A HIERARCHICAL GENETIC SYSTEM FOR SYMBOLIC FUNCTION IDENTIFICATION

Mingda Jiang (cs__mj@selway.umt.edu)
Alden H. Wright (cs_ahw@selway.umt.edu)

Department of Computer Science
University of Montana
Missoula, MT 59812

Jan. 20, 1992

Abstract

Given data in the form of a collection of (x,y) pairs of real numbers, the symbolic function identification problem is to find a functional model of the form y = f(x) that fits the data. This paper describes a system for solution of symbolic function identification problems that combines a genetic algorithm and the Levenberg-Marquardt nonlinear regression algorithm. The genetic algorithm uses an expression-tree representation rather than the more usual binary-string representation. Experiments were run with data generated using a wide variety of function models. The system was able to find a function model that closely approximated the data with a very high success rate.

Keywords: genetic algorithm, nonlinear regression, function identification, machine learning

1 INTRODUCTION

The function identification problem is a fundamental problem of science, medicine, and engineering. The function identification problem is to find a functional model of an experimental system, in symbolic form, that fits the experimental data points produced by the system. In this problem, fundamental properties of an experimental system are to be determined from observed behavior of that system. The functional model is a mathematical idealization that is used as an approximate model of the system, and is chosen to give good fit to the given experimental data according to a chosen evaluation criterion.

This paper describes a computer program to solve the one-variable symbolic function identification problem. Researchers in artificial intelligence (machine learning) have investigated the mechanisms of machine discovery and designed some machine learning systems to find empirical laws (function models) from the observations, such as BACON (Langley, Bradshaw, & Simon, 1983; Langley, Simon, Bradshaw, & Zytkow, 1987), ABACUS (Falkenhainer & Michalski, 1986), FAHRENHEIT (Zytkow, 1987), and IDS (Nordhausen & Langley, 1990). These systems can solve relatively simple concept learning tasks of function identification problems and "rediscover" such classical scientific laws as Ohm's law, Newton's law of universal gravitation, Kepler's law, and Snell's law of refraction from the given sample data points. Most of the methods these systems used for symbolic function identification problems are typically based on heuristic rule sets which guide the search process. However, most function identification problems are not readily approached by standard rule-based search techniques. For instance, it is difficult to apply heuristic rules to general function identification problems. Not only are the combinations of rules that govern the movement of learning systems very large, but the rules themselves need to change. Rules that might work in one realm need adjustment to work in another realm. The characteristically large search spaces pose formidable obstacles for traditional search techniques. The combinatorial complexity of such problems is a major deterrent to the application of most simple solution strategies.

One technique for solving these kinds of problems, called genetic algorithms, comes from analogy to nature. These algorithms are an outgrowth of a theory of adaptation developed by John Holland (Holland 1975). They are motivated by standard models of heredity and evolution in the field of population genetics, embodying abstractions of the mechanisms of adaptation present in natural systems. Nature provides the best demonstration of the power of

genetic search wherein the best suited "structures" of organisms evolve in response to environmental pressures. This genetic metaphor encompasses a wide range of (domain independent) search strategies. It is a very flexible way to get computers to learn how to solve problems for themselves. Genetic algorithms have been successfully applied to function optimization problems (DeJong 1975, Brindle 1981, Mühlenbein 1991, Wright 1991), maze walk problem (Smith 1980), robotic planning, and automatic programming (Koza 1989, 1990). The authors combine Koza's hierarchical genetic algorithm and the Levenberg-Marquardt nonlinear regression algorithm to build the general-purpose machine function identification system which can fits a functional model to a given sample data set without any priori knowledge about the functional model of the experimental system.

Mühlenbein and his coworkers (Mühlenbein 1987, 1989, 1991) have suggested that local optimization precede function evaluation in the use of genetic algorithms for optimization. They have had excellent success in applying a parallel genetic algorithm that includes this feature to difficult combinatorial and real-parameter optimization problems.

In this paper, a nonlinear regression method, the Levenberg-Marquardt algorithm, is used as a local optimization method before function evaluation.

2 NONLINEAR REGRESSION

Regression analysis is a statistical technique for investigating and modeling the relationship between variables. Suppose that we have n pairs of observation data points (x_1, y_1), (x_2, y_2), ..., (x_N, y_N), and the regression equations are as following

$$y_k = f(x_k, B) + \varepsilon_k \qquad (2.1)$$

where f(x, B) is the known response function, x is an independent variable, $B = [b_1, b_2, \ldots, b_M]^T$ is an M-dimensional vector of parameters to be estimated, the ε_k represents a random error with mean zero and unknown variance σ^2, and the subscript k = 1, 2, ..., N ranges over the N observations. The sequence of values of the independent variable $\{x_k\}$ is treated as a fixed known sequence of constants, not as a random variable (Gallant 1987).

A linear regression model is the model in which all the parameters appear linearly. That is, the response variable

y has some linear relationship with the unknown parameters b_1, b_2, ..., b_M. For instance, the following two regression models are linear in the parameters.

$$y=b_1+b_2x+b_3x^2+b_4x^3 \qquad where\ B=[b_1\ b_2\ b_3\ b_4]^T$$

$$y=b_1+b_2x+b_3e^x \qquad where\ B=[b_1\ b_2\ b_3]^T$$

A nonlinear regression model is one in which at least one of its parameters appears nonlinearly. Frequently, nonlinear regression models arise in instances where a specific scientific discipline specifies the form that data ought to follow, and this form is nonlinear in the parameters. For instance, the logistic model,

$$y = \frac{\alpha}{1.0 + \gamma e^{\beta x}}$$

which produces sigmoidal or "S-shaped" growth curves, is nonlinear. This model is widely used in biology, agriculture engineering, and economics. In this case $B = [\alpha\ \beta\ \gamma]^T$.

