc. In Holland's original design, the alphabet was limited to binary digits. Since then, problem representation has been the subject of much investigation. It has been shown that more natural representations are more efficient and produce better solutions[Michalewicz 1994]. One useful representation of an individual or chromosome for function optimization involves genes or variables from an alphabet of floating point numbers with values within the variables upper and lower bounds. Michalewicz[Michalewicz 1994] has done extensive experimentation comparing real-valued and binary GAs and shows that the real-valued GA is an order of magnitude more efficient in terms of CPU time. He also shows that a real-valued representation moves the problem closer to the problem representation which offers higher precision with more consistent results across replications. [Michalewicz 1994]

2.2 Selection Function

The selection of individuals to produce successive generations plays an extremely important role in a genetic algorithm. A probabilistic selection is performed based upon the individual's fitness such that the better individuals have an increased chance of being selected. An individual in the population can be selected more than once with all individuals in the population having a chance of being selected to reproduce into the next generation. There are several schemes for the selection process: roulette wheel selection and its extensions, scaling techniques, tournament, elitist models, and ranking methods [Goldberg 1989; Michalewicz 1994].

A common selection approach assigns a probability of selection, P_j, to each individual, j based on its fitness value. A series of N random numbers is generated and compared against the cumulative probability, $C_i = \sum_{j=1}^{i} P_j$, of the population. The appropriate individual, i, is selected and copied into the new population if $C_{i-1} < U(0,1) \leq C_i$. Various methods exist to assign probabilities to individuals: roulette wheel, linear ranking and geometric ranking.

Roulette wheel, developed by Holland [Holland 1975], was the first selection method. The probability, P_i, for each individual is defined by:

$$P[\text{ Individual } i \text{ is chosen }] = \frac{F_i}{\sum_{j=1}^{PopSize} F_j}, \tag{1}$$

where F_i equals the fitness of individual i. The use of roulette wheel selection limits

the genetic algorithm to maximization since the evaluation function must map the solutions to a fully ordered set of values on $\Re^+$. Extensions, such as windowing and scaling, have been proposed to allow for minimization and negativity.

Ranking methods only require the evaluation function to map the solutions to a partially ordered set, thus allowing for minimization and negativity. Ranking methods assign P_i based on the rank of solution i when all solutions are sorted. Normalized geometric ranking, [Joines and Houck 1994], defines P_i for each individual by:

$$P[\text{ Selecting the } i\text{th individual }] = q\prime(1-q)^{r-1}; \tag{2}$$

where:

q = the probability of selecting the best individual,
r = the rank of the individual, where 1 is the best.
P = the population size
$q\prime = \frac{q}{1-(1-q)^P}$

Tournament selection, like ranking methods, only requires the evaluation function to map solutions to a partially ordered set, however, it does not assign probabilities. Tournament selection works by selecting j individuals randomly, with replacement, from the population, and inserts the best of the j into the new population. This procedure is repeated until N individuals have been selected.

2.3 Genetic Operators

Genetic Operators provide the basic search mechanism of the GA. The operators are used to create new solutions based on existing solutions in the population. There are two basic types of operators: crossover and mutation. Crossover takes two individuals and produces two new individuals while mutation alters one individual to produce a single new solution. The application of these two basic types of operators and their derivatives depends on the chromosome representation used.

Let $\bar{X}$ and $\bar{Y}$ be two m-dimensional row vectors denoting individuals (parents) from the population. For $\bar{X}$ and $\bar{Y}$ binary, the following operators are defined: binary mutation and simple crossover.

Binary mutation flips each bit in every individual in the population with probability p_m according to equation 3.

$$x_i' = \begin{cases} 1 - x_i, & \text{if } U(0,1) < p_m \\ x_i, & \text{otherwise} \end{cases} \tag{3}$$

Simple crossover generates a random number r from a uniform distribution from 1 to m and creates two new individuals ($\bar{X}\prime$ and $\bar{Y}\prime$) according to equations 4 and 5.

$$x_i' = \begin{cases} x_i, & \text{if } i < r \\ y_i, & \text{otherwise} \end{cases} \tag{4}$$

$$y_i' = \begin{cases} y_i, & \text{if } i < r \\ x_i, & \text{otherwise} \end{cases} \tag{5}$$

Operators for real-valued representations, i.e., an alphabet of floats, were developed by Michalewicz [Michalewicz 1994]. For real $\bar{X}$ and $\bar{Y}$, the following op-

erators are defined: uniform mutation, non-uniform mutation, multi-non-uniform mutation, boundary mutation, simple crossover, arithmetic crossover, and heuristic crossover. Let a_i and b_i be the lower and upper bound, respectively, for each variable i.

