Automatic Creation of both the Topology and Parameters for a Robust Controller by Means of Genetic Programming

John R. Koza
Section on Medical Informatics
School of Medicine
Stanford University
Stanford, California 94305
koza@stanford.edu
http://www.smi.stanford.edu/people/koza

Martin A. Keane
Econometrics Inc.
111 E. Wacker Dr.
Chicago, Illinois 60601
makeane@ix.netcom.com

Forrest H Bennett III
Genetic Programming Inc.
Box 1669
Los Altos, California 94023
forrest@evolute.com
http://www.genetic-programming.com

Jessen Yu
Genetic Programming Inc.
Box 1669
Los Altos, California 94023
jyu@cs.stanford.edu

William Mydlowec
Genetic Programming Inc.
Box 1669
Los Altos, California 94023
myd@cs.stanford.edu

Oscar Stiffelman
Computer Science Department
Stanford University
Stanford, California 94305
ozzie@cs.stanford.edu

Abstract

The paper describes a general automated method for synthesizing the design of both the topology and parameter values for controllers. The automated method automatically makes decisions concerning the total number of processing blocks to be employed in the controller, the type of each block, the topological interconnections between the blocks, the values of all parameters for the blocks, and the existence, if any, of internal feedback between the blocks of the overall controller. Incorporation of time-domain, frequency-domain, and other constraints on the control or state variables (often analytically intractable using conventional methods) can be readily accommodated.

The automatic method described in the paper (genetic programming) is applied to a problem of synthesizing the design of a robust controller for a plant with a second-order lag. A textbook PID compensator preceded by a lowpass pre-filter delivers credible performance on this problem. However, the automatically created controller employs a second derivative processing block (in addition to proportional, integrative, and derivative blocks and a pre-filter). It is better than twice as effective as the textbook controller as measured by the integral of the time-weighted absolute error, has only two-thirds of the rise time in response to the reference (command) input, and is 10 times better in terms of suppressing the effects of disturbance at the plant input.

1 Introduction

The process of creating (synthesizing) the design of a controller entails making decisions concerning the total number of processing blocks to be employed in the controller, the type of each block (e.g., lead, lag, gain, integrator, differentiator, adder, inverter, subtractor, and multiplier), the interconnections between the blocks, the values of all parameters for the blocks, and the existence, if any, of internal feedback between the processing blocks. This process is typically channeled along lines established by existing mathematical techniques. For example, existing techniques often lead to a PID-type controller consisting of one proportional, one integrative, and one derivative processing block.

It would be desirable to have an automatic system for creating the design of a controller that did not require the user to prespecify the topology of the controller (e.g. PID), but, instead, automatically produced both the overall topology and parameter values directly from a high-level statement of the requirements of the controller.

This paper describes how genetic programming can be used to automatically create both the topology and parameter values for a controller. The automatically created controllers can accommodate one or more externally supplied reference (command) signals, external feedback of one or more plant outputs to the controller, computations of error between the reference signals and the corresponding external plant outputs, one or more internal state variables of the plant, and one or more control variables. These automatically created controllers can also accommodate internal feedback of one or more signals from one part of the controller to another part of the controller. The automatically created controllers can be composed of processing elements such as gain blocks, lead blocks, lag blocks, inverter blocks, differential input integrators,

differentiators, adders and subtractors and multipliers of time-domain signals, and adders and subtractors and multipliers of numerical values. These controllers can also contain conditional operators that operate on time-domain signals.

In addition, the design process described in this paper for automatically creating controllers can readily accommodate time-domain, frequency-domain, and other constraints (often analytically intractable using conventional methods) on the control or state variables.

Section 2 describes an illustrative control problem involving the design of a robust controller for a plant with a second-order lag. Section 3 provides general background on genetic programming. Section 4 describes how genetic programming is applied to control problems. Section 5 describes the preparatory steps necessary to apply genetic programming to the illustrative control problem. Section 6 presents the results.

2 Statement of the Problem

The technique for automatically synthesizing a controller will be illustrated by a problem calling for the design of a robust controller for a plant with a second-order lag. The problem (Dorf and Bishop 1998, page 707) is to create both the topology and parameter values for a controller for a plant whose transfer function is

$$G(s) = \frac{K}{(1+\tau s)^2},$$

where the internal gain K = 1 and 2 and where the time-constant τ = 0.5 and 1.0, such that the plant output reaches the level of the reference signal in minimal time and such that the overshoot in the response to a step input is less than 2%.