A univariate nonlinear regression model can be written in a convenient vector form

$$Y = F(X,B) + E \qquad (2.2)$$

where

$$Y = [y_1\ y_2\ \ldots\ y_N]^T,$$

$$F(B) = [f(x_1, B)\ f(x_2, B)\ \ldots\ f(x_N, B)]^T,$$

$$E = [\varepsilon_1\ \varepsilon_2\ \ldots\ \varepsilon_N]^T.$$

The error sum of squares for the nonlinear model and the given data is defined as

$$S(B) = \sum_{k=1}^{N} [y_k - f(x_k, B)]^2 \qquad (2.3)$$

Using vector form, equation (2.3) becomes

$$S(B) = [Y-F(B)]^T[Y-F(B)] = |H(B)|^2 \qquad (2.4)$$

where

$$H(B) = Y - F(B)$$

$$= [\{y_1 - f(x_1,B)\}\ \{y_2 - f(x_2,B)\}\ \ldots\ \{y_N - f(x_N,B)\}]^T$$

$$= [h_1(B)\ h_2(B)\ \ldots\ h_N(B)]^T$$

Note that since y_k and x_k are fixed observations, the sum of squared residuals is a function of B. We denote by B_{min}, a least squares estimate of B, that is a value of B which minimizes S(B). To find the least squares estimate B_{min} we need to differentiate equation (2.3) with respect to B. This provides the M normal equations, which must be solved for B_{min}.

The normal equations take the form

$$\sum_{k=1}^{N} \{y_k - f(x_k, B_{min})\}\left[\frac{\partial f(x_k,B)}{\partial b_i}\right]_{B=B_{min}} = 0 \qquad (2.5)$$

for i = 1, 2, ..., M, where the quantity denoted by brackets is the derivative of $f(x_k, B)$ with respect to b_i with all B's replaced by the corresponding B_{min}'s which have the same subscript.

There are several algorithms for the estimation of nonlinear models. The Levenberg-Marquardt algorithm works very well in practice and has become the standard of nonlinear least-squares routines (Press 1988). The iteration equation to solve equation (2.5) using the Levenberg-Marquardt method is as follow:

$$[J^T(B_i)J(B_i) + \lambda^2 I](B_{i+1} - B_i) = -J^T(B_i)H(B_i) \qquad (2.6)$$

where λ^2 is a chosen scaler. The matrix J is the N*M Jacobian matrix of the vector H(B).

$$J(B_i) = \left[\begin{array}{ccc} \frac{\partial h_1(B)}{\partial b_1} & \cdots & \frac{\partial h_1(B)}{\partial b_M} \\ \vdots & \ddots & \vdots \\ \frac{\partial h_N(B)}{\partial b_1} & \cdots & \frac{\partial h_N(B)}{\partial b_M} \end{array}\right]_{B=B_i}$$

The Levenberg-Marquardt algorithm is as follows (Press, Flannery, Teukolsky, and Vetterling 1988):

(1) Give an initial guess for the set of fitted parameters $B_0 = [b_1^{(0)}, b_2^{(0)}, \ldots, b_M^{(0)}]^T$;
(2) Compute the sum of squared residuals $S(B_0)$ using the equation (2.3);
(3) Pick a modest value for λ, say $\lambda = 0.001$, Set

iterative count K and J to be 0;
(4) While (J < 2) and (K < MaxIterations) Do
(a) K <-- K + 1;
(b) Solve the linear equation (2.5) for dB_K and evaluate $S(B_K + dB_K)$ where $dB_K = B_{K+1} - B_K$;
(c) If $S(B_k + dB_k) < S(B_k)$ and
$|S(B_k + dB_k) - S(B_k)| < 0.001$
J <-- J + 1
Else
J <-- 0;
(d) If $S(B_K + dB_K) \geq S(B_K)$, increase λ by a factor of 10 (or any other substantial factor) and go back to (4);
(e) If $S(B_K + dB_K) < S(B_K)$, decrease λ by a factor of 10, update the trial solution B_{K+1} <-- $B_K + dB_K$, and go back to (4);
(5) End

Figure 1: The Levenberg-Marquardt algorithm

Note that there are two cutoff values used to stop iterative calculation in our implementation of the Levenberg-Marquardt nonlinear regression algorithm. The first one is the convergence criterion for the sum of squared residuals. When S(B) decreases by less than 0.001 on two consecutive iterations, the fit is considered complete. The second one is the maximum number of iterations allowed, MaxIterations.

The Levenberg-Marquardt algorithm appears to enlarge considerably the number of practical problems that can be tackled by nonlinear estimation. The Levenberg-Marquardt's method represents a compromise between the linearization (Gauss-Newton) method and the steepest descent method and appears to combine the best features of both while avoiding their most serious limitations. It is good in that it almost always converges and does not "slow down" as the steepest descent method often does (Draper & Smith 1981).

3 GENETIC ALGORITHMS

Genetic algorithms are randomized search algorithms based on the mechanics of natural selection and natural genetics (Goldberg 1989). A population of potential solutions to the problem is generated randomly. These potential problem solutions are called individuals, and each individual is represented by a "chromosome". Each individual is assigned a fitness that represents how well it solves the problem. In an optimization problem, the fitness is normally a rescaling of the objective function. In

HGSFI, each individual consists of a function model together with values of parameters for the function model.

Most genetic algorithms use a generational approach. The process of producing a new generation involves two major phases. In the selection phase, the more highly fit individual are reproduced (cloned), and the less fit individuals are deleted from the population. In the genetic operator phase, genetic operators are applied to the population to generate new individuals. The most commonly used genetic operators are recombination (crossover) where two individuals are combined to produce one or two new individuals, and mutation, where a single indiviudal is modified in a random way to produce a new individual. It is expected that the genetic operators have some chance of producing individuals that are superior to their parents. The genetic algorithm goes through a sequence of generations.

The most common way of representing the chromosomes of individuals is as a bit-string. However, many genetic algorithms that have done well in practice have used a chromosome representation that is tailored to the particular problem.

3.1 KNOWLEDGE REPRESENTATION

Knowledge representation is a key issue in any AI (learning) system because the representation scheme can severely limit the window by which the system observes its world. Using a genetic algorithm, one represents strategies as chromosomes. Each chromosome serves a dual purpose: it provides a representation of the problem solution, and it also provides the actual material which can be transformed to yield new genetic material for the next generation.

To represent various complex function models, we chose the hierarchical structure representation to represent chromosomes (or population members), which was developed by John R. Koza (Koza 1989 and 1990). In a hierarchical structure representation, population members for a particular domain of interest are represented by expression trees. For instance, the cubic polynomial model

$$y = c_1 + c_2X + c_3X^2 + c_4X^3$$

can be represented as the following expression tree.