Uniform mutation randomly selects one variable, j, and sets it equal to an uniform random number $U(a_i, b_i)$:

$$x_i' = \begin{cases} U(a_i, b_i), & \text{if } i = j \\ x_i, & \text{otherwise} \end{cases} \tag{6}$$

Boundary mutation randomly selects one variable, j, and sets it equal to either its lower or upper bound, where $r = U(0, 1)$:

$$x_i' = \begin{cases} a_i, & \text{if } i = j, r < 0.5 \\ b_i, & \text{if } i = j, r \geq 0.5 \\ x_i, & \text{otherwise} \end{cases} \tag{7}$$

Non-uniform mutation randomly selects one variable, j, and sets it equal to an non-uniform random number:

$$x_i' = \begin{cases} x_i + (b_i - x_i) f(G) & \text{if } r_1 < 0.5, \\ x_i - (x_i + a_i) f(G) & \text{if } r_1 \geq 0.5, \\ x_i, & \text{otherwise} \end{cases} \tag{8}$$

where

$$\begin{aligned} f(G) &= (r_2(1 - \tfrac{G}{G_{max}}))^b, \qquad (9) \\ r_1, r_2 &= \text{a uniform random number between (0,1)}, \\ G &= \text{the current generation}, \\ G_{max} &= \text{the maximum number of generations}, \\ b &= \text{a shape parameter}. \end{aligned}$$

The multi-non-uniform mutation operator applies the non-uniform operator to all of the variables in the parent $\bar{X}$.

Real-valued simple crossover is identical to the binary version presented above in equations 4 and 5. Arithmetic crossover produces two complimentary linear combinations of the parents, where $r = U(0, 1)$:

$$\bar{X}' = r\bar{X} + (1 - r)\bar{Y} \tag{10}$$

$$\bar{Y}' = (1 - r)\bar{X} + r\bar{Y} \tag{11}$$

Heuristic crossover produces an linear extrapolation of the two individuals. This is the only operator that utilizes fitness information. A new individual, $\bar{X}'$, is created using equation 12, where $r = U(0, 1)$ and $\bar{X}$ is better than $\bar{Y}$ in terms of fitness. If $\bar{X}'$ is infeasible, i.e., $feasibility$ equals 0 as given by equation 14, then generate a new random number r and create a new solution using equation 12, otherwise stop. To ensure halting, after t failures, let the children equal the parents and stop.

$$\bar{X}' = \bar{X} + r(\bar{X} - \bar{Y}) \tag{12}$$
$$\bar{Y}' = \bar{X} \tag{13}$$

$$feasibility = \begin{cases} 1, \text{ if } x_i' \geq a_i, x_i' \leq b_i \ \ \forall i \\ 0, \text{ otherwise} \end{cases} \tag{14}$$

2.4 Initialization, Termination, and Evaluation Functions

The GA must be provided an initial population as indicated in step 1 of Fig. 1. The most common method is to randomly generate solutions for the entire population. However, since GAs can iteratively improve existing solutions (i.e., solutions from other heuristics and/or current practices), the beginning population can be seeded with potentially good solutions, with the remainder of the population being randomly generated solutions.

The GA moves from generation to generation selecting and reproducing parents until a termination criterion is met. The most frequently used stopping criterion is a specified maximum number of generations. Another termination strategy involves population convergence criteria. In general, GAs will force much of the entire population to converge to a single solution. When the sum of the deviations among individuals becomes smaller than some specified threshold, the algorithm can be terminated. The algorithm can also be terminated due to a lack of improvement in the best solution over a specified number of generations. Alternatively, a target value for the evaluation measure can be established based on some arbitrarily "acceptable" threshold. Several strategies can be used in conjunction with each other.

