# USE OF GENETIC PROGRAMMING TO FIND AN IMPULSE RESPONSE FUNCTION IN SYMBOLIC FORM( *)

John R. Koza (1), Martin A. Keane (2) and James P. Rice (3)

*Abstract: The recently developed genetic programming paradigm provides a way to genetically breed a computer program to solve a wide variety of problems. Genetic programming starts with a population of randomly created computer programs and iteratively applies the Darwinian reproduction operation and the genetic crossover (sexual recombination) operation in order to breed better individual programs. This paper illustrates how genetic programming can be used to find the symbolic form of a good approximation to the impulse response function for a linear time-invariant system using only the observed response of the system to a particular forcing function.*

*Keywords: Genetic programming, system identification, impulse response function, control engineering*

## 1 INTRODUCTION AND OVERVIEW

Genetic programming provides a way to search the space of all possible functions composed of certain terminals and primitive functions to find a function which solves a problem. In this paper, we use genetic programming to find an approximation to the impulse response function, in symbolic form, for a linear time-invariant system using only the observed response of the system to a particular known forcing function.

(*) Submitted September 21, 1992 to RAIRO journal.

(1) Computer Science Department, 224 Margaret Jacks Hall, Stanford University, Stanford, California 94305, Koza@cs.stanford.edu, 415-941-0336

(2) Third Millennium Venture Capital Limited, 5733 Grover, Chicago, Illinois 60630, 312-777-1524

(3) Stanford University Knowledge Systems Laboratory, 701 Welch Road, Palo Alto, California 94304, Rice@sumex-aim.stanford.edu, 415-723-8405

## 2 BACKGROUND ON GENETIC METHODS

Since the invention of the genetic algorithm by John Holland [1975], the genetic algorithm has proven successful at finding an optimal point in a search space for a wide variety of problems [Goldberg 1989, Davis 1987, Davis 1991, Michalewicz 1992, Belew and Booker 1991, Whitley 1992].

However, for many problems the most natural representation for the solution to the problem is an function (i.e., a composition of primitive functions and terminals), not merely a single numerical point in the search space of the problem. The size, shape, and contents of the composition needed to solve the problem is generally not known in advance.

The book *Genetic Programming: On the Programming of Computers by Means of Natural Selection* [Koza 1992a] describes the recently developed genetic programming paradigm and demonstrates that populations of computer programs (i.e., compositions of primitive functions and terminals) can be genetically bred to solve a surprising variety of different problems in a wide variety of different fields. Specifically, genetic programming has been successfully applied to problems such as

- discovering optimal control strategies (e.g., centering a cart and balancing a broom on a moving cart in minimal time by applying a "bang bang" force to the cart) [Koza and Keane 1990],
- discovering control strategies for backing up a tractor-trailer truck [Koza 1992b],
- discovering inverse kinematic equations to control the movement of a robot arm to a designated target point [Koza 1992a],
- evolution of a subsumption architecture for robotic control [Koza 1992c],
- symbolic "data to function" regression, symbolic integration, symbolic differentiation, and symbolic solution to general functional equations (including differential equations with initial conditions, and integral equations) [Koza 1992a], and
- empirical discovery (e.g., rediscovering the well-known non-linear econometric "exchange equation" MV = PQ from actual, noisy time series data for the money supply, the velocity of money, the price level, and the gross national product of an economy) [Koza 1992a, Koza and Rice 1992b, Koza 1992c].

A videotape visualization of 22 applications of genetic programming can be found in the *Genetic Programming: The Movie* [Koza and Rice 1992]. See also Koza [1992e].

In genetic programming, the individuals in the genetic population are compositions of primitive functions and terminals appropriate to the particular problem domain. The set of primitive functions used typically includes arithmetic operations, mathematical functions, conditional logical operations, and domain-specific functions. The set of terminals used typically includes inputs appropriate to the problem domain and various constants.

The compositions of primitive functions and terminals described above correspond directly to the parse tree that is internally created by most compilers and to the programs found in programming languages such as LISP (where they are called symbolic expressions or S-expressions).

One can now view the search for a solution to the problem as a search in the hyperspace of all possible compositions of functions that can be recursively composed of the available primitive functions and terminals.