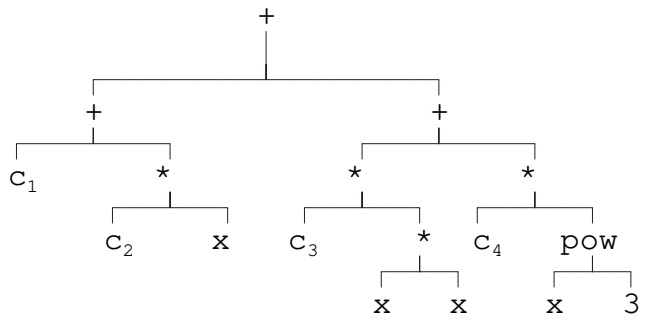

Figure 2: One expression tree for the cubic polynomial model

Note that this representation is not unique for the function model. Another expression tree to represent the cubic polynomial model is shown in Figure 3.

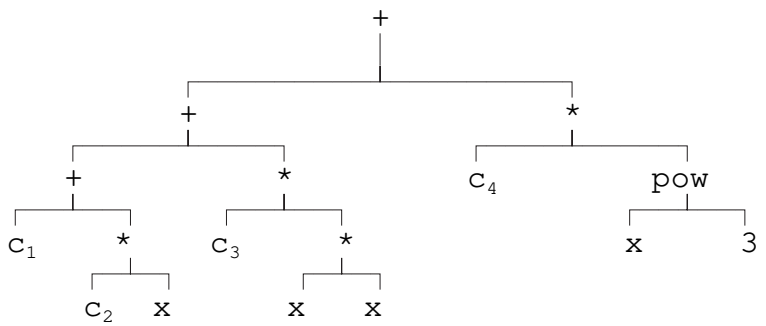

Figure 3: Another expression tree for the cubic polynomial model

Suppose that in a function identification problem the available set of n functions is **F**={f_1, f_2, ..., f_n} and the available set of m terminals is **T**={a_1, a_2, ..., a_m}. The "terminals" may be variable atomic arguments or constants. Each particular function f in **F** takes a specified number z(f) of arguments b_1, b_2, ..., $b_{z(f)}$. Depending on the particular problem of interest, the functions may be standard arithmetic operations (such as addition, subtraction, multiplication, and division), standard mathematical functions (such as exp, sin, etc.).

The solution domain is the set of valid function models that can be recursively created by compositions of the available function operations and the available terminals for the problem. This is represented in the computer as the set of expression trees with a terminal at each leaf and an function operation at each interior node.

Using the hierarchical structure representation scheme, we can easily represent a wide variety of function models of different size, shape, and complexity, can dynamically modify the knowledge structures (function models) in the operations of crossover and mutation during learning process, and can efficiently evaluate the values of function models. In our system, the knowledge (i.e. function models) are represented as expression trees.

3.2 GENERATION OF THE INITIAL POPULATION

Genetic algorithms begin with an initial population. Generation of the initial random population in HGSFI begins by selecting one of the functions from the set **F** at random to be the root of the tree. For an interior node with an k-argument function, k lines are created to radiate out from the node. Then, for each line so created, an element is selected at random from entire combined set **S** = **F** U **T** (which is the set of functions and terminals) to be the label for the endpoint of that line. If a terminal is chosen to be any node, the process is then complete for that portion of the tree. If a function is chosen to be the label for any such node, the process continues.

The probability distribution over the terminals and functions in the combined set **S** and the number of arguments taken by each function implicitly determines an average size for the trees generated by this initial random generation process. In genetic algorithms, this distribution is usually a uniform random probability distribution over the entire set **S** (with the exception that the root of the tree must be a function).

In our test problems, all initial populations are randomly generated as above. In addition, the initial function model trees are controlled by a system parameter, InitMaxDepth, which specifies the maximum depth of the randomly generated initial function model trees.

3.3 THE FITNESS FUNCTION

Statistians usually use the following two methods to describe the difference between the predicted model f(x, B) and real value y_k (see equation 2.1). The first method is the sum of the squares between the given values of observations and computed values of the dependent variable. That is

The another one is the sum of absolute residuals

$$SSResid = \sum_{k=1}^{N} [y_k - f(x_k, B)]^2$$

$$SAResid = \sum_{k=1}^{N} | y_k - f(x_k, B) |$$

If we choose the sum of squared residuals to be the performance measure, then we have a performance measure function which is consistent with that of nonlinear regression algorithms. However, the sum of squared residuals is not directly proportional to the residual of each point. The sum of absolute residuals has a linear relationship with the residual of each point. Our experimental results indicated that using the sum of absolute residuals as the performance measure worked better than using the sum of squared residuals. In our system, the performance measure Perf(i, t) of an individual function model i in the population at generation t is defined as follows:

$$Perf(i,t) = \sum_{k=1}^{N} | y_k - f_i^{(t)}(x_k, B_i^{(t)}) | \qquad (3.1)$$

where $f_i^{(t)}(x_k, B_i^{(t)})$ is the function equation of individual i at generation t. $B_i^{(t)}$ is the estimated parameter vector of individual i at generation t. The smaller the performance, the better an individual in the population. We might try to directly use this performance to calculate the fitness as follow:

$$f(i,t) = 1 - \frac{Perf(i,t)}{\sum_{k=1}^{P_S} Perf(k,t)}$$

where P_S is the population size. This gives a fitness that is always between 0 and 1, where better individuals have higher fitness. However, the performance of the worst individual in a population might be much larger than the performances of other individuals in the population. Then

$$\sum_{k=1}^{P_S} Perf(k,t) \doteq \text{the performance of the worst guy}$$

In this case, the fitness of the worst individual would be very close to 0, and the fitness of the others would be very close to 1. This fitness function would do a poor job of

distinguishing the relative performances of the better individuals in the population. In other words, this performance measure does not stress good performance; the genetic algorithm may fail to converge on good results in a reasonable time and will be more likely to lose the best members of its population. For this reason, we need to employ a normalization technique for the performance measure. We define the normalized performance of individual i at generation t as following

$$Perf_{Norm}(i,t) = \frac{1}{1+SAResid(i,t)} = \frac{1}{1+\sum_{k=1}^{N} \left| y_k - f_i^{(t)}(x_k, B_i^{(t)}) \right|} \qquad (3.2)$$

Then, the fitness of individual i at generation t is calculated as

$$f(i,t) = \frac{Perf_{Norm}(i,t)}{\sum_{k=1}^{P_S} Perf_{Norm}(k,t)} \qquad (3.3)$$

The fitness lies between 0 and 1 and is larger for better individuals in the population. The sum of the fitness values in a population is 1. Then the expected number of individuals cloned from an individual in the selection phase is this normalized fitness times the population size.