Evaluation functions of many forms can be used in a GA, subject to the minimal requirement that the function can map the population into a partially ordered set. As stated, the evaluation function is independent of the GA (i.e., stochastic decision rules).

3. TESTING AND CONCLUSIONS

The Matlab implementation of the algorithm has been tested with respect to efficiency and reliability by optimizing a family of multi-modal non-linear test problems. The family of test problems is taken from Corana, [Corana et al. 1987], which compare the use of the simulated annealing algorithm to the simplex method of Nelder-Mead and adaptive random search. In [Houck et al. 1995a] we report in detail the effectiveness of the genetic algorithm for solving the continuous location-allocation problem, and in [Houck et al. 1995b] on the use of the genetic algorithm in conjunction with local-improvement heuristics for non-linear function optimization, location-allocation, and the quadratic assignment problem.

The Corana family[Corana et al. 1987] of parameterized functions,q_n, are very simple to compute and contain a large number of local minima. This function is basically a n-dimensional parabola with rectangular pockets removed and where the global minima occurs at the origin $(0, 0, \ldots, 0)$. This family is defined as follows:

$$
\begin{aligned}
D_f &\equiv \{\mathbf{x} \in \Re^n : -a_1 \le x_1 \le a_1, \ldots, -a_n \le x_n \le a_n; \mathbf{a} \in \Re_+^n\} \\
d_{k_1,\ldots,k_n} &\equiv \begin{cases} \mathbf{x} \in D_f : k_1 s_1 - t_1 < x_1 < k_1 s_1 + t_1, \ldots, k_n s_n - t_n < x_n < k_n s_n + t_n; \\ \qquad k_1, \ldots, k_n \in \mathcal{Z}; \bar{t}, \bar{s} \in \Re_+^n; t_i < \frac{s_i}{2}, i = 1, \ldots, n \end{cases} \\
D_m &\equiv \bigcup_{k_1 \ldots, k_n \in \mathcal{Z}} d_{k_1,\ldots,k_n} - d_{0,0,\ldots,0} \\
D_r &\equiv D_f - D_m \\
q_n(\mathbf{x}) &\equiv \sum_{i=1}^{n} d_i x_i^2, \quad \mathbf{x} \in D_r, \mathbf{d} \in \Re_+^n, \\
q_n(\mathbf{x}) &\equiv \sum_{i=1}^{n} d_i z_i^2, \quad \mathbf{x} \in d_{k_1,\ldots,k_n}, (k_1, \ldots, k_n) \neq 0, \\
z_i &= \begin{cases} k_i s_i + t_i & \text{if } k_i < 0, \\ 0 & \text{if } k_i = 0, \\ k_i s_i - t_i & \text{if } k_i > 0, \end{cases}
\end{aligned}
$$

For the optimization of the test function two different representations were used. A real-valued alphabet was employed in conjunction with the selection, mutation and crossover operators with their respective options as shown in table I. Also, a binary representation was used in conjunction with the selection, mutation and crossover operators with their respective options as shown in table II. A description of the options for each of the functions is provided in the following section, Section 4.

Table I. GAOT Parameters used for Real-Valued Corana Function Optimization

Name	Parameters
Uniform Mutation	4
Non-Uniform Mutation	[4 G_{max} 3]
Multi-Non-Uniform Mutation	[6 G_{max} 3]
Boundary Mutation	4
Simple Crossover	4
Arithmetic Crossover	4
Heuristic Crossover	[2 3]
Normalized Geometric Selection	0.08

Table II. GAOT Parameters used for Binary Corana Function Optimization

Name	Parameters
Binary Mutation	0.05
Simple Crossover	0.6
Normalized Geometric Selection	0.08

Two different evaluation functions were used for both the float and binary genetic algorithm, the first simply returned the value of the Corana function at the point

as determined by the genetic string. The second evaluation function utilizes a Sequential Quadratic Programming (SQP) (available in Matlab) method to optimize the Corana function starting from the point as determined by the genetic string. This provides the genetic algorithm with a local improvement operator which, as shown in [Houck et al. 1995b], can greatly enhance the performance of the genetic algorithm. Many researchers have shown that GAs perform well for a global search but perform very poorly in a localized search [Davis 1991; Michalewicz 1994; Houck et al. 1995a; Bersini and Renders 1994]. GAs are capable of quickly finding promising regions of the search space but may take a relatively long time to reach the optimal solution.