A textbook controller consisting of a PID compensator and a lowpass pre-filter is presented in Dorf and Bishop 1998. It delivers credible performance on this problem. We added two additional constraints of the type that are often implicit in controller design. These added constraints are, in fact, satisfied by the controller presented in Dorf and Bishop 1998. The first constraint is that the input to the plant is limited to the range between -40 and +40 volts. The second constraint is that the closed loop frequency response of the system must lie below a 40 dB per decade lowpass curve whose corner is at 100 Hz.

The problem involves one reference (command) signal, denoted by $R(s)$, in figure 1. The plant $G(s)$ has one input and one output $Y(s)$. The reference signal $R(s)$ is fed through pre-filter $G_p(s)$. The plant output $Y(s)$ is passed through $H(s)$ and then subtracted, in continuous time, from the pre-filtered reference signal and the difference (error) is fed into the compensator $G_c(s)$. $G_c(s)$ has one input and one output $U(s)$. Disturbance $D(s)$ may be added to the output $U(s)$ of $G_c(s)$ and the sum is subject to the limitation that it be in the range between -40 and +40 volts.

3 Background on Genetic Programming

Genetic programming is an automatic technique for generating computer programs to solve, or approximately solve, problems. In particular, genetic programming is capable of automatically creating the design of complex structures. Genetic programming approaches a program synthesis problem or a design problem in terms of "what needs to be done" — as opposed to "how to do it".

Genetic programming (Koza 1992; Koza and Rice 1992) is an extension of the genetic algorithm (Holland 1975). Genetic programming is capable (Koza 1994a, 1994b) of evolving reusable, parametrized, hierarchically-called automatically defined functions (subroutines).

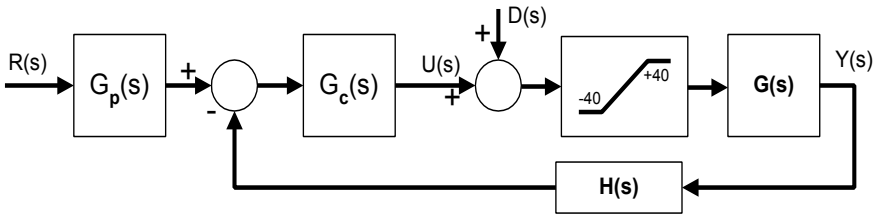

Figure 1 Overall model.

Architecture-altering operations (Koza, Bennett, Andre, and Keane 1999; Koza, Bennett, Andre, Keane, and Brave 1999) enable genetic programming to automatically determine the number of automatically defined functions, the number of arguments that each possesses, and the nature of the

hierarchical references, if any, among such automatically defined functions. Architecture-altering operations also enable genetic programming to automatically determine whether and how to use internal memory, iterations, and recursion.

Genetic programming breeds computer programs to solve problems by executing the following three steps:

(1) Randomly create an initial population of individual computer programs.
(2) Iteratively perform the following substeps (called a *generation*) on the population of programs until the termination criterion has been satisfied:
  (a) Assign a fitness value to each individual program in the population using the fitness measure.
  (b) Create a new population of individual programs by applying the following three genetic operations. The genetic operations are applied to one or two individuals in the population selected with a probability based on fitness (with reselection allowed).
    (i) Reproduce a selected individual by copying it into the new population.
    (ii) Create two new individual programs from two selected parental individuals by genetically recombining subtrees from each parental program using the crossover operation at randomly chosen crossover points in the parental programs.
    (iii) Create a new individual from a selected parental individual by randomly mutating one randomly chosen subtree of the parental program.
    (iv) *Architecture-altering operations*: Choose an architecture-altering operation from the repertoire of such operations available for the run (if any) and create one new offspring program for the new population by applying the architecture-altering operation to the one selected program.
(3) Designate the individual computer program that is identified by the method of result designation (e.g., the *best-so-far* individual) as the result of the run of genetic programming. This result may represent a solution (or an approximate solution) to the problem.

Genetic Programming: Darwinian Invention and Problem Solving (Koza, Bennett, Andre, and Keane 1999) and the accompanying videotape (Koza, Bennett, Andre, Keane, and Brave 1999) demonstrate that genetic programming is capable of synthesizing the design of both the topology and sizing for a wide variety of analog electrical circuits from a high-level statement of the circuit's desired behavior and characteristics. Nine of the evolved analog circuits in that book were previously patented.

Genetic programming often creates novel designs because it is a probabilistic process that is not encumbered by the preconceptions that often channel human thinking down familiar paths.