## 3 STEPS REQUIRED TO EXECUTE GENETIC PROGRAMMING

Genetic programming, like the conventional genetic algorithm, is a domain independent method. It proceeds by genetically breeding populations of compositions of the primitive functions and terminals (i.e., computer programs) to solve problems by executing the following three steps:

(1) Generate an initial population of random computer programs composed of the primitive functions and terminals of the problem.
(2) Iteratively perform the following sub-steps until the termination criterion has been satisfied:
    (a) Execute each program in the population and assign it a fitness value according to how well it solves the problem.
    (b) Create a new population of programs by applying the following two primary operations. The operations are applied to program(s) in the population selected with a probability based on fitness (i.e., the fitter the program, the more likely it is to be selected).
        (i) *Reproduction*: Copy existing programs to the new population.
        (ii) *Crossover*: Create two new offspring programs for the new population by genetically recombining randomly chosen parts of two existing programs.

(3) The single best computer program in the population produced during the run is designated as the result of the run of genetic programming. This result may be a solution (or approximate solution) to the problem.

## 3.1 Crossover Operation

The genetic crossover (sexual recombination) operation operates on two parental computer programs and produces two offspring programs using parts of each parent.

For example, consider the following computer program (LISP symbolic expression):

```
(+ (* +0.234 Z) (- X 0.789)),
```

which we would ordinarily write as

$$0.234\ Z + X - 0.789.$$

This program takes two inputs (X and Z) and produces a floating point output. In the prefix notation used, the multiplication function `*` is first applied to the terminals `0.234` and `Z` to produce an intermediate result. Then, the subtraction function `-` is applied to the terminals `X` and `0.789` to produce a second intermediate result. Finally, the addition function `+` is applied to the two intermediate results to produce the overall result.

Also, consider a second program:

```
(* (* Z Y) (+ Y (* 0.314 Z))),
```

which is equivalent to

$$ZY\ (Y + 0.314\ Z).$$

In figure 1, these two programs are depicted as rooted, point-labeled trees with ordered branches. Internal points (i.e., nodes) of the tree correspond to functions (i.e., operations) and external points (i.e., leaves, endpoints) correspond to terminals (i.e., input data). The numbers on the function and terminal points of the tree appear for reference only.

The crossover operation creates new offspring by exchanging sub-trees (i.e., sub-lists, subroutines, subprocedures) between the two parents.

Assume that the points of both trees are numbered in a depth-first way starting at the left. Suppose that the point no. 2 (out of 7 points of the first parent) is randomly selected as the crossover point for the first parent and that the point no. 5 (out of 10 points of the second parent) is randomly selected as the crossover point of the second parent. The crossover points in the trees above are therefore the `*` in the first parent and the `+` in the second parent. The two crossover fragments are the two sub-trees shown in figure 2.

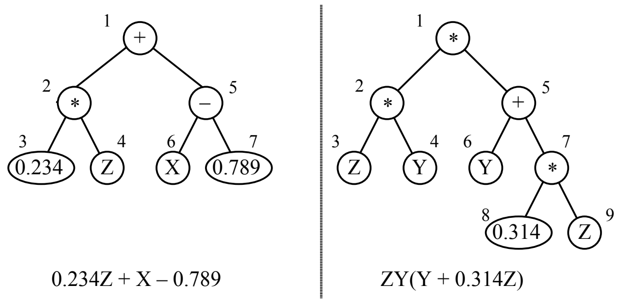

**Figure 1 Two Parental computer programs**

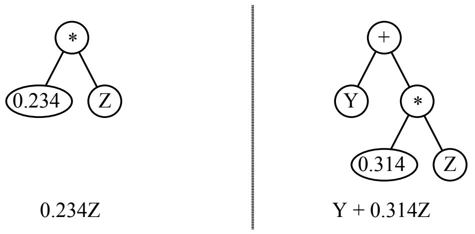

**Figure 2 Two Crossover Fragments**

These two crossover fragments correspond to the underlined sub-programs (sub-lists) in the two parental computer programs. The two offspring resulting from crossover are

```
(+ (+ Y (* 0.314 Z)) (- X 0.789))
```

and

```
(* (* Z Y) (* +0.234 Z)).
```

The two offspring are shown in figure 3.

Thus, crossover creates new computer programs using parts of existing parental programs. Because entire sub-trees are swapped, this crossover operation always produces syntactically and semantically valid programs as offspring regardless of the choice of the two crossover points. Because programs are selected to participate in the crossover operation with a probability based on fitness, crossover allocates future trials to parts of the search space whose programs contains parts from promising programs.