Both the float genetic algorithm (FGA) and binary genetic algorithm (BGA) were run 10 times with different random seeds. The simulated annealing (SA) results are taken from the 10 replications of these test problems reported in [Corana et al. 1987]. The resulting solution value found and the number of function evaluations to obtain that solution are shown in Table III. Since Corana et al. did not use an improvement procedure, both the FGA and BGA were run without the use of SQP. As shown in the table, the FGA outperformed both BGA and SA in terms of computational efficiency and solution quality. With respect to the epsilon of $1e^{-6}$ as used in [Corana et al. 1987], FGA found the optimal in all three cases in all replications, while SA was unable to find the optimal two times for the 4 dimensional case and not at all for the 10 dimensional case. The table also shows that the use of the local improvement operator significantly increases the power of the genetic algorithm in terms of solution quality and speed of convergence to the optimal.

Table III. Solution Quality and Procedure Efficiency

Dim.	Method	Avg Sol.	Std. of Sol.	Min. Sol.	Avg. # of eval.	Std. # of eval.	Min # of eval.
2	FGA	$5.75e^{-7}$	$2.87e^{-7}$	$2.09e^{-7}$	$6.90e^{+3}$	$1.33e^{+3}$	$5.87e^{+3}$
	FGA-SQP	$0.00e^{+0}$	$0.00e^{+0}$	$0.00e^{+0}$	$6.02e^{+2}$	$1.89e^{+2}$	$3.95e^{+2}$
	BGA	$4.51e^{-7}$	$3.40e^{-7}$	$3.31e^{-8}$	$9.60e^{+3}$	$3.56e^{+3}$	$4.56e^{+3}$
	BGA-SQP	$8.45e^{-15}$	$2.67e^{-15}$	$5.40e^{-79}$	$8.48e^{+2}$	$2.61e^{+2}$	$6.03 + 2$
	SA	$1.13e^{-8}$	$1.42e^{-8}$	$4.21e^{-10}$	$6.89e^{+5}$	$1.73e^{+4}$	$6.56e^{+5}$
4	FGA	$6.80e^{-7}$	$3.35e^{-7}$	$1.58e^{-7}$	$1.06e^{+5}$	$5.56e^{+4}$	$4.81e^{+4}$
	FGA-SQP	$0.00e^{+0}$	$0.00e^{+0}$	$0.00e^{+0}$	$3.76e^{+3}$	$1.27e^{+3}$	$1.66e^{+3}$
	BGA	$5.34e^{-7}$	$2.99e^{-7}$	$3.57e^{-9}$	$3.07e^{+5}$	$7.25e^{+4}$	$1.93e^{+5}$
	BGA-SQP	$2.53e^{-9}$	$7.71e^{-9}$	$6.80e^{-26}$	$3.32e^{+4}$	$1.78e^{+4}$	$1.32e^{+4}$
	SA	$6.18e^{-4}$	$1.40e^{-3}$	$8.70e^{-8}$	$1.38e^{+6}$	$1.11e^{+5}$	$1.18e^{+6}$
10	FGA	$6.15e^{-7}$	$4.01e^{-7}$	$1.68e^{-8}$	$2.31e^{+5}$	$3.06e^{+4}$	$1.77e^{+5}$
	FGA-SQP	$0.00e^{+0}$	$0.00e^{+0}$	$0.00e^{+0}$	$5.38e^{+4}$	$3.29e^{+4}$	$3.80e^{+4}$
	BGA	$1.74e^{+2}$	$1.85e^{+2}$	$2.29e^{+1}$	$1.47e^{+6}$	$6.96e^{+4}$	$1.34e^{+6}$
	BGA-SQP	$5.74e^{+2}$	$1.09e^{+3}$	$4.23e^{+0}$	$8.26e^{+2}$	$1.63e^{+2}$	$5.27e^{+2}$
	SA	$5.40e^{-4}$	$0.00e^{+0}$	$5.40e^{-4}$	$1.62e^{+6}$	$3.65e^{+4}$	$1.55e^{+6}$

The results of this testing show that the use of genetic algorithms for function optimization is highly efficient and effective. The use of a local improvement

procedure, in this case SQP, can greatly enhance the performance of the genetic algorithm.