Additional information on current research in genetic programming can be found in Banzhaf, Nordin, Keller, and Francone 1998; Langdon 1998; Kinnear 1994; Angeline and Kinnear 1996; Spector, Langdon, O'Reilly, and Angeline 1999; Koza, Goldberg, Fogel, and Riolo 1996; Koza, Deb, Dorigo, Fogel, Garzon, Iba, and Riolo 1997; Koza, Banzhaf, Chellapilla, Deb, Dorigo, Fogel, Garzon, Goldberg, Iba, and Riolo 1998; Banzhaf, Poli, Schoenauer, and Fogarty 1998; and in the new journal *Genetic Programming and Evolvable Machines* (from Kluwer Academic Publishers starting in January 2000).

4 Genetic Programming and Control

Genetic programming has been previously applied to certain simple control problems, including discrete-time problems where the evolved program receives the system's current state as input and computes a value for a control variable (Koza and Keane 1990a, 1990b; Koza 1992; Koza, Bennett, Andre, and Keane 1999, chapter 13). In addition, genetic programming has been previously used to evolve an analog electrical circuit for a discrete-time robotic controller (Koza, Bennett, Andre, and Keane 1999, chapter 48).

The output of a controller causes the actual response(s) of a process (called the *plant*) to match desired response(s), called the *reference signal(s)*.

In a closed loop controller, the difference(s) between the actual plant response(s) and the reference signal(s) are computed and fed back into the controller. Controllers are often represented by directed graphs in which the internal points represent certain functions, in which external points represent the controller's input(s), and in which cycles correspond to internal feedback within the controller.

Genetic programming can be extended to the problem of creating both the topology and parameter values for a closed loop controller by establishing a mapping between the program trees used in genetic programming and the specialized type of directed graphs with cycles germane to controllers.

Various functions and terminals may be included in the repertoire from which controllers can be

constructed. The functions correspond to blocks in a block diagram representing a controller. Some terminals represent time-domain signals while others represent numerically valued constants or variables. Some functions operate on continuous time-domain signals while others operate on numerically valued constants or variables.

The number of result-producing branches in the to-be-evolved controller equals the number of control variables that are to be passed from the controller to the plant. Each result-producing branch is a composition of the functions and terminals from a repertoire (below) of functions and terminals appropriate for control problems.

Programs trees in the population during the initial random generation (generation 0) consist only of result-producing branch(es). Automatically defined functions are introduced incrementally (and sparingly) into the population on subsequent generations by means of the architecture-altering operations. Each automatically defined function is a composition of the functions and terminals appropriate for control problems, all existing automatically defined functions, and (possibly) dummy variables (formal parameters) that permit parameterization of the automatically defined function. In control problems, automatically defined functions provide a mechanism for internal feedback within the to-be-evolved controller.

Each branch of each program tree in the initial random population is created in accordance with a constrained syntactic structure. Each genetic operation (crossover, mutation, reproduction, and architecture-altering operation) produces offspring that comply with the constrained syntactic structure.

4.1 Repertoire of Functions

The functions may include the following:

The two-argument GAIN function multiplies the time-domain signal represented by its first argument by a constant numerical factor represented by its second argument.

The one-argument INVERTER function negates the time-domain signal represented by its argument.

The DIFFERENTIAL_INPUT_INTEGRATOR function has two arguments and integrates the time-domain signal representing the difference between its two arguments.

The two-argument LAG function applies the transfer function $1 / (1 + \tau s)$, where s is the Laplace operator and τ is a constant parameter. The first argument is the time-domain input signal. The second argument, τ, is a numerically valued expression that is interpreted (in the same manner as the constants $\Re$) as a floating-point number in seconds.

The three-argument LAG2 function applies the transfer function $\omega_0 / (s^2 + 2\zeta\omega_0 + \omega_0^2)$, where s is the Laplace operator, ζ is the damping ratio, and ω_0 is the corner frequency.

The two-argument LEAD function applies the transfer function $1 + \tau s$, where s is the Laplace operator and τ is a constant parameter. The first argument is the time-domain input signal. The second argument, τ, is a numerically valued expression that is interpreted (in the same manner as the constants $\Re$) as a floating-point number in seconds.

The one-argument DIFFERENTIATOR function differentiates the time-domain signal represented by its argument.

The two-argument ADD_SIGNAL and SUB_SIGNAL functions perform addition or subtraction, respectively, on the time-domain signals represented by their two arguments.

The three-argument ADD_3_SIGNAL adds the time-domain signals represented by its three arguments.

The two-argument ADD_NUMERIC and SUB_NUMERIC functions perform addition or subtraction, respectively, on the constant numerical values represented by their two inputs.