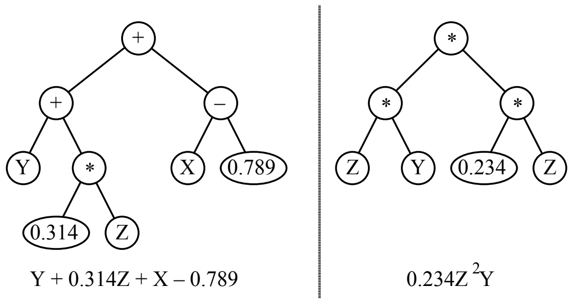

**Figure 3 Two Offspring**

## 4 FINDING AN IMPULSE RESPONSE FUNCTION

For many problems of control engineering, it is desirable to find a function, such as the impulse response function or transfer function (or an approximation to such function), for a system for which one does not have an analytical model. The reader unfamiliar with control engineering should focus on the fact that we are searching the space of possible computer programs (i.e., functions) for a program which meets certain requirements.

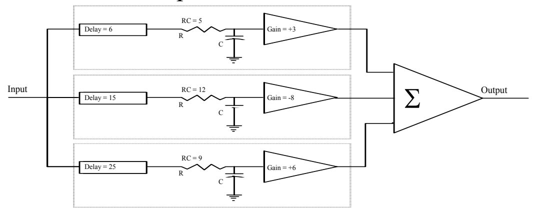

**Figure 4 Linear time-invariant system**

Figure 4 shows a linear time-invariant system (plant) which sums the output of three major components, each consisting of a pure time delay element, a lag circuit containing a resistor and capacitor, and a gain element. For example, for the first component of this system, the time delay is 6, the gain is +3, and the time constant, *RC*, is 5. For computational simplicity and without loss of generality, we use the discrete-time version of this system in this paper.

In this problem, one is given the response of the unknown system to a particular input. Figure 5 shows (a) a particular square input *i(t)* that rises from an amplitude of 0 to 1 at time 3 and falls back to an amplitude of 0 at time 23 and

(b) the response *o(t)* of the linear time-invariant system when this square input is used as a forcing function.

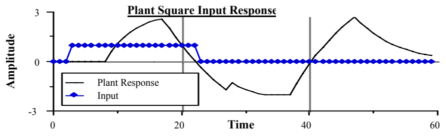

**Figure 5 Plant response when a square input is the forcing function**

The output of a linear time-invariant system is the discrete-time convolution of the input *i(t)* and the impulse response function *H(t)*. That is,

$$o(t) = \sum_{\tau=-\infty}^{t} i(t-\tau)\, H(\tau)\,.$$

The impulse response function *H(t)* for the system is known to be

$$\begin{cases} 0 \text{ if } t < 6 \\ \dfrac{3(1-\frac{1}{5})^{t-6}}{5} \end{cases} + \begin{cases} 0 \text{ if } t < 15 \\ \dfrac{-8(1-\frac{1}{12})^{t-15}}{12} \end{cases} + \begin{cases} 0 \text{ if } t < 25 \\ \dfrac{6(1-\frac{1}{9})^{t-25}}{9} \end{cases}$$

Figure 6 shows the impulse response function *H(t)* for this system.

We now show how an approximation to this impulse response can be discovered via genetic programming using just the observed output from the square forcing function. The discrete-time version of the square input and the system's discrete-time response to it shown in figure 5 are what is given in this problem; the goal is to find the impulse response of figure 6 or a good approximation to it. Note that although we know the correct impulse response function, genetic programming does not have access to it.

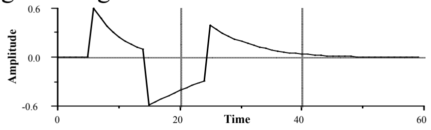

**Figure 6 Impulse response function**

## 5 PREPARATORY STEPS FOR USING GENETIC PROGRAMMING

There are five major steps in preparing to use genetic programming, namely, determining

(1) the set of terminals,
(2) the set of primitive functions,
(3) the fitness measure,
(4) the parameters for controlling the run, and
(5) the method for designating a result and the criterion for terminating a run.

The first major step in preparing to use genetic programming is the identification of the set of terminals. The individual impulse response functions in the population are compositions of the primitive functions and terminals. The single independent variable in the impulse response function is the time `T`. In addition, the impulse response function may contain various numerical constants. The ephemeral random floating point constant, ←, takes on a different random floating point value between –10.000 and +10.000 whenever it appears in an individual in the initial random population. Thus, terminal set $\mathcal{T}$ for this problem consists of

$\mathcal{T}$ = {T, ←}.