4. GAOT: A MATLAB IMPLEMENTATION

Matlab is a technical computing environment for high-performance numeric computation. Matlab integrates numerical analysis, matrix computation and graphics in an easy-to-use environment. User-defined Matlab functions are simple text files of interpreted instructions. Therefore, Matlab functions are completely portable from one hardware architecture to another without even a recompilation step.

The algorithm discussed in Section 2 has been implemented as a Matlab toolbox, i.e., a group of related functions, named GAOT, Genetic Algorithms for Optimization Toolbox. Each module of the algorithm is implemented using a Matlab function. This provides for easy extensibility, as well as modularity. The basic function is the **ga** function, which runs the simulated evolution. The basic call to the **ga** function is given by the following Matlab command.

```
[x,endPop,bPop,traceInfo] = ga(bounds,evalFN,evalParams,params,startPop,...
termFN,termParams,selectFN,selectParams,xOverFNs,xOverParams,mutFNs,mutParams)
```

Output parameters

—x is the best solution string, i.e. final solution,

—*endPop*(**optional**) is the final population,

—*bPop*(**optional**) is a matrix of the best individuals and the corresponding generation they were found,

—*traceInfo*(**optional**) is a matrix of maximum and mean functional value of the population for each generation.

Input parameters

—*bounds* is a matrix of upper and lower bounds on the variables,

—*evalFN* is the evaluation function, usually a .m file,

—*evalParams*(**optional**) is a row matrix of any parameters to the evaluation function defaults to $[NULL]$,

—*params*(**optional**) is a vector of options, i.e. [*epsilon prob_param disp_param*] where *epsilon* is the change required to consider two solutions different and *prob_params* is 0 if you want to use the binary version of the algorithm, or 1 for the float version. *disp_param* controls the display of the progress of the algorithm, 1 displays the current generation and the the value of the best solution in the population, while 0 prevents any output during the run. This parameter defaults to $[1e^{-6}\ 1\ 0]$.

—*startPop*(**optional**) is a matrix of solutions and their respective functional values. The starting population defaults to a randomly created population created with **initialize**,

—*termFN*(**optional**) is the name of the termination function which defaults to ['*maxGenTerm*'],

—*termParams*(**optional**) is a row matrix of parameters which defaults to [*100*],

—*selectFN*(**optional**) is the name of the selection function which defaults to [*'normGeomSelect'*],

—*selectParams*(**optional**) is a row matrix of parameters for the selection function which defaults to [*0.08*],

—*xOverFNs*(**optional**) is a blank separated string of the names of the cross-over functions which defaults to [*'arithXover heuristicXover simpleXover'*] for the float version and [*'simpleXover*] for the binary version.

—*xOverParams*(**optional**) is a matrix of the crossover parameters which default to [*2 0;2 3;2 0*] for the float version and [*0.6*] for the binary

—*mutFNs*(**optional**) is a blank separated string of mutation operators which default to [*'boundaryMutation multiNonUnifMutation nonUnifMutation unifMutation'*] for the float version and ['binaryMutation'] for the binary version.

—*mutParams*(**optional**) is a matrix of mutation parameters which defaults to *[4 0;6 100 3;4 100 3;4 0]* for the float version and [*0.05*] for the binary.

GA performs the simulated evolution using the *evalFN* to determine the fitness of the solution strings. The GA uses the operators *xOverFNs* and *mutFNs* to alter the solution strings during the search. The program has been run successfully on a DecStation 3100, a DecStation 5000/25, Motorolla 604 and an HP 715.