The three-argument IFC function (not used in this paper) operates on three time-domain signals and produces a time-domain signal. If, for a given time, the value of the time-domain function in the first argument of the IFC function is positive, the value of the IFC function is the value of the time-domain function in the second argument of the IFC function. If, for a given time, the value of the time-domain function in the first argument of the IFC function is negative, the value of the IFC function is the value of the time-domain function in the third argument of the IFC function. If, for a given time, the value of the time-domain function in the first argument of the IFC function is exactly zero, the value of the IFC function is the average of the value of the time-domain functions in the second and third arguments of the IFC function.

4.2 Repertoire of Terminals

The terminals may include the following:

The REFERENCE_SIGNAL is the time-domain signal representing the reference signal (desired plant response).

The PLANT_OUTPUT is the plant output.

The CONTROLLER_OUTPUT is the time-domain signal representing the output of the controller. Note that this signal can be used, if desired to provide feedback of the controller's output directly back into the controller.

The ERROR terminal is the difference between the time-domain signal representing the reference signal

(desired plant response) and the time-domain signal representing the actual plant response.

If the plant has internal state(s) that are available to the controller, then STATE_0, STATE_1 , etc. are the plant's internal state(s).

The CONSTANT_0_SIGNAL function is a time-domain signal that is always zero.

Numerical constant terminals, ℜ, are floating-point constants in the range -5.0 to +5.0. Numerical constant terminals may be combined in expressions by means of arithmetic functions (such as addition and subtraction). A three-step process is used to interpreting numerical expressions appearing in programs. First, the numerical expression is evaluated. We call the floating-point number that it returns, *X*. Second, *X* is used to produce an intermediate value *U* in the range of –5 to +5 in the following way: If the return value *X* is between –5.0 and +5.0, an intermediate value *U* is set to the value *X* returned by the subtree. If the return value *X* is less than –100 or greater than +100, *U* is set to a saturating value of zero. If the return value *X* is between –100 and –5.0, *U* is found from the straight line connecting the points (–100, 0) and (–5, -5). If the return value *X* is between +5.0 and +100, *U* is found from the straight line connecting (5, 5) and (100, 0). Third, the actual value is the antilogarithm (base 10) of the intermediate value *U* (i.e., 10^{U}).

5 Preparatory Steps

Before applying genetic programming to a problem, six major preparatory steps are required: (1) identify the terminals for the program trees, (2) identify the functions for the program trees, (3) define the fitness measure, (4) choose control parameters for the run, (5) determine the termination criterion and method of result designation, and (6) determine the architecture of the program trees.

5.1 Program Architecture

Since there is one result-producing branch in the program tree for each output from the controller and this problem involves a one-output controller, each program tree in the population has one result-producing branch. Each program tree also has up to five automatically defined functions. Each program tree in the initial random population (generation 0) has no automatically defined functions. However, in subsequent generations, architecture-altering operations may insert and delete automatically defined functions to particular individual program trees in the population. The insertion of an automatically defined function is a precondition for the creation of internal feedback. Thus, the architecture-altering operations that create automatically defined functions are the vehicle by which genetic programming can create internal feedback within a controller.

5.2 Terminal Set

The terminal set, T, for the result-producing branch and any automatically defined functions for this problem is

T = {ℜ, CONSTANT_0, REFERENCE_SIGNAL, CONTROLLER_OUTPUT, PLANT_OUTPUT}.

5.3 Function Set

The function set, F, for the result-producing branch and any automatically defined functions for this problem is

F = {GAIN, INVERTER, LEAD, LAG, LAG2, DIFFERENTIAL_INPUT_INTEGRATOR, DIFFERENTIATOR, ADD_SIGNAL, SUB_SIGNAL, ADD_3_SIGNAL, ADD_NUMERIC, SUB_NUMERIC, ADF0, ADF1, ADF2, ADF3, ADF4}.

5.4 Fitness

Genetic programming is a probabilistic search algorithm through the space of compositions of the available functions and terminals. The search is guided by a fitness measure. The fitness measure is usually couched in terms of the high-level requirements of the problem. It expresses "what needs to be done" — not "how to do it."

The fitness of each individual in the population is determined by executing the program tree (i.e., the result-producing branch and any automatically defined functions that may be present) to produce an interconnected sequence of signal processing blocks — that is, the block diagram for the controller. Each individual controller is then evaluated by simulating the controller using our modified version of the SPICE simulator (Quarles, Newton, Pederson, and Sangiovanni-Vincentelli 1994).

