Finding an Impulse Response Function Using Genetic Programming

Martin A. Keane
Third Millennium Venture Capital Limited
5733 Grover
Chicago, Illinois 60630
312-777-1524

John R. Koza
Computer Science Department
Margaret Jacks Hall
Stanford University
Stanford, California 94305
Koza@Cs.Stanford.Edu
415-941-0336

James P. Rice
Stanford University Knowledge Systems Laboratory
701 Welch Road
Palo Alto, California 94304
Rice@Cs.Stanford.Edu
415-723-8405

Abstract

For many practical problems of control engineering, it is desirable to find a function, such as the impulse response function or transfer function, for a system for which one does not have an analytical model. The finding of the function, in symbolic form, that satisfies the requirements of the problem (rather than merely finding a single point) is usually not possible when one does not have an analytical model of the system. This paper illustrates how the recently developed genetic programming paradigm, can be used to find an approximation to the impulse response, in symbolic form, for a linear time-invariant system using only the observed response of the system to a particular known forcing function. The method illustrated can then be applied to other problems in control engineering that require the finding of a function in symbolic form.

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 impulse response function for a linear time-invariant system. We thereby also illustrate the process by which genetic programming can be used to solve other problems in control engineering requiring the finding of a function satisfying certain requirements.

2. Background on Genetic Methods

Since its invention in the 1970s by John Holland, the genetic algorithm [1] has proven successful for finding an optimal point in a search space for a wide variety of problems [2, 3, 4, 5, 6, 7].

However, for many problems the most natural representation for the solution to the problem is an entire computer program or function (i.e., a composition of primitive functions and terminals), not merely a single point in the search space of the problem. The size, shape, and contents of the function needed to solve the problem is generally not known in advance and therefore should be discovered as part of the problem-solving process, not specified in advance.

The book *Genetic Programming: On the Programming of Computers by Means of Natural Selection* [8] describes the recently developed genetic programming paradigm and demonstrates that populations of computer programs (i.e., functions) can be genetically bred to solve problems in a surprising variety of different areas. A videotape visualization of 22 applications of genetic programming can be found in the *Genetic Programming: The Movie* [9]. Specifically, genetic programming has been successfully applied to problems such as

- discovering control strategies for backing up a tractor-trailer truck [12],
- 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) [10, 11],
- discovering inverse kinematic equations to control the movement of a robot arm to a designated target point [8],
- evolution of a subsumption architecture for robotic control [13],
- 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) [8], 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) [8, 9, 14].

Genetic Programming

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. 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.

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.

Steps Required to Execute the Genetic Programming

Genetic programming is a domain independent method that genetically breeds a population 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. See [8] for details.

(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 (see below).

(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.

Crossover Operation

The crossover (recombination) operation for the genetic programming paradigm is a sexual operation that operates on two parental S-expressions and produces two offspring S-expressions using parts of each parent. Typically the two parents are hierarchical compositions of functions of different size and shape. In particular, the crossover operation starts by selecting a random crossover point in each parent and then creates two new offspring expressions by exchanging the sub-expressions (i.e. sub-trees) between the two parents. Because entire sub-expressions are swapped, this genetic crossover operation produces syntactically and semantically valid S-expressions as offspring.

For example, consider a first parental expression

```
(OR (NOT D1) (AND D0 D1))
```

and a second parental expression

```
(OR (OR D1 (NOT D0))
    (AND (NOT D0) (NOT D1)))
```

These two expressions can be depicted graphically, if desired, as rooted, point-labeled trees with ordered branches. Assume that the points of both expressions are numbered in a depth-first way starting at the left. Suppose that the second point (out of 6 points of the first parent) is randomly selected as the crossover point for the first parent and that the sixth point (out of 10 points of the second parent) is randomly selected as the crossover point of the second parent. The crossover points are therefore the `NOT` in the first parent and the `AND` in the second parent. The two crossover fragments are two bold, underlined sub-expressions in the two parents.

The first offspring resulting from the crossover operation is

```
(OR (AND (NOT D0) (NOT D1)) (AND D0 D1)),
```

and the second offspring is

```
(OR (OR D1 (NOT D0)) (NOT D1)).
```

Details are in [8].

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.