The system maintains a high degree of modularity and flexibility as a result of the decision to pass the selection, evaluation, termination functions to the GA as well as a list of genetic operators. Thus, the base genetic algorithm is able to perform evolution using any combination of selection, crossover, mutation, evaluation and termination functions that conform to the functional specifications as outlined below or can easily be used with the default parameters.

4.1 Evaluation Function

The evaluation function is the driving force behind the GA. The evaluation function is called from the GA to determine the fitness of each solution string generated during the search. An example evaluation function is given below:

```
function [x, val] = gaDemo1Eval(x,parameters)
val = x(1) + 10*sin(5*x(1))+7*cos(4*x(2));
```

To run the **ga** using this test function use either of the following function calls from Matlab.

```
bstX = ga([0 10; 0 -10],'gaDemo1Eval')
bstX = ga([0 10; 0 -10],'x(1) + 10*sin(5*x(1))+7*cos(4*x(2))');
```

where gaDemo1Eval.m is contains the evaluation function as given above. Usually, a .m file will be more convenient to use as the evaluation function will be more complex than the simple example provided. This function call will use all of the default parameters of the **ga** and return only the best solution found during the course of the simulated evolution.

Note that the evaluation function must take two parameters, x and *options*. x is a row vector of $n+1$ elements where the first n elements are the parameters of interest. The $n+1$'th element is the value of this solution. The *parameters* matrix is a row matrix of

```
[current_generation, evalParams]
```

The evaluation function must return both the value of the string, *val* and the string itself, x. This is done so that an evaluation can repair or improve the string if desired. This allows for the use of local improvement procedures as discussed in Section 3.

An evaluation function is unique to the optimization of the problem at hand therefore, every time the **ga** is used for a different problem, an evaluation function must be developed to determine the fitness of the individuals.

The remainder of this section describes the other modules of the genetic toolbox. While GAOT allows for easy modification of any of these modules, the defaults as given work well for a wide class of optimization problems as shown in [Houck et al. 1995b].

4.2 Operator Functions

Operators provide the search mechanism of the GA. The operators are used to create new solutions based on existing solutions in the population. There are two basic types of operators, crossover and mutation. Crossover takes two individuals and produces two new individuals while mutation alters one individual to produce a single new solution. The **ga** function calls each of the operators to produce new solutions. The function call for crossovers is as follows:

```
[c1,c2] = crossover(p1,p2,bounds,params)
```

where *p1* is the first parent, [*solution_string function_value*], *p2* is the second parent, *bounds* is the bounds matrix for the solution space and *params* is the vector of *[current generation, operatorParams]*, where *operatorParams* is the appropriate row of parameters for this crossover/mutation operator. The first value of the *operatorParams* is frequency of application of this operator. For the float ga, this is the discrete number of times to call this operator every generation, while for the binary ga it is the probability of application to each member of the population. The mutation function call is similar, but only takes one parent and returns one child:

```
[c1] = mutation(p1,bounds,params)
```

The crossover operator must take all four arguments, the two parents, the bounds of the search space, the information on how much of the evolution has taken place and any other special options required. Similarly, mutations must all take the three arguments and return the resulting child. Table IV shows the operators implemented in Matlab, their corresponding file names, and any options that the operator takes in addition to the first option, the number of applications per generation.

4.3 Selection Function

The selection function determines which of the individuals will survive and continue on to the next generation. The **ga** function calls the selection function each generation after all the new children have been evaluated to create the new population from the old one.

The basic function call used in **ga** for selection is:

Table IV. Matlab Implemented Operator Functions

Name	File	Options
Arithmetic Crossover	arithXover.m	none
Heuristic Crossover	heuristicXover.m	number of retries (t)
Simple Crossover	simpleXover.m	none
Boundary Mutation	boundary.m	none
Multi-Non-Uniform Mutation	multiNonUnifMut.m	max num of generations, shape parameter (b)
Non-Uniform Mutation	nonUnifMut.m	max num of generations, shape parameter (b)
Uniform Mutation	unifMut.m	none

```
[newPop] = selectFunction(oldPop,options)
```

where *newPop* is the new population selected, *oldPop* is the current population, and *options* is a vector for any other optional parameters.