For this problem, the fitness of a controller is measured using 10 components, including eight components based on a modified ITAE ("Integral of Time-weighted Absolute Error"), one frequency-based component, and one time-domain-based component.

The fitness of an individual controller is the sum of the detrimental contributions of these 10 measurements. The smaller the fitness, the better.

The first eight components of the fitness measure represent choices of a particular one of two different values of the internal gain *K*, a particular one of two different values of the time constant τ, and a particular one of two different values for the height of the reference signal (which is a step function). The

two values of K are 1.0 and 2.0. The two values of τ are 0.5 and 1.0. The first reference signal is a step function that rises from 0 to 1 volts at t = 100 milliseconds. Two values of K and τ are used in order to obtain a robust controller. The second reference signal rises from 0 to 1 microvolts at t = 100 milliseconds. We use two step functions to deal with the system's nonlinearity caused by the voltage limiter. The contribution to fitness for each of these eight components of the fitness measure is

$$\int_{t=0}^{9.6} t\left|e(t)\right| A(e(t))dt \ .$$

Here $e(t)$ is the difference at time t between the plant output and the reference signal. A is an additional weight that varies depending on the overshoot. It heavily penalizes all unacceptable values of overshoot (with a weight of 10) and slightly penalizes acceptable values (with a weight of 1).

The ninth component of the fitness measure is based on 121 fitness cases representing an AC sweep over 20 sampled frequencies (equally spaced on a logarithmic scale) in each of six decades of frequency between 0.01 Hz and 10,000 Hz. A gain of 0 dB is ideal for the 80 fitness cases in the first four decades of frequency between 0.01 Hz and 100 Hz; however, a gain of up to +3 dB is acceptable. The contribution to fitness for each of these 80 fitness cases is zero if the gain is ideal or acceptable, but 18/121 per fitness case otherwise. The ideal gain for the 41 fitness cases in the two decades between 100 Hz and 10,000 Hz is given by the straight line connecting (100 Hz, -3 dB) and (10,000 Hz, -83 dB) with a logarithmic horizontal axis and a linear vertical axis. The contribution to fitness for each of these fitness cases is zero if the gain is on or below this straight line, but otherwise 18/121 per fitness case.

The tenth component of the fitness measure is based on a time-domain analysis of the effect for 120 microseconds of a 10-nanosecond pulse that rises to 10^{-9} volts at time t = 0. If the absolute value of plant output goes above 10^{-8} volts (i.e., a order of magnitude greater than the pulse's magnitude), then the contribution to fitness is 500(120 - t), where t is first time (in microseconds) at which the absolute value of plant output goes above 10^{-8} volts, but 0 volts otherwise.

5.5 Control Parameters

The population size, M, is 66,000. The maximum size of each result-producing branch is 150 points (functions and terminals). The maximum size of each automatically defined function is 100 points. The architecture-altering operations are generally used sparingly on each generation. The percentages of the genetic operations on each generation on and after generation 5 are 86% one-offspring crossover, 10% reproduction, 1% mutation 1% subroutine creation, 1% subroutine duplication, and 1% subroutine deletion. Since all the programs in generation 0 have a minimalist architecture consisting of just one result-producing branch, we accelerate the appearance of automatically defined functions in the population by using an increased percentage for the architecture-altering operations prior to generation 5. Specifically, the percentages for the genetic operations on each generation up to and including generation 5 are 78% one-offspring crossover, 10% reproduction, 1% mutation, 5% subroutine creation, 5% subroutine duplication, and 1% subroutine deletion. Other parameters for controlling the run are those found in Koza, Bennett, Andre, and Keane 1999.

5.6 Termination

The maximum number of generations, G, is set to an arbitrary large number (e.g., 501) and the run was manually monitored and manually terminated when the fitness of many successive best-of-generation individuals appeared to have reached a plateau. The single best-so-far individual is harvested and designated as the result of the run.

5.7 Parallel Implementation

This problem was run on a home-built Beowulf-style (Sterling, Salmon, Becker, and Savarese 1999) parallel cluster computer system consisting of 66 processing nodes (each containing a 533-MHz DEC Alpha microprocessor and 64 megabytes of RAM) arranged in a two-dimensional 6 × 11 toroidal mesh. The system has a DEC Alpha type computer as host. The processing nodes are connected with a 100 megabit-per-second Ethernet. The processing nodes and the host use the Linux operating system. The distributed genetic algorithm (Andre and Koza 1996) was used with a population size of Q = 1,000 at each of the D = 66 demes (semi-isolated subpopulations). Generations are asynchronous on the nodes. On each generation, four boatloads of emigrants, each consisting of B = 2% (the migration rate) of the node's subpopulation (selected probabilistically on the basis of fitness) were dispatched to each of the four adjacent processing nodes. Details are found in Koza, Bennett, Andre, and Keane 1999 and Bennett, Koza, Shipman, and Stiffelman 1999.