3.4 GENETIC OPERATORS

The genetic operators manipulate individual structures in a population and produce new structures. The three primary operations for modifying the structures undergoing adaptation are Darwinian fitness proportionate reproduction, crossover (recombination), and mutation. A fourth genetic operator, permutation is also used in this paper.

Reproduction implements the process of choosing individuals for the next generation from the individuals in the current generation. The individuals in reproduction pool (new generation) are chosen from old population by a randomized reproduction procedure that ensures that the expected number of times an individual is chosen is approximately proportional to that individual's fitness.

In the crossover operation, individuals in the reproduction pool are randomly chosen for crossover, employing a user-specified crossover probability. Crossover

can be implemented as the following. First, two parents are chosen from the reproduction pool randomly. Then one tree node in each parent is selected randomly and independently according to a probability distribution. The two offspring are produced by exchanging the subtrees lying below the crossover points. For instance, consider the two parental expression trees in Figure 4.

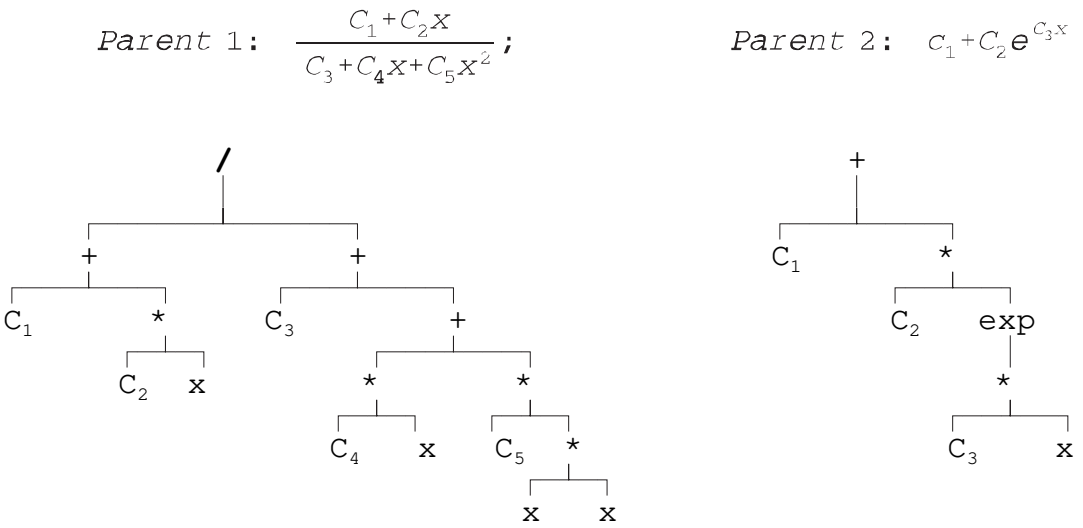

Figure 4: The two parental expression trees

Assume that the points of trees are numbered in a depth-first way starting at the left. Suppose that the seventh node (out of the 17 nodes of the first parent) was selected as the crossover node for the first parent and that the fifth node (out of the 8 nodes of the second parent) was selected as the crossover node of the second parent. The two crossover subtrees are shown in Figure 5.

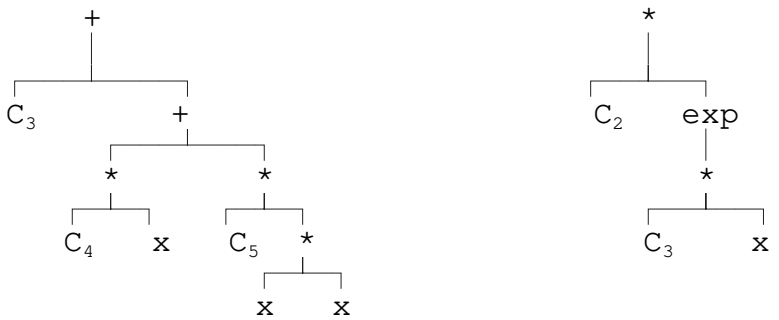

Figure 5: The two crossover subtrees

The two offspring resulting from crossover are shown in Figure 6.