The second major step in preparing to use genetic programming is the identification of a sufficient set of primitive functions from which the impulse response function may be composed. For this problem, it seems reasonable for the function set to consist of the four arithmetic operations, the exponential function `EXP`, and the decision function `IFLTE` ("If Less Than or Equal"). Thus, the function set $\mathcal{F}$ for this problem is

$\mathcal{F}$ = {+, -, *, %, EXP, IFLTE},

taking 2, 2, 2, 2, 1, and 4 arguments, respectively. Of course, if we had some particular knowledge about the plant being analyzed suggesting the utility of certain other functions (e.g., sine), we could also include those functions in $\mathcal{F}$.

Each computer program in the population is a composition of primitive functions from the function set $\mathcal{F}$ and terminals from the terminal set $\mathcal{T}$. In this paper, we use the prefix notation (as used in programming languages such as LISP) in writing these compositions. Thus, we would write

$4.567 + 0.234\ T$

as the program

```
(+ 4.567 (* T 0.234)).
```

wherein the primitive function + is applied to the two arguments 4.567 and the result of applying the primitive function * to the two arguments T and 0.234.

Each primitive function in the function set should be well defined for any combination of arguments from the range of every primitive function that it may encounter and every terminal that it may encounter.

Since genetic programming operates on an initial population of randomly generated compositions of the available functions and terminals (and later performs genetic operations, such as crossover, on these individuals), it is necessary to protect against the possibility of division by zero and the possibility of creating extremely large or small floating point values. Accordingly, the protected division function `%` ordinarily returns the quotient; however, if division by zero is attempted, it returns 1.0. The one-argument exponential function `EXP` ordinarily returns the result of raising *e* to the power indicated by its one argument. If the result of evaluating any function would be greater than $10^{10}$ or less than $10^{-10}$, then the nominal value $10^{10}$ or $10^{-10}$, respectively, is returned.

The four-argument conditional branching function `IFLTE` evaluates and returns its third argument if its first argument is less than or equal to its second argument and otherwise evaluates and returns its fourth argument. For example, `(IFLTE 2.0 3.5 A B)` evaluates to the value of `A`.

The third major step in preparing to use genetic programming is the identification of the fitness measure for evaluating the goodness of an individual impulse response function in the population. For this problem, the fitness of an individual impulse response function in the population is measured in terms of the difference between the known observed response of the system to a particular forcing function and the response computed by convolving the individual impulse response function and the forcing function. The smaller the difference, the better. The exact impulse response of the system would lead to a difference of zero.

Specifically, each individual in the population is tested against a simulated environment consisting of $N_{fc}$ = 60 fitness cases, each representing the output *o(t)* of the given system for various times between 0 and 59 when the particular square input *i(t)* shown in figure 5 is used as the forcing function for the system. The fitness of any given genetically produced individual impulse response function *G(t)* in the population is the sum, over the $N_{fc}$ = 60 fitness cases, of the squares of the differences between the observed response *o(t)* of the system to the forcing function *i(t)* (i.e., the square input) and the response computed by convolving the forcing function and the given genetically produced impulse response *G(t)*. That is, the fitness is

$$f(G) = \sum_{i=1}^{N_{fc}} \left[ o(t_i) - \sum_{\tau=-\infty}^{t} i(t_i - \tau)\, G(\tau) \right]^2 .$$

The fourth major step in preparing to use genetic programming is the selection of values for certain parameters. Our choice of 4,000 as the population size and our choice of 50 as the maximum number of generations to be run reflect an estimate on our part as to the likely complexity of the solution to this problem and the limitations of available computer time and memory. Our choice of values for the various secondary parameters that control the run of genetic programming are the same default values as we have used on numerous other problems [Koza 1992a].

Finally, the fifth major step in preparing to use genetic programming is the selection of the criterion for terminating a run and the selection of the method for designating a result. We terminate a run after 51 generations.

## 6 RESULTS FOR ONE RUN

A review of one particular run will serve to illustrate how genetic programming progressively approximates the desired impulse response function.

One would not expect any individual from the randomly generated initial population to be very good. In generation 0, the fitness of the worst individual impulse response function among the 4,000 individuals in the population was very poor; its fitness was the enormous value of $4.7 \times 10^{77}$. This worst-of-generation individual consisted of 7 points (i.e., functions and terminals) and was

```
(- T (* (EXP T) (EXP T))),
```

which is equivalent to

$$t - e^{2t}.$$

However, even in a randomly created created population, some individuals are better than others. The fitness of the worst 40% of the population was $10^{10}$ (or worse). The median individual was, when simplified, equivalent to

$$-9.667 - 2.407t$$

and had a fitness of 10,260,473.