6 Results

The best individual from generation 0 of our one and only run of this problem had a fitness of 8.26.

The best-of-run individual emerged in generation 32 and had a near-zero fitness of 0.1639. Figure 2 shows this individual controller in the form of a block

diagram. Table 1 shows the contribution of each of the 10 components of the fitness measure.

Table 1 Fitness of best-of-run individual of generation 32.

	Step size (volts)	Internal Gain	Time constant	Fitness
1	1	1	1.0	0.0220
2	1	1	0.5	0.0205
3	1	2	1.0	0.0201
4	1	2	0.5	0.0206
5	10^{-6}	1	1.0	0.0196
6	10^{-6}	1	0.5	0.0204
7	10^{-6}	2	1.0	0.0210
8	10^{-6}	2	0.5	0.0206
9	AC sweep			0.0000
10	Pulse test			0.0000
TOTAL FITNESS				0.1639

After applying standard block diagram manipulations, the transfer function for the pre-filter of the best-of-run individual from generation 32 is

$$G_{p32}(s) = \frac{1(1+.126s)(1+.203s)}{(1+.038s)(1+.052s)(1+.084s)(1+.16s)(1+.17s)}$$

The transfer function for the compensator of the best-of-run individual from generation 32 is

$$G_{c32}(s) = \frac{7487(1+.03851s)(1+.05146s)(1+.08375s)}{s}$$

Note that this transfer function indicates that the compensator consists of a second derivative processing block in addition to proportional, integrative, and derivative blocks.

For comparison, the transfer function for the pre-filter of the controller presented in Dorf and Bishop 1998 is

$$G_{p-dorf}(s) = \frac{42.67}{42.67 + 11.38s + s^2}$$

and the transfer function for the compensator is

$$G_{c-dorf}(s) = \frac{12(42.67 + 11.38s + s^2)}{s}.$$

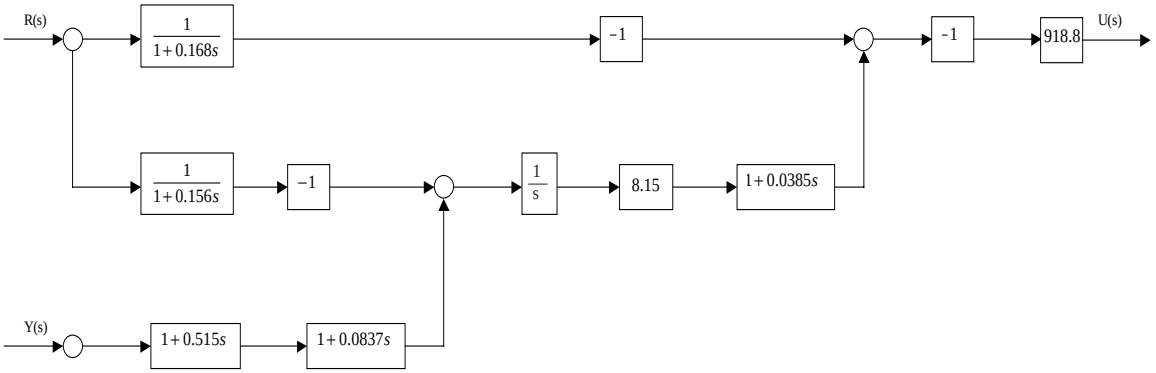

Figure 2 Best-of-run genetically evolved controller from generation 32.

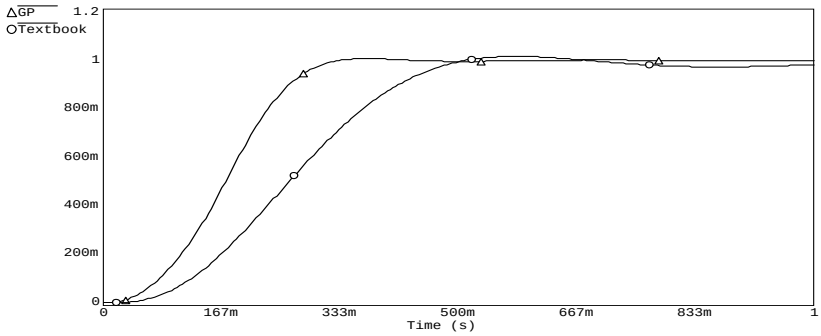

Figure 3 Time-domain response to step input.