3. Finding an Impulse Response Function

Figure 1 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, we use the discrete-time version of this system in this paper.

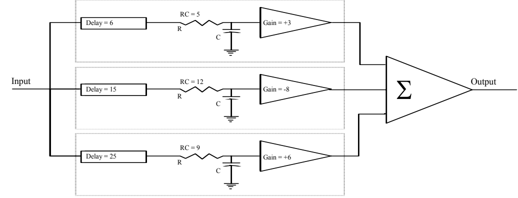

Figure 1 Linear time-invariant system

Figure 2 shows (a) a 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 from figure 1 when this particular square input is used as a forcing function.

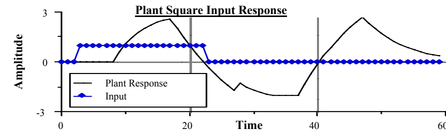

Figure 2 Plant response when a square input is the forcing function

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 shown in figure 1 is known to be

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

Figure 3 shows the impulse response function *H(t)* for this system.

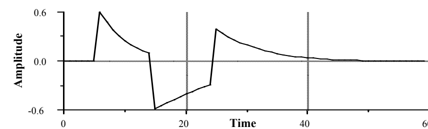

Figure 3 Impulse response function

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. Note that although we know this correct impulse response function, genetic programming does not have access to it.

4. 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

$$4.567 + 0.234T$$

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.

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 and the other three arithmetic functions (+, -, *) ordinarily return their usual result; however, if any such result 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 its third argument if its first argument is less than or equal to its second argument and otherwise evaluates 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 is measured in terms of error. The smaller the error, the better. An error of exactly zero would occur if the particular function was the exact impulse response of the system.

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 2 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 absolute value 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

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

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. 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 [8].

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

5. Results

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 (i.e., generation 0) to be very good. However, some individuals are better than others. In generation 0, the fitness of the worst individual impulse response function among the 4,000 individuals in the population was very poor (i.e., fitness was the enormous value of 6 x 10^{39}). The fitness of the worst 30% of the population was 10^{10} (or worse). The fitness of the median individual was 59,585. The fitness of the best individual impulse response function was 62.18 (i.e., an average error of about 1.04 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 4 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 3. As can be seen, there is little resemblance between this best-of-generation individual and the correct impulse response.

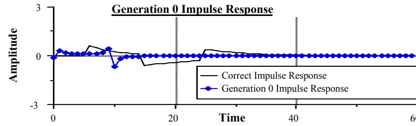

Figure 4 Comparison of best genetically produced individual from generation 0 with the correct impulse response function

Each successive generation of the population of size 4,000 is then created from the preceding generation by applying the fitness proportionate reproduction operation to 10% of the population and by applying the crossover operation to 90% of the population (with both parents selected using tournament selection with a group size of 7, as described in [8]).

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 59.06 for generation 3, 56.06 for generation 4, 55.50 for generation 5, 52.63 for generation 7, and 41.64 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 37.08. 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))))).
```

Figure 5 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 than was the case in figure 4.

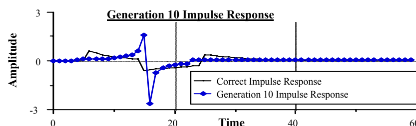

Figure 5 Generation 10 comparison of impulse response functions

By generation 20, the best-of-generation individual had 188 points and a fitness value of 25.68. By generation 30, the best-of-generation individual had 236 points and a fitness value of 20.40. By generation 40, the best-of-generation individual had 294 points and a fitness value of 14.89.

By generation 50, the best-of-generation individual shown below had 286 points and a fitness value of 12.58:

```
(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 6 compares (a) the genetically produced best-of-generation individual impulse response function from generation 10 and (b) the correct impulse response.

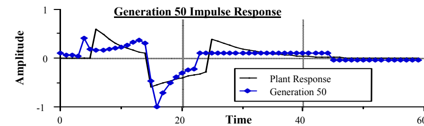

Figure 6 Generation 50 comparison of impulse response functions

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.

The performance of the genetically produced best-of-generation impulse response functions from generations 0, 10, and 50 can be seen by examining the response of the system to the square input (shown in figure 2).