The fitness of the best individual impulse response function was 93.7 (i.e., an average squared error of about 1.56 for each of the 60 fitness cases). This best-of-generation individual program had 7 points (i.e., functions and terminals) and was

```
(% (% -2.46 T) (+ T -9.636)),
```

which is generally equivalent to

$$\frac{\frac{-2.46}{t}}{t - 9.636} = \frac{-2.46}{t^2 - 9.636t}.$$

Figure 7 compares (a) the genetically produced best-of-generation individual impulse response function from generation 0 and (b) the correct impulse response *H(t)* for the system previously shown in figure 6. As can be seen, there is little resemblance between this best-of-generation individual and the correct impulse response. Indeed, the sign of the values returned by this best-of-generation individual are incorrect for all but a few values of time.

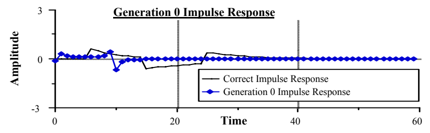

**Figure 7 Comparison of best genetically produced individual from generation 0 with the correct impulse response function**

In succeeding generations, the fitness of the worst-of-generation individual in the population, the median individual, and the best-of-generation individual all tend to improve (i.e., drop). In addition, the average fitness of the population as a whole tends to improve. The fitness of the best-of-generation individual dropped to 81.88 for generation 3, 76.09 for generation 5, 70.65 for generation 7, and 48.26 for generations 8 and 9. Of course, the vast majority of individual computer programs in the population of 4,000 were still very poor.

By generation 10, the fitness of the best-of-generation individual had improved to 40.02. This individual had 111 points and is shown below:

```
(IFLTE (EXP (IFLTE T T T 2.482)) (EXP (+ (% 9.39 T) (IFLTE
(IFLTE T 9.573 T -6.085) (% T 0.217001) (EXP -6.925) (% T
T)))) (EXP (* (- (+ T -4.679) (EXP T)) (IFLTE (% -5.631 T)
(% -1.675 -1.485) (+ T 2.623) (EXP T)))) (% (EXP (- T T)) (-
(+ (* -1.15399 -5.332) (% (% (* (IFLTE -8.019 T 0.338 T) (%
T 8.571)) (- (* (- 1.213 T) (+ (EXP T) (+ 7.605 6.873)))
(IFLTE (+ T T) (* -5.749 T) (+ T T) (- T T)))) (* T 6.193)))
(IFLTE (% (EXP T) (EXP (* -3.817 T))) (* T 6.193) (- -8.022
7.743) (+ T -9.464))))).
```

Note the subexpression (- -8.022 7.743) in the last line which evaluates to -15.765. The needed value -15.765 was evolved, via subtraction, from the ephemeral random constants -8.022 and 7.743 over the generations.

Figure 8 compares (a) the genetically produced best-of-generation individual impulse response function from generation 10 and (b) the correct impulse response. As can be seen, this individual bears some resemblance to the correct impulse response for the system.

By generation 20, the best-of-generation individual had 188 points and a fitness value of 19.85. By generation 30, the best-of-generation individual had 236 points and a fitness value of 12.37. By generation 40, the best-of-generation individual had 294 points and a fitness value of 6.97.

By generation 50, the best-of-generation individual shown below had 286 points and a fitness value of 5.22 (i.e., an average squared error of only about 0.87 for each of the 60 fitness cases):