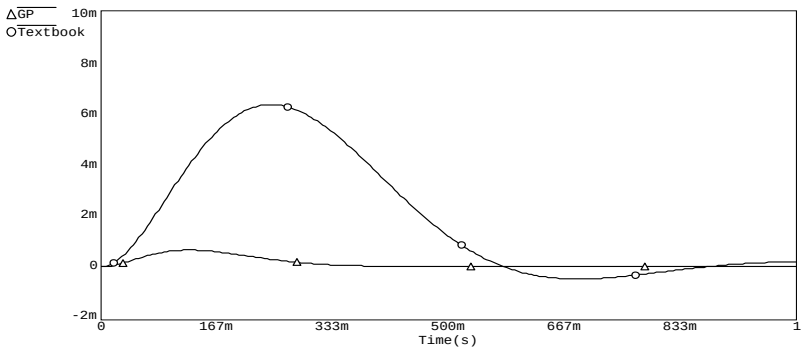

Figure 4 Time-domain response to distriburance.

The faster rising curve in figure 3 is the time-domain response of the best-of-run controller from generation 32 for a 1 volt unit step with K=1 and τ=1. The slower-rising curve shows the time-domain response of the controller presented in Dorf and Bishop 1998. The curves for other values of K and τ similarly favor the genetically evolved controller.

The upper curve in figure 4 is the time-domain response to a 1-volt disturbance signal with K=1 and τ=1 for the controller presented in Dorf and Bishop 1998. The lower curve applies to the best-of-run controller from generation 32. The curves for other values of K and τ are similar.

Figure 5 shows the magnitude portion of the Bode plot of the open loop transfer function versus the frequency of input.

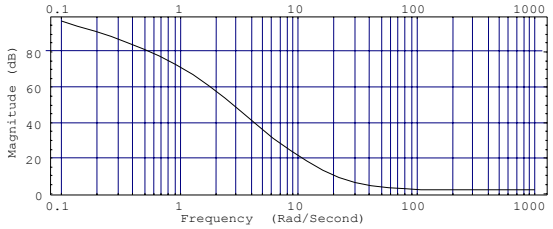

Figure 5 Magnitude portion of Bode plot.

Figure 6 shows the phase portion of the Bode plot of the open loop transfer function versus the frequency of input.

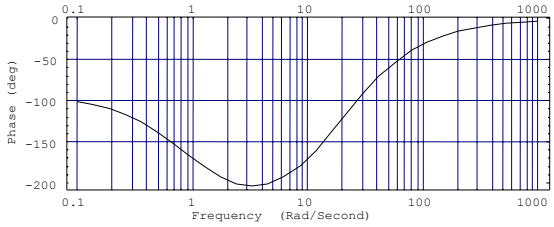

Figure 6 Phase portion of Bode plot.

Most of the computer time was consumed by the fitness evaluation of candidate individuals in the population. The fitness evaluation (involving 10 SPICE simulations) averaged 2.57×10^9 computer cycles (4.8 seconds) per individual. The best-of-run individual from generation 32 was produced after evaluating 2.178×10^6 individuals (66,000 times 33). This required 44.5 hours on our 66-node parallel computer system — that is, the expenditure of 5.6×10^{15} computer cycles (5 peta-cycles).

7 Conclusion

Genetic programming automatically created a robust controller for a plant with a second-order lag without the benefit of user-supplied information concerning the total number of processing blocks to be employed in the controller, the type of each processing block, the topological interconnections between the blocks, the values of parameters for the blocks, or the existence of internal feedback, if any. The genetically evolved controller employs a second derivative processing block (in addition to proportional, integrative, and derivative blocks and a pre-filter).

Table 2 compares the genetically evolved controller to the controller for this problem presented in Dorf and Bishop 1998.

Table 2 Comparison.

	PID / Dorf	**Genetically evolved controller**	**Units**
ITAE	46	19	millivolt sec^2
Rise time	465	296	milliseconds
Disturbance sensitivity	5,775	644	μVolts /Volt
Bandwidth	1	1.5	Hz

The genetically evolved controller is better than twice as effective as the textbook controller as measured by the integral of the time-weighted absolute error (ITAE), has only two-thirds of the rise time in response to the reference input, and is 10 times better in terms of suppressing the effects of disturbance at the plant input. The system bandwidth of two controllers are comparable.

References

Andre, David and Koza, John R. 1996. Parallel genetic programming: A scalable implementation using the transputer architecture. In Angeline, P. J. and Kinnear, K. E. Jr. (editors). 1996. *Advances in Genetic Programming 2*. Cambridge: MIT Press.