Figure 7 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 error is 12.58 over the 60 fitness cases (i.e., an average error of only about 0.21 per fitness case).The performance of the genetically discovered impulse response function can be further 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. As we will see, the evolved program generalizes well (i.e., there was no overfitting). Note that since we are operating in discrete time, there is no generalization of the system in the time domain.

Figure 8 shows (a) the plant response (which is the same in all three panels of this figure) 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 generations 0, 10, and 50. The first and second panels of this figure show the progressive improvement between generations 0 and 10 of the run. 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 for this ramp input (i.e., a total error of only 7.2 over the 60 fitness cases).

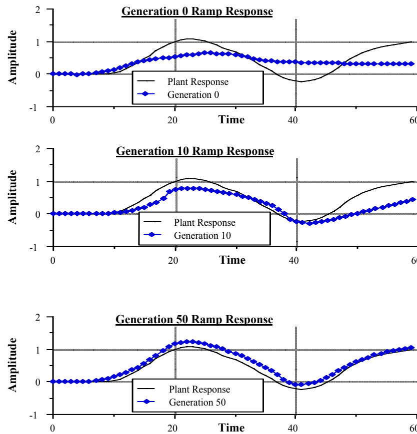

Figure 8 Response of generations 0, 10, and 50 to ramp input

Figure 9 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 (i.e., a total error of only 12.9 over the 60 fitness cases).

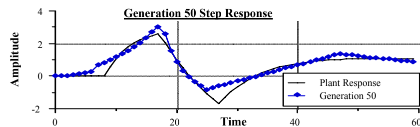

Figure 9 Response of generation 50 to step input

Figure 10 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 (i.e., a total error of only 17.1 over the 60 fitness cases).

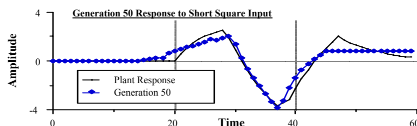

Figure 10 Response of generation 50 to short square input

Figure 11 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.

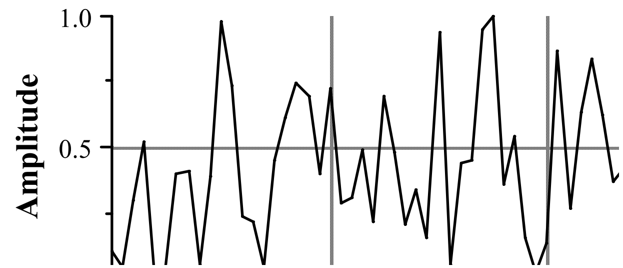

Figure 11 Noise signal

Figure 12 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 (i.e., a total error of only 18.3 over the 60 fitness cases).

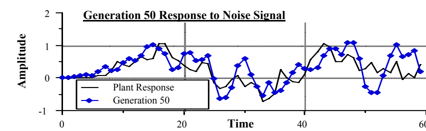

Figure 12 Response of generation 50 to noise signal

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.

We have achieved similar results in other runs of this problem.

6. 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. This specific demonstration of genetic programming suggests how other problems in control engineering requiring the finding of a function, in symbolic form, might be solved using genetic programming.

7. Acknowledgements

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

8. References

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

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

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

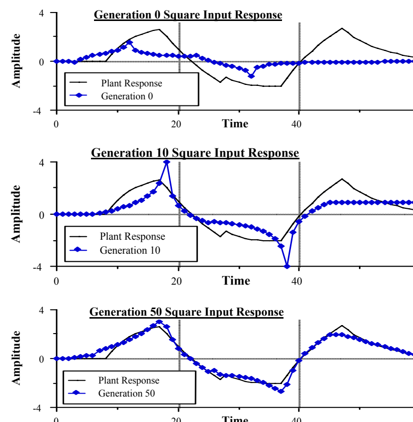

Figure 7 Response of generations 0, 10, and 50 to square input

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

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

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

[7] 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.

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

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

[10] Koza, John R., and Keane, Martin A. Cart centering and broom balancing by genetically breeding populations of control strategy programs. In *Proceedings of International Joint Conference on Neural Networks*, *Washington, January 15-19, 1990*. Volume I, Pages 198-201. Hillsdale, NJ: Lawrence Erlbaum 1990.

[11] 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.

[12] 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.

[13] 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.

[14] 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.