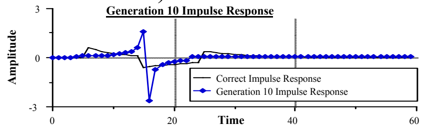

**Figure 8 Generation 10 comparison of impulse response functions**

```
(IFLTE (EXP (IFLTE T T T 2.482)) (EXP (- -8.022 7.743)) (EXP
(* (- (% (% (* (IFLTE -8.019 T 0.338 T) (- -5.392 T)) T) (%
(* (EXP (EXP -5.221)) (IFLTE (* T T) (IFLTE (% -5.631 T) (%
-1.675 -1.485) (+ T 2.623) (EXP T)) (- 9.957 -4.115) (% -
8.978 T))) (- (IFLTE (EXP T) T 1.1 (EXP 2.731)) (% (* (* -
3.817 T) (% T 8.571)) (IFLTE -8.019 T 0.338 T))))) (EXP
(IFLTE T 9.573 T -6.085))) (IFLTE (% -5.631 T) (% -1.675 -
1.485) (+ T 2.623) (EXP T)))) (% (EXP (IFLTE -8.019 T 0.338
T)) (- (+ (* -1.15399 -5.332) (% (% (* (IFLTE -8.019 T 0.338
T) 8.571) (% T 8.571)) (- (+ (* -1.15399 -5.332) (% (% (*
(IFLTE -8.019 T 0.338 T) (% T 8.571)) T) (% (* (EXP (EXP -
5.221)) (IFLTE (* T T) (IFLTE (EXP (- -8.022 7.743)) (% -
1.675 -1.485) (+ T 2.623) (EXP T)) (- 9.957 -4.115) (% -
8.978 T))) (- (IFLTE (EXP T) T 1.1 (EXP 2.731)) (% (- -8.022
7.743) (EXP 2.731)))))) (IFLTE (% (EXP T) (- 9.957 -4.115))
(* T 6.193) (IFLTE (% -5.631 T) (% -1.675 -1.485) (+ T
2.623) (EXP T)) (+ T -9.464))))) (IFLTE (% (EXP T) (- -8.022
7.743)) (* T 6.193) (IFLTE (% (EXP T) (EXP (* -3.817 T))) (-
(IFLTE (+ T -4.679) (- -5.392 T) 1.1 (EXP 2.731)) (% (+ T T)
(* -1.15399 -5.332))) (- -8.022 (% -8.022 (- (* (- (* (-
1.213 T) 0.217001) (% -5.631 T)) (+ (EXP T) (- (IFLTE (+ T -
4.679) (- -5.392 T) 1.1 (EXP 2.731)) (EXP (% T 0.217001)))))
(IFLTE (+ T T) T (* T 6.193) (- T T))))) (+ T -9.464)) (+ T
-9.464))))).
```

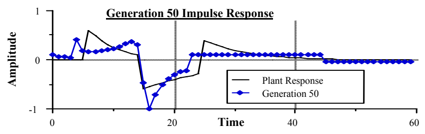

**Figure 9 Generation 50 comparison of impulse response functions**

Figure 9 compares (a) the genetically produced best-of-generation individual impulse response function from generation 50 and (b) the correct impulse response.

The above impulse response function is not the exact impulse response function *H(t)* for this system. However, this genetically created impulse response function is very good. It is an approximately correct computer program that emerged from a Darwinian process that searches the space of possible functions for a satisfactory result.

Since the fitness measure (which is the driving force of genetic programming) is based on the response of the system to the square input, the performance of the genetically produced best-of-generation impulse response functions from generations 0, 10, and 50 can be appreciated examining the response of the system to the square input (shown in figure 5).

Figure 10 compares (a) the plant response (which is the same in all three panels of this figure) to the square input and (b) the response to the square input using the best-of-generation individual impulse response functions from generations 0, 10, and 50. As can be seen in the first and second panels of this figure, the performance of the best-of-generation individuals from generations 0 and 10 were not very good, although generation 10 was considerably better than generation 0. However, as can be seen in the third panel of this figure, the performance of the best-of-generation individual from generation 50 is close to the plant response (the total squared error being only 5.22 over the 60 fitness cases).

If we define a hit to be a fitness case (out of the 60) for which the response to the square input of the genetically produced individual comes within 0.5 of the plant response, then the number of hits improved from 17 for the best-of-generation individual for generation 0, to 29 for generation 10, and to 54 for generation 50.

The ability of the genetically discovered impulse response function to generalize can be demonstrated by considering four additional forcing functions to the system – a ramp input, a unit step input, a shorter unit square input, and a noise signal. Note that since we are operating in discrete time, there is no generalization of the system in the time domain.

Figure 11 shows (a) the plant response to a particular unit ramp input (whose amplitude is 0 between times 0 and 3, whose amplitude ramps up from 0 to 1 between times 3 and 23, and whose amplitude is 1 between times 23 and 59) and (b) the response to this ramp input using the best-of-generation individual from generation 50. As can be seen, the performance of the best-of-generation individual from generation 50 is close to the plant response for this ramp input (the total squared error being only 7.2 over the 60 fitness cases).