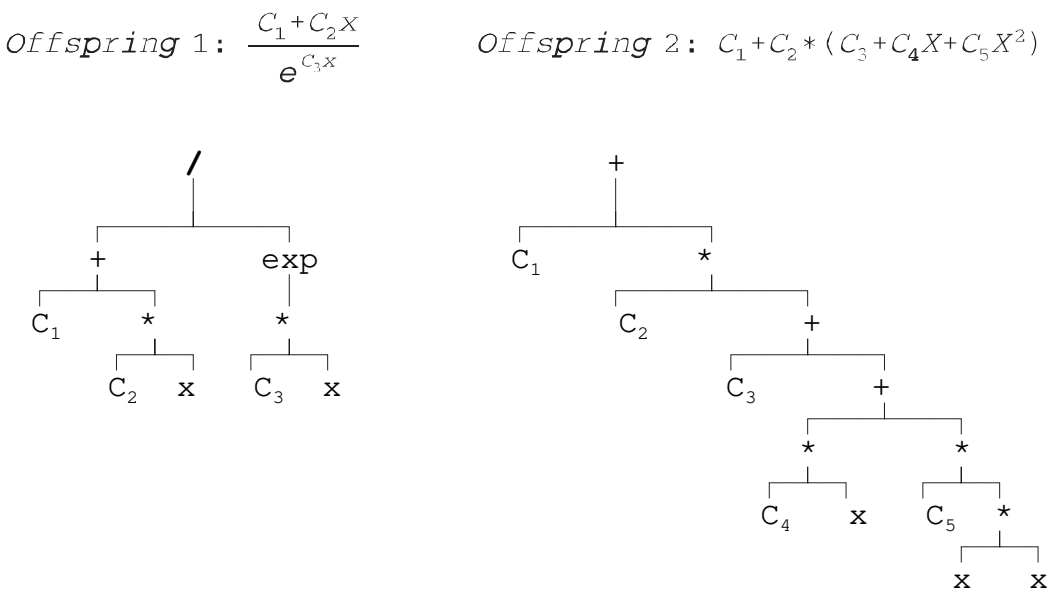

Figure 6: The two offspring resulting from crossover

After crossover, a small number of random nodes of some individuals in reproduction pool are selected and changed through mutation. The mutation operation selects an individual from the new generation and a node of the selected expression tree at random. Then this operation randomly chooses a member from the set **S** of terminals and functions to replace the function or terminal at the selected node. (We can think a constant and an independent variable as a special function operation which has no arguments.) If the new function operation has different number of arguments with old function operation at the selected node, the mutation operation needs to add a randomly generated tree branch to the node or delete the extra branch from the selected node.

The permutation operation selects an individual from new population, and a function (internal) node of the selected individual randomly. If the function at the selected node is a binary function, then left child and right child are swapped. Note that if the function at the selected node is commutative, there is no immediate effect from the permutation operation on the individual tree.

4 THE FRAMEWORK OF HGSFI

The Figure 7 contains the overall algorithm used by HGFSI.

```
BEGIN
   (1)  Choose the initial population of size Ps, call this
        population P(0).  Set the generation counter t to be
        0;
   (2)  Optimize (using the Levenberg-Marquardt algorithm)
        the constants in each evaluated individual;
   (3)  Evaluate the performance of each individual in P(t);
   (4)  Measure each individual in P(t) to check convergence
        (if an individual meets the system convergence
        condition, set ConvergedFlag);
   (5)  If ConvergedFlag is on or
           generation counter t ≥ maximum generation
        then terminate;
   (6)  Increase generation counter by 1;
   (7)  Reproduce P(t) from P(t-1), call population P(t) as
        new_pop, population P(t-1) as old_pop;
   (8)  Crossover P(t);
   (9)  Mutate P(t);
   (10) Permute P(t);
   (11) If EliteFlag is on then call Elitist (to insure that
        the best member of P(t-1) is saved into P(t);
   (12) For each individual (call i) in P(t) do
        (i)  If new_pop[i].needEvaluation then
             (a) If SimplifyFlag is on then Simplify the
                 new_pop[i].model of individual i at
                 generation t;
             (b) Count the number of nodes in
                 new_pop[i].model;
             (c) Count the number of constants in
                 new_pop[i].model;
             (d) Build constant list of individual i at
                 generation t;
        (ii) Endif;
   (12) Go to (2);
END.
```

Figure 7: The main loop of HGSFI

The kernel of HGSFI is the genetic algorithm. HGSFI is implemented using the C programming language (Jiang 1991). The main loop of HGSFI is an iterative procedure which maintains a constant-size population P(t) of candidate solutions. The main loop is shown in Figure 7.

Note that the new generation function models and their fitness are analyzed by the **measure** procedure with the given performance criterion. The **measure** procedure compares the fitness of individuals in the population to determine the "best" individual. It measures each individual function model to check if it has converged. This procedure calculates the convergence test value of individual i at

generation t as the following

$$convg(i,t) = 1.0 - \frac{\sum_{k=1}^{N} | y_k - f_i^{(t)}(x_k, B) |}{\sum_{k=1}^{N} | y_k |} \qquad (4.1)$$

where $f_i^{(t)}(x, B)$ is the function equation of individual i at generation t. If the convergence test value for some individual is greater than the system convergence condition (e.g. 0.99), then the function model has converged and the system variable ConvergedFlag is set as TRUE.

There are three option processes in HGSFI. The option (e) of the **elitist** process stipulates that the individual with the best fitness in the current generation always survives intact to next generation. The elite individual is placed in the last position in new generation, and is not changed through crossover, mutation, or permutation. In the absence of the elitist strategy (i.e., EliteFlag is off), it is possible for the best individual to disappear, due to sampling error, crossover, mutation, or permutation. The option (o) of the Levenberg-Marquardt nonlinear regression process indicates whether the system includes the optimizer for a particular running in order to compare the two approaches. The third option process is the option (s) of simplifying function models. If this option is chosen, the system variable SimplifyFlag is set as TRUE. This means that during each iteration after processing genetic operations, the function models of the modified individuals will be simplified using the set of simplification rules. For instance, the equation

$$C_1 * (C_2 + C_3 * X)$$

can be simplified as

$$C_1' + C_2' * X$$

where $C_1' = C_1 * C_2$ and $C_2' = C_1 * C_3$. Using the simplification option can save computer resources: space and time. It may improve overall performance slightly for some problems. However, our experience showed that it could not improve overall performance for most of linear and nonlinear regression model problems we tested.

5 EXPERIMENTAL RESULTS

We used a wide range of function models from many task domains to test and evaluate the performance of HGSFI machine function identification system. These experiments were run under the UNIX (Ultrix) operating system on a DEC-5500 system. The HGSFI program was compiled using gcc (GNU

project C Compiler) version 1.36.

First we did some experiments to compare the system performance with and without the Levenberg-Marquardt optimization. Section 5.1 presents the details of this comparison. Then we chose two sets of task domains to test HGSFI. Section 5.2 discusses the results of the experiments using linear regression model problems. The experimental results on nonlinear regression model problems are presented in section 5.3.

For all of the test problems, we randomly generated initial function models. The only prior knowledge used was the primitive functions, such as addition, multiplication, exponentiation, etc., that were used in representing the population of function models.