Angeline, Peter J. and Kinnear, Kenneth E. Jr. (editors). 1996. *Advances in Genetic Programming 2*. Cambridge, MA: The MIT Press.

Banzhaf, Wolfgang, Nordin, Peter, Keller, Robert E., and Francone, Frank D. 1998. *Genetic Programming – An Introduction*. San Francisco, CA: Morgan Kaufmann and Heidelberg: dpunkt.

Banzhaf, Wolfgang, Poli, Riccardo, Schoenauer, Marc, and Fogarty, Terence C. 1998. *Genetic Programming: First European Workshop. EuroGP'98. Paris, France, April 1998 Proceedings. Paris, France. April l998*. Lecture Notes in Computer Science. Volume 1391. Berlin, Germany: Springer-Verlag.

Bennett, Forrest H III, Koza, John R., Shipman, James, and Stiffelman, Oscar. 1999. Building a parallel computer system for $18,000 that performs a half peta-flop per day. In Banzhaf, Wolfgang, Daida, Jason, Eiben, A. E., Garzon, Max H., Honavar, Vasant, Jakiela, Mark, and Smith, Robert E. (editors). 1999. *GECCO-99: Proceedings of the Genetic and Evolutionary Computation Conference, July 13-17, 1999, Orlando, Florida USA*. San Francisco, CA: Morgan Kaufmann.

Dorf, Richard C. and Bishop, Robert H. 1998. *Modern Control Systems*. Eighth edition. Menlo Park, CA: Addison-Wesley.

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

Kinnear, Kenneth E. Jr. (editor). 1994. *Advances in Genetic Programming*. Cambridge, MA: The MIT Press.

Koza, John R., and Keane, Martin A. 1990a. 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*. Hillsdale, NJ: Lawrence Erlbaum. Volume I, Pages 198-201.

Koza, John R., and Keane, Martin A. 1990b. 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*. Berlin: Springer-Verlag. Pages 47-56.

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

Koza, John R. 1994a. *Genetic Programming II: Automatic Discovery of Reusable Programs*. Cambridge, MA: MIT Press.

Koza, John R. 1994b. *Genetic Programming II Videotape: The Next Generation*. Cambridge, MA: MIT Press.

Koza, John R., Banzhaf, Wolfgang, Chellapilla, Kumar, Deb, Kalyanmoy, Dorigo, Marco, Fogel, David B., Garzon, Max H., Goldberg, David E., Iba, Hitoshi, and Riolo, Rick. (editors). 1998. *Genetic Programming 1998: Proceedings of the Third Annual Conference*. San Francisco, CA: Morgan Kaufmann.

Koza, John R., Bennett III, Forrest H, Andre, David, and Keane, Martin A. 1999. *Genetic Programming III: Darwinian Invention and Problem Solving*. San Francisco, CA: Morgan Kaufmann. Forthcoming.

Koza, John R., Bennett III, Forrest H, Andre, David, Keane, Martin A., and Brave Scott. 1999. *Genetic Programming III Videotape*. San Francisco, CA: Morgan Kaufmann. Forthcoming.

Koza, John R., Deb, Kalyanmoy, Dorigo, Marco, Fogel, David B., Garzon, Max, Iba, Hitoshi, and Riolo, Rick L. (editors). 1997. *Genetic Programming 1997: Proceedings of the Second Annual Conference* San Francisco, CA: Morgan Kaufmann.

Koza, John R., Goldberg, David E., Fogel, David B., and Riolo, Rick L. (editors). 1996. *Genetic Programming 1996: Proceedings of the First Annual Conference*. Cambridge, MA: The MIT Press.

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

Langdon, William B. 1998. *Genetic Programming and Data Structures: Genetic Programming + Data Structures = Automatic Programming!* Amsterdam: Kluwer.

Quarles, Thomas, Newton, A. R., Pederson, D. O., and Sangiovanni-Vincentelli, A. 1994. *SPICE 3 Version 3F5 User's Manual*. Department of Electrical Engineering and Computer Science, Univ. of California. Berkeley, CA. March 1994.

Spector, Lee, Langdon, William B., O'Reilly, Una-May, and Angeline, Peter (editors). 1999. *Advances in Genetic Programming 3*. Cambridge, MA: The MIT Press.

Sterling, Thomas L., Salmon, John, Becker, Donald J., and Savarese, Daniel F. 1999. *How to Build a Beowulf: A Guide to Implementation and Application of PC Clusters*. Cambridge, MA: MIT Press.