Notice that all selection routines must take both parameters, the old population from which to select members from, and any specific options to that particular selection routine. The function must return the new population. Table V shows the selection routines that have been implemented in GAOT. The file names are provided, as they are the function names to be used in Matlab, and the options for each function is also provided.

Table V. Matlab Implemented Selection Functions

Name	File	Options
Roulette Wheel	roulette.m	None
Normalized Geometric Select	normGeomSelect.m	Probability of Selecting Best
Tournament	tourn.m	Number of individuals in each tournament

4.4 Initialization and Termination Functions

Initialization of a population to provide the **ga** a starting point is usually done by generating random strings within the search space, and this is the default behavior of the **ga** function. However, it is possible to 'seed' the initial population with individuals, or generate solutions in some other form. The **ga** allows for this with the optional *startPop* parameter which provides the **ga** with an explicit starting population.

The termination function determines when to stop the simulated evolution and return the resulting population. The **ga** function calls the termination function once every generation after the application of all of the operator functions and the evaluation function for the resulting children. The function call is of the format:

```
done = terminateFunction(options,bestPop,pop)
```

where *options* is a vector of termination options the first of which is always the current generation. *bestPop* is a matrix of the best individuals and the respective generation it was found. *pop* is the current population. Table VI shows the termination routines that have been implemented in GAOT. The file names are provided

as they are the function names to be used in Matlab, and the options for each function is also provided.

Table VI. Matlab Implemented Termination Functions

Name	File	Options
Terminate at Specified Generation	maxGenTerm.m	final generation
Terminate at Optimal or max gen	maxGenOptTerm.m	final generation, optimal value, epsilon

4.5 Online Tutorial

Several Matlab demos are provided as a tutorial to the genetic algorithm toolbox. The first demo, *gademo1*, gives a brief introduction to GAs using a simple one variable function. The second demo, *gademo2*, uses a more complicated example, the 4-dimensional Corana function, to further illustrate the use of the toolbox. The final demo, *gademo3*, is a reference to the format used for the operator, selection, evaluation, and termination functions.

5. SUMMARY

A genetic algorithm capable of either using a floating point representation or a binary representation has been implemented as a Matlab toolbox. This toolbox provides a modular, extensible, portable algorithm in an environment rich in mathematical capabilities. The toolbox has been tested on a series of non-linear, non-convex, multi-modal functions. The results of these tests show that the algorithm is capable of finding better solutions with less function evaluations than simulated annealing.

REFERENCES

Bersini, H. and Renders, B. 1994. Hybridizing genetic algorithms with hill-climbing methods for global optimization: Two possible ways. In *1994 IEEE International Symposium Evolutionary Computation*, Orlando, Fl, pp. 312–317.

Corana, A., Marchesi, M., Martini, C., and Ridella, S. 1987. Minimizing multimodal functions of continuous variables with the "simulated annealing" algorithm. *ACM Transactions on Mathematical Software 13,* 3, 262–280.

Davis, L. 1991. *The Handbook of Genetic Algorithms.* Van Nostrand Reingold, New York.

Goldberg, D. 1989. *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley.

Holland, J. 1975. *Adaptation in natural and artificial systems.* The University of Michigan Press, Ann Arbor.

Houck, C., Joines, J., and Kay, M. 1995a. A comparison of genetic algorithms, random restart, and two-opt switching for solving large location-allocation problems. *Computers & Operations Research forthcoming in special issue on evolution computation.*

Houck, C., Joines, J., and Kay, M. 1995b. The effective use of local improvement procedures in conjunction with genetic algorithms. Technical Report NCSU-IE Technical Report 95, North Carolina State University.

Joines, J. and Houck, C. 1994. On the use of non-stationary penalty functions to solve constrained optimization problems with genetic algorithms. In *1994 IEEE International Symposium Evolutionary Computation*, Orlando, Fl, pp. 579–584.

MICHALEWICZ, Z. 1994. *Genetic Algorithms + Data Structures = Evolution Programs.* AI Series. Springer-Verlag, New York.