Running HGSFI entails setting a number of control parameter values. Let HGSFI = (**S**, P_S, R_c, R_m, R_p, C_v, **O**) represent a particular running of HGSFI with the settings of control parameters for the given sample data points. Note that **S** is the set of terminals and functions. P_S is the population size. R_c is the crossover rate which controls the frequency with which the crossover operation is applied. R_m is the mutation rate. R_p is the permutation rate. C_v is the convergence value which serves as the convergence condition of the system. Usually we chose the convergence value C_v to be 0.99. This means that the system was considered to be successful in solving the given problem if at least one individual function model had the average of absolute differences between the predicted values of the model f(x, B) and the given values of observations within 1% of the average of absolute values of observations (see equation (4.1)). **O** is the option set, which includes the option of the elitist selection strategy (e), the option of the LM optimization (o), and the option of simplifying function model (s).

In addition, there are three control parameters which had the same values for all runs of our test problems. First, the maximum number of generations was set to be 50. Second, the maximum depth of initial function models was 4. Third, the maximum depth of function models was 15, which prevents large amounts of computer time being expended on the extremely complicated function models.

5.1 COMPARISON OF THE SYSTEM PERFORMANCE WITH AND WITHOUT LEVENBERG-MARQUARDT OPTIMIZATION

This set of experiments was conducted to compare the system performance with and without the Levenberg-Marquardt optimization. The test function equations are listed in Table 1. The sample data (no noise) of the test problems

Table 1: The first set of test problems

Problem	Function Equation	X range	Pts
5.1	$Y = X + X^2 + X^3 + X^4$	[-2.0,2.0]	100
5.2	$Y = \sin(X) + \cos(X) + X + X^2$	[-5.0,5.0]	150
5.3	$Y = \cos(X + X)$	[0.0, 6.0]	20
5.4	$Y = X^2 - X^7$	[-5.0,5.0]	150
5.5	$Y = 3.1416*X+2.718*X^2$	[-5.0,5.0]	100
5.6	$Y = 0.808162*\exp(-1.21*X)$	[-2.0,8.0]	100
5.7	$Y = -2.3+3.0*X+0.45*X^2-1.23*X^3$	[-5.0,5.0]	100
5.8	$Y = 21.10-19.81*\exp(-0.00177*X^{3.180})$	[0.0,14.0]	140

were uniformly generated using the test function equations in the given X ranges with the given number of points. For the first set of test problems, except for problem (5.8), we chose the following control parameters

```
HGSFI = ({c, x, +, -, *, /, exp, log, sin, cos},
         500, 0.9, 0.02, 0.0, 0.99, {e})
```

to run the HGSFI without the option of LM optimization, where c is a random constant and x is the independent variable. The control parameters for problem (5.8) without the Levenberg-Marquardt optimization were

```
HGSFI = ({c, x, +, -, *, /, **, sq, exp, log, sin, cos},
         500, 0.9, 0.02, 0.0, 0.99, {e}).
```

where sq(X) means X^2 and X**C means X^C. For the runs with the Levenberg-Marquardt optimization, the option of LM optimization was added to the control parameters. The experimental results for these test problems with and without LM optimization are shown in Tables 2 and 3, respectively.

We defined a run to be successful for the given sample data set if at least one function model in the run satisfied the given system convergence condition within the maximum number of generations.

Table 2: Experimental results of the first set of test problems without LM optimization

Prob-lem	Tot-al runs	Suc-cess Rate (%)	Avg. gen. for suc. runs	CPU time per run (min)	Average test value for suc. runs	Average test value for all runs	Average best $R_{squares}$ for suc. runs
5.1	30	43.3	32.85	5.411	.998040	.965313	.999980
5.2	30	73.3	23.73	6.190	.996840	.987427	.999962
5.3	25	100	3.36	0.048	.999633	.999633	.999997
5.4	30	0.0	–	4.274	–	.449476	–
5.5	30	56.7	32.17	6.702	.992964	.983683	.999894
5.6	30	50.0	25.47	8.549	.994589	.976314	.999965
5.7	30	0.0	–	10.82	–	.919131	–
5.8	30	0.0	–	2.825	–	.890589	–

In the tables, the **total runs** column contains the total number of runs for the given problem. The **success rate** means the ratio of the number of successful runs to the total number of runs for a given problem. The converged generation for a successful run is the generation at which at least one function model satisfies the system convergence condition within the maximum number of generations. The **avg. gen. for suc. runs** means the average number of the converged generations for successful runs. Note that the generation counter started from 1 instead of 0 when the average number of converged generations was calculated for successful runs. That is, initial generation is 1, the first generation is 2, and so on. The **CPU time per run** is the average CPU time for all runs. The average test values for successful runs and all runs are the averages of the convergence test values (equation 4.1) of the best function models for successful runs and all runs, respectively. The average best $R_{squares}$ for successful runs is the average of $R_{squares}$ of the best function models for successful runs. Note that the definition of $R_{squares}$ is

$$R_{squares} = 1 - \frac{\sum_{k=1}^{N} [\, y_k - f(x_k, B)\,]^2}{\sum_{k=1}^{N} [\, y_k - y_{avg}]^2}$$

where y_{avg} is the average value of y_k (k=1, 2, ..., N).

Table 3: Experimental results of the first set of test problems with LM optimization

Prob-lem	Tot-al runs	Suc-cess rate (%)	Avg. gen. for suc. runs	CPU time per run (min)	Average test value for suc. runs	Average test value for all runs	Average best $R_{squares}$ for suc. runs
5.1	26	100	10.5	37.58	.998411	.998411	.999994
5.2	20	100	10.65	79.33	.998450	.998450	.999983
5.3	30	100	1.066	0.425	.999952	.999952	1.00000
5.4	30	80.0	31.13	84.44	.997754	.923638	.999957
5.5	20	100	3.65	6.94	.999929	.999929	1.00000
5.6	20	100	2.05	3.92	.999880	.999880	.999994
5.7	50	100	10.44	30.10	.998385	.998385	.999992
5.8	49	100	14.31	107.4	.993469	.993469	.999791

If we define "the correct function model" for a problem as:

(1) after simplification, the form of "the correct function model" is same as the form of the function equation generated the sample data, except for the difference of some constant term, and

(2) its $R_{squares} > 0.999999$,

then the success rates for finding the correct function models of problems (5.1) to (5.8) with and without LM optimization are listed in Table 4.