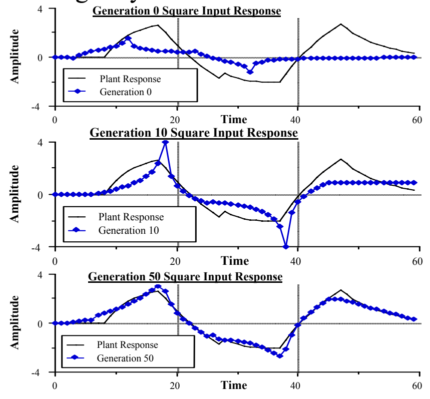

**Figure 10 Response of generations 0, 10, and 50 to square input**

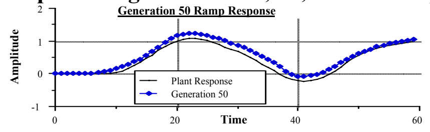

**Figure 11 Response of generations 0, 10, and 50 to ramp input**

Figure 12 compares (a) the plant response to a particular unit step input (where the amplitude of the input steps up from 0 to 1 at time 3) and (b) the response to this step input using the best-of-generation individual from generation 50. As can be seen, the performance of the best-of-generation individual from generation 50 is also close to the plant response for this unit step input (the total squared error being only 12.9 over the 60 fitness cases).

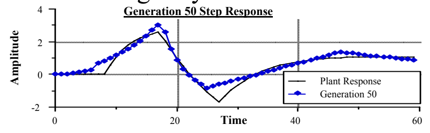

**Figure 12 Response of generation 50 to step input**

Figure 13 compares (a) the plant response to a particular short unit square input (whose amplitude steps up from 0 to 1 at time 15 and steps down at time 23) and (b) the response to this short square input using the best-of-generation individual from generation 50. As can be seen, the performance of the best-of-generation individual from generation 50 is also close to the plant response for this short unit square (the total squared error being only 17.1 over the 60 fitness cases).

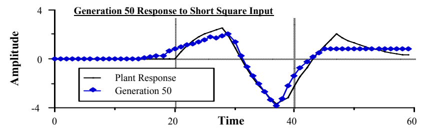

**Figure 13 Response of generation 50 to short square input**

Figure 14 shows a noise signal with amplitude in the range [0,1] which we will use as the forcing function for our fourth and final test.

Figure 15 compares (a) the plant response to this noise signal and (b) the response to this noise signal using the best-of-generation individual from generation 50. As can be seen, the performance of the best-of-generation individual from generation 50 is also close to the plant response for this noise signal (the total squared error being only 18.3 over the 60 fitness cases).

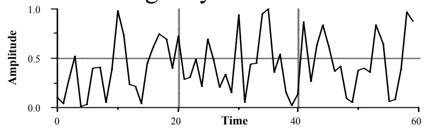

**Figure 14 Noise signal**

Genetic programming is well suited to difficult control problems where no exact solution is known and where an exact solution is not required. The solution to a problem produced by genetic programming is not just a numerical solution applicable to a single specific combination of numerical input(s), but, instead, comes in the form of a general function (i.e., a computer program) that maps the input(s) of the problem into the output(s). There is no need to specify the exact size and shape of the computer program in advance. The needed structure is evolved in response to the selective pressures of Darwinian natural selection and sexual recombination.

Note also that we did not pre-specify the size and shape of the result. We did not specify that the result obtained in generation 50 would have 286 points. As we proceeded from generation to generation, the size and shape of the best-of-generation individuals changed as a result of the selective pressure exerted by the fitness measure and the genetic operations.

## 7 RESULTS OVER A SERIES OF RUNS

Figure 16 presents two curves, called the performance curves, relating to the problem of finding the impulse response function over a series of runs. The curves are based on 18 runs with a population size *M* of 4,000 and a maximum number of generations to be run *G* of 51.

The rising curve in figure 16 shows, by generation, the experimentally observed cumulative probability of success, *P*(*M*,*i*), of solving the problem by generation *i* (i.e., finding at least one S-expression in the population which has a value of fitness of 20.0 or less over the 60 fitness cases). As can be seen, the experimentally observed value of the cumulative probability of success, *P*(*M*,*i*), is 11% by generation 25 and 61% by generation 46 over the 18 runs.

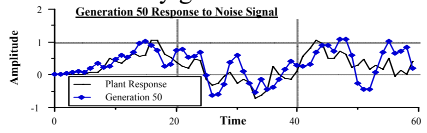

**Figure 15 Response of generation 50 to noise signal**

The second curve in figure 16 shows, by generation, the number of individuals that must be processed, *I*(*M*,*i*,*z*), to yield, with probability *z*, a solution to the problem by generation *i*. *I*(*M*,*i*,*z*) is derived from the experimentally observed values of *P*(*M*,*i*). Specifically, *I*(*M*,*i*,*z*) is the product of the population size *M*, the generation number *i*, and the number of independent runs *R(z)* necessary to yield a solution to the problem with probability *z* by generation *i*. In turn, the number of runs *R(z)* is given by

$$R(z) = \left\lceil \frac{log(1-z)}{log(1-P(M,i))} \right\rceil ,$$

where the square brackets indicates the ceiling function for rounding up to the next highest integer. Throughout this paper, the probability *z* will be 99%.