Table 4: The success rate for finding the correct function models of the first set of test problems

Problem	Total runs without LM	Success rate without LM option	Total runs with LM	Success rate with LM option
5.1	30	30.00%	26	53.85%
5.2	30	30.00%	20	60.00%
5.3	25	96.00%	30	100%
5.4	30	0.00%	30	23.33%
5.5	30	3.33%	20	95.00%
5.6	30	3.33%	20	85.00%
5.7	30	0.00%	50	56.00%
5.8	30	0.00%	49	0.00%

Comparing the system performance with and without the option of LM optimization, the average convergence test value of all runs and the average $R_{squares}$ of successful runs with nonlinear regression were better than those without nonlinear regression. The success rate with LM optimization was much higher than that without LM optimization except in some extremely simple problems such as problem (5.3). In addition, the average number of the converged generations for successful runs with LM optimization was less than that without LM optimization. In other words, the system performance with the Levenberg-Marquardt optimization was better than that without the Levenberg-Marquardt optimization. However, the average CPU time to solve the given problems with LM optimization was longer than that without LM optimization, because the optimizer spent a lot of time to optimize the coefficients of function models.

5.2 LINEAR REGRESSION MODEL PROBLEMS

For this set of experimental problems, we used the function equations in Table 5 to generate the corresponding experimental data (no noise). For this set of test problems, we chose the following control parameters:

```
HGSFI = ({c, x, +, -, *, /, exp, log, sin, cos},
          500, 0.9, 0.02, 0.0, 0.99, {e, o}).
```

Note that LM optimization was included on all runs for this set of test problems. The experimental results of the second set of test problems are listed in Table 6.

Table 5: The second set of test problems

Prob-lem	Function Equation	X range	Pts
5.7	$Y = -2.3+3.0*X+0.45*X^2-1.23*X^3$	[-5.0,5.0]	100
5.9	Y = 3.2*sin(X)+2.5*cos(X)	[-2.0, 18]	200
5.10	Y = 0.45+6.032*exp(-X)	[0.0,20.0]	100
5.11	Y = 0.1-2*X+0.5*log(X)	(0.0,20.0]	100

The success rates for finding the correct function models of the second set of problems by HGSFI with LM optimization are listed in Table 7. Note that the success rate for finding the correct function models of problem 5.11 was quite low. But when we increased the convergence value to 0.999, the success rate was 11/30, or 36.7%.

Table 6: Experimental results of the second set of test problems with LM optimization

Prob-lem	Tot-al runs	Suc-cess Rate (%)	Avg. Gen. for suc. runs	CPU time per run (min)	Average test value for suc. runs	Average test value for all runs	Average best $R_{squares}$ for suc. runs
5.7	50	100	10.44	30.10	.998385	.998385	.999992
5.9	20	100	4.8	16.62	.999776	.999776	1.00000
5.10	23	95.7	10.36	95.75	.999231	.991547	.999993
5.11	25	100	1.0	2.04	.996607	.996607	.999830

For almost all of the linear regression model problems we tested, HGSFI rapidly converged to the function equations that were used to generate the sample data.

Table 7: The success rate for finding the correct function models of the second set of test problems

Problem	5.7	5.9	5.10	5.11
Total runs	50	20	23	25

Success rate with LM option	56.00%	85.00%	82.61%	8.0%

5.3 NONLINEAR REGRESSION MODEL PROBLEMS

For the third set of test problems, we use the function equations in Table 8 to generate the experimental data (no noise).

The settings of control parameters for problems (5.12) and (5.13) were

```
HGSFI = ({c, x, +, -, *, /, **, sq, exp, log, sin, cos},
         500, 0.9, 0.02, 0.0, 0.999, {e, o}).
```

The only difference of the settings of control parameters to run the HGSFI for problems (5.8) and (5.14) was the convergence value. The latter used 0.99 instead of 0.999. The summary of experimental results of these test problems is shown in Table 9.

Table 8: The third set of test problems

Prob-lem	Function Equation	X range	Pts
5.8	$Y = 21.10 - 19.81 * \exp(-0.00177 * X^{3.180})$	[0.0,14.0]	140
5.12	$Y = X/(0.3 + 0.06*X)$	[0.0,10.0]	100
5.13	$Y = 10.0/(1.0 + 25.0*\exp(-0.8*X))$	[0.0,10.0]	100
5.14	$Y = 0.6*(e^{-0.3X} - e^{-0.6X})/(0.6-0.3)$	[0.0,20.0]	100

This set of experimental problems was challenging for the current implementation of HGSFI. Function discovery and nonlinear regression are both global optimization problems. To discover the "correct" function model and the "correct" values of the coefficients of the function model, the system needs to search a large function space and a large constant space. The two together make the problem very hard. In addition, so far there are no known efficient global optimization algorithms for nonlinear regression. The HGSFI implementation used the Levenberg-Marquardt optimization algorithm to find the "best" fit coefficients of function models for the given sample data set, which is a local optimization algorithm. If a function model is "correct"

for the given problem but the starting values of the coefficients of the function model are not around the global optimizing point, then the poor starting values may result in convergence to an unwanted stationary point of the sum of squares surface (Draper and Smith 1981). This unwanted point may have coefficient values which do not provide the true minimum value of S(B) (equation 2.3). Then the "correct" function model may be thrown away due to the poor fitness.

Table 9: Experimental results of the third set of test problems with LM optimization

Prob-lem	Tot-al runs	Suc-cess Rate (%)	Avg. Gen. for suc. runs	CPU time per run (min)	Average test value for suc. runs	Average test value for all runs	Average best $R_{squares}$ for suc. runs
5.8	49	100	14.31	107.4	.993469	.993469	.999791
5.12	20	100	8.4	99.84	.999674	.999674	.999996
5.13	30	96.7	18.55	350.3	.999344	.998610	.999998
5.14	20	95.0	11.42	152.5	.995469	.994697	.999951

For these nonlinear regression model problems, HGSFI still quickly found function models to fit the given sample data points with a very high success rate. Sometimes it found the function model that was used to generate the sample data set. For example, the "best" function model HGSFI found for problem (5.12) in a run was

$$y = \frac{x}{0.30000+0.06000x}$$

which is equivalent to the function equation (5.12). In most situations, however, the HGSFI found a high-degree polynomial function model to fit the given sample data points. The reasonably good fit function model HGSFI found in a run for problem (5.8) was equivalent to

$$y = 1.257+0.3169X-0.39541X^2+0.19104X^3-0.01958X^4+0.000598X^5 .$$

The success rates for finding the correct function models of the third set of problems by HGSFI with LM optimization are listed in Table 10.

Table 10: The success rate for finding the correct function models of the third set of test problems

Problem	5.8	5.12	5.13	5.14
Total runs	49	20	30	20
Success rate with LM option	0.00%	20.00%	0.00%	0.00%

6 SUMMARY AND CONCLUSIONS

The HGSFI system combines a hierarchical genetic algorithm and the Levenberg-Marquardt nonlinear regression algorithm to solve symbolic function identification problems. The only a-priori knowledge needed about the function model is the primitive functions used to construct the function model.

Experimental results showed that using the Levenberg-Marquardt algorithm gave much better preformance than a system that used only a hierarchical genetic algorithm. This result provides further evidence that using local optimization in genetic algorithms can improve preformance.

On experiments run using data generated using 14 different function models, HGSFI was able to find a function model that closely approximated the data with a high success rate. (Closely approximated meant that the sum of the absolute deviations was less than 1% of the sum of the absolute y values.) Using at least 20 runs per problem, HGSFI succeeded in this sense 100% of the time for 10 of the problems, and 80%, 96%, 97% and 95% of the time for the remaining 4 problems.

HGSFI was less successful at finding the same function model that generated the data. Especially for complex nonlinear models, HGSFI was likely to find a high-degree poloynomial that closely approximated the data. In part, this is because the space of parameterized function models will include many function models that closely approximate a given data set. In addition, a polynomial model that approximates the data reasonably well can be improved by adding on additional high-degree terms with small coefficients.

Limited experiments show that choosing a stricter convergence criterion would improve the ability of HGSFI to find the function model that generated the data. We

speculate that a search strategy that forces HGSFI to concentrate on simpler models in early generations before going on to more complex models in later generations would substantially improve the performance of HGSFI in finding the "correct" function model.

REFERENCES

Brindle, A. (1981), **Genetic Algorithms for Function Optimization**, Ph.D. dissertation, University of Alberta, Edmonton, Canada

Davis, Lawrence (1987), ed., **Genetic Algorithms and Simulated Annealing**, Los Altos, CA: Morgan Kaufmann

DeJong, K.A. (1975), Analysis of the Behavior of a Class of Genetic Adaptive Systems, Ph.D. dissertation, Department of Computer and Communication Science, University of Michigan

Draper, N.R. and Smith, H. (1981), **Applied Regression Analysis**, 2nd ed., New York: John Wiley & Sons

Falkenhainer, Brian C. and Michalski, Ryszard S. (1986), Integrating Quantitative and Qualitative Discovery: the ABACUS System, **Machine Learning** 1, 367-401

Gallant, A. Ronald (1987), **Nonlinear Statistical Models**, New York: John Wiley & Sons

Goldberg, David E. (1989), **Genetic Algorithms in Search, Optimization, and Machine Learning**, Reading, Massachusetts: Addison-Wesley

Holland, John H. (1975), **Adaptation in Natural and Artificial Systems**, Ann Arbor, MI: University of Michigan Press

Jiang, Mingda (1991), A Machine Function Identification System Based on Genetic Algorithms, Master's thesis, Department of Computer Science, University of Montana

Kennedy, W. J. and Gentle, J.E. (1980), **Statistical Computing**, New York: Dekker

Koza, John R. (1989), Hierarchical Genetic Algorithms Operating on Populations of Computer Programs, P**roceedings of the 11th International Joint Conference on Artificial Intelligence (IJCAI)**, San Mateo, CA: Morgan Kaufmann

Koza, John R. (1990), Genetic Programming: a Paradigm for Genetically Breeding Populations of Computer Programs to Solve Problems, Technical Report STAN-CS-90-1314, Computer Science Department, Stanford University

Langley, P., Bradshaw, G.L., and Simon, H.A. (1983), **Rediscovering chemistry with the BACON system**, in R.S. Michlski, J.G. Carbonell, & T.M. Mitchell (Eds.), **Machine Learning: An artificial intelligence approach**, San Mateo, CA: Morgan Kaufmann

Langley, P., Simon, H.A., Bradshaw, G.L., and Zytkow, J.M. (1987), **Scientific Discovery: Computational Explorations of the Creative Processes**, Cambridge, MA: MIT Press

Mühlenbein, H., Gorges-Schleuter, M., and Krämer, O. (1987), New solutions to the mapping problem of parallel systems - the evolution approach. **Parallel Computing** 6, 269-279

Mühlenbein, H. (1989), Parallel genetic algorithms,

population genetics and combinatorial optimization, in Schaffer, J. David (1989), ed., **Proceedings of the Third International Conference on Genetic Algorithms**, San Mateo, CA: Morgan Kaufmann

Mühlenbein, H., Schomisch, M., and Born, J. (1991) The parallel genetic algorithm as function optimizer, **Parallel Computing**

Nordhausen, Bernd and Langley, P. (1990), An Integrated Approach to Empirical Discovery, in J. Shrager & P. Langley (Eds.), **Computational Models of Scientific Discovery and Theory Formation**, San Mateo, CA: Morgan Kaufmann

Press, William H., Flannery, Brian P., Teukolsky, Saul A., and Vetterling, William T. (1988), **Numerical Recipes in C: The Art of Scientific Computing**, New York: Cambridge University Press

Ratkowsky, David A. (1983), **Nonlinear Regression Modeling: A Unified Practical Approach**, New York: Marcel Dekker

Smith, S.F. (1980), **A Learning System Based on Genetic Adaptive Algorithms**, Ph.D. dissertation, Dept. of Computer Science, University of Pittsburgh

Wright, Alden H. (1991), Genetic Algorithms for Real Parameter Optimization, in Gregory J.E. Rawlins (Eds.), **Foundations of Genetic Algorithms**, San Mateo, CA: Morgan Kaufmann, 205–218

Zytkow, J. (1987), Combining many searches in the FAHRENHEIT discovery system, **Proceedings of the Fourth International Workshop on Machine Learning** 281–287, Irvine, CA: Morgan Kaufmann