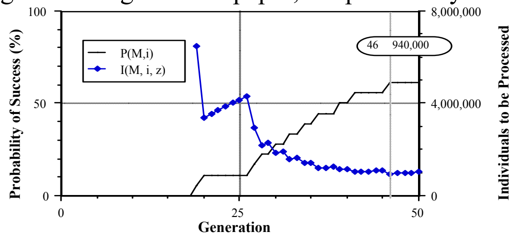

**Figure 16 Performance curves showing that it is sufficient to process 940,000 individuals to yield a solution with 99% probability.**

As can be seen, the $I(M,i,z)$ curve reaches a minimum value at generation 46 (highlighted by the light dotted vertical line). For a value of $P(M,i)$ of 61%, the number of independent runs $R(z)$ necessary to yield a solution to the problem with a 99% probability by generation $i$ is 5. The two summary numbers (i.e., 46 and 940,000) in the oval indicate that if this problem is run through to generation 46 (the initial random generation being counted as generation 0), processing a total of 940,000 individuals (i.e., 4,000 ∞ 47 generations ∞ 5 runs) is sufficient to yield a solution to this problem with 99% probability. This number, 940,000, is a measure of the computational effort necessary to yield a solution to this problem with 99% probability.

## 8 CONCLUSIONS

We demonstrated the use of the genetic programming paradigm to genetically breed a good approximation to an impulse response function for a time-invariant linear system. Genetic programming produced an impulse response function, in symbolic form, using only the observed response of the unknown system to a unit square input.

## ACKNOWLEDGEMENTS

Simon Handley of the Computer Science Department at Stanford University made helpful comments on this paper.

## REFERENCES

Belew, Richard and Booker, Lashon (editors) *Proceedings of the Fourth International Conference on Genetic Algorithms*. San Mateo, CA: Morgan Kaufmann Publishers Inc. 1991.

Davis, Lawrence (editor) *Genetic Algorithms and Simulated Annealing.* London: Pittman l987.

Davis, Lawrence. *Handbook of Genetic Algorithms*. New York: Van Nostrand Reinhold 1991.

Goldberg, David E. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Reading, MA: Addison-Wesley l989.

Holland, John H. *Adaptation in Natural and Artificial Systems*. Ann Arbor, MI: University of Michigan Press 1975. Revised Second Edition 1992 from The MIT Press.

Koza, John R. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* Cambridge, MA: The MIT Press 1992. 1992a.

Koza, John R. A genetic approach to finding a controller to back up a tractor-trailer truck. In *Proceedings of the 1992 American Control Conference*. Evanston, IL: American Automatic Control Council 1992. Volume III. Pages 2307-2311. 1992b.

Koza, John R. Evolution of subsumption using genetic programming. In Varela, Francisco J., and Bourgine, Paul (editors). *Toward a Practice of Autonomous Systems: Proceedings of the first European Conference on Artificial Life*. Cambridge, MA: The MIT Press 1992. Pages 110-119. 1992c.

Koza, John R. A genetic approach to econometric modeling. In Bourgine, Paul and Walliser, Bernard. *Economics and Cognitive Science*. Oxford: Pergamon Press 1991. Pages 57-75. 1991d.

Koza, John R. The genetic programming paradigm: Genetically breeding populations of computer programs to solve problems. In Soucek, Branko and the IRIS Group (editors). *Dynamic, Genetic, and Chaotic Programming*. New York: John Wiley 1992. 1992e.

Koza, John R. and Keane, Martin A. Genetic breeding of non-linear optimal control strategies for broom balancing. In *Proceedings of the Ninth International Conference on Analysis and Optimization of Systems. Antibes,France, June, 1990.* Pages 47-56. Berlin: Springer-Verlag 1990. 1990.

Koza, John R. and Rice, James P. *Genetic Programming: The Movie.* Cambridge, MA: The MIT Press 1992.

Michalewicz, Zbignlew. *Genetic Algorithms + Data Structures = Evolution Programs*. Springer-Verlag 1992.

Whitley, Darrell (editor). *Proceedings of Workshop on the Foundations of Genetic Algorithms and Classifier Systems*, *Vail, Colorado 1992.* San Mateo, CA: Morgan Kaufmann Publishers Inc. 1992.