Automatic Design of Both Topology and Tuning of a Common Parameterized Controller for Two Families of Plants using Genetic Programming

Jessen Yu

Genetic Programming Inc., Los Altos, California jyu@cs.stanford.edu

Martin A. Keane

Econometrics Inc., Chicago, Illinois makeane@ix.netcom.com

John R. Koza Stanford University, Stanford, California koza@stanford.edu

ABSTRACT

This paper demonstrates that a technique of evolutionary computation can be used to automatically create the design for both the topology and parameter values (tuning) for a common controller (containing various parameters representing the overall characteristics of the plant) for two families of plants. The automatically designed controller is created by means of genetic programming using a fitness measure that attempts to optimize step response and disturbance rejection while simultaneously constraints imposing on maximum sensitivity and sensor noise attenuation. The automatically designed controller outperforms the controller designed with conventional techniques. In particular, the automatically designed controller is superior to the Astrom and Hagglund controller for all plants of both families for the integral of the time-weighted absolute error (ITAE) for a step input, the ITAE for disturbance rejection, and maximum sensitivity. Averaged over all plants of both families, the ITAE for the step input for the automatically designed controller is only 58% of the value for the conventional controller; the ITAE for disturbance rejection is 91% of the value for the conventional controller; and the maximum sensitivity. $M_{\rm s}$. for the automatically designed controller is only 85% of the value for the conventional controller. The automatically designed controller is "general" in the sense that it contains free variables and therefore provides a solution to an entire category of problems (i.e., all the plants in the two families) — not merely a single instance of the problem (i.e., a particular single plant).

1 Introduction

Genetic programming has recently been used to automatically create the design for both the topology and parameter values (tuning) for a controller for a particular two-lag plant and a particular three-lag plant from a high-level statement of the controller's desired behavior and characteristics (Koza, Keane, Yu, Bennett, and Mydlowec 2000). However, each of these two (different) automatically designed controllers applied only to one particular plant. Moreover, both plants belonged to the same family (the *n*-lag plants).

The question arises as to whether it is possible to evolve a single "general" parameterized controller (containing parameters representing the overall characteristics of the plant) that can perform well for an entire family of plants (say, the *n*-lag plants) and also one or more additional families of plants.

In their recent influential book, Astrom and Hagglund (1995) identified four families of plants "that are representative for the dynamics of typical industrial processes." Astrom and Hagglund (1995) then develop a common method for designing controllers and demonstrate improved performance for their common method over the Ziegler-Nichols rules (Ziegler and Nichols 1942) on all the plants in all four of their families of plants.

One of the families of plants in Astrom and Hagglund 1995 consists of the n-lag plants represented by the transfer functions of the form

$$G(s) = \frac{1}{\left(1+s\right)^n} \tag{1}$$

where n = 3, 4, and 8.

Another family consists of plants represented by the transfer functions of the form

$$G(s) = \frac{1}{(1+s)(1+\alpha s)(1+\alpha^2 s)(1+\alpha^3 s)}$$
(2)

where $\alpha = 0.2, 0.5, \text{ and } 0.7$.

The methods developed by Astrom and Hagglund use pairs of parameters representing the overall characteristics of a plant. These parameters are not, of course, a complete representation of the behavior of the plant; however, they offer the practical advantage of usually being obtainable for a given plant by means of relatively straight-forward testing. In one version of their method, Astrom and Hagglund use two frequency domain parameters. They are the ultimate gain, $K_{\rm u}$ (the minimum value of the gain that must be introduced into the feedback path to cause a system to oscillate) and the ultimate period, $T_{\rm u}$ (the period of this lowest frequency oscillation). In another version of their method, Astrom and Hagglund use the time constant, $T_{\rm r}$, and the dead time, L. Astrom and Hagglund describe a procedure for estimating these two parameters from the plant's response to a step input. These two parameters are, in one instance, obtained by approximating the plant with a transfer function of the form

$$\frac{e^{-sL}}{\left(1+sT_r\right)^2}$$

This paper shows that genetic programming can be used to automatically create the design for both the topology and tuning for a common parameterized controller for all plants belonging to the two families of plants described by equations (1) and (2). The automatically designed controller is created using a fitness measure that attempts to optimize step response and disturbance rejection while simultaneously imposing constraints on maximum sensitivity and sensor noise attenuation. The automatically designed controller outperforms the controller designed using the techniques of Astrom and Hagglund 1995.

Section 2 discusses how genetic programming can be used to automatically synthesize the design for both the topology and tuning of controllers. Section 3 itemizes the preparatory steps necessary to apply genetic programming to the above two families of plants. Section 4 presents the results.

2 Genetic Programming and Control

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; Koza 1994a, 1994b) is an extension of the genetic algorithm (Holland 1975).

Genetic programming starts with a primordial ooze of thousands of randomly created computer programs and uses the Darwinian principle of natural selection, recombination (crossover), mutation, gene duplication and deletion to breed a population of programs over a series of generations.

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

(1) Generate an initial population of compositions (typically random) of the problem's functions and terminals.

(2) Iteratively perform the following substeps (a generation) on the population of programs until the termination criterion has been satisfied:

(A) Execute each program in the population and assign it a value using the fitness measure.

(B) Create a new population of programs by applying the following operations. The operations are applied to program(s) selected from the population with a probability based on fitness (with reselection allowed).

(i) Reproduction: Copy the selected program to the new population.

(ii) Crossover: Create a new offspring program for the new population by recombining randomly chosen parts of two selected programs.

(iii) Mutation: Create one new offspring program for the new population by randomly mutating a randomly chosen part of the selected program.

(iv) Architecture-altering operations: Select an architecture-altering operation from the repertoire of such operations and create one new offspring program for the new population by applying the selected operation to the selected program.

(3) Designate the individual program that is identified by result designation (e.g., the best-so-far individual) as the result of the run of genetic programming. This result may be a solution (or an approximate solution) to the problem.

Genetic programming is capable of evolving reusable, parametrized, hierarchically-called automatically defined functions (subroutines). 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.

Genetic programming is capable of automatically 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 (Koza, Bennett, Andre, and Keane 1999). Nine of the automatically designed analog circuits in Koza, Bennett, Andre, and Keane 1999 were previously patented. Five of the automatically designed circuits infringe on previously issued patents. 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. The fact that genetic programming can design both the topology and sizing of circuits suggests that it might also be capable of designing other types of complex topological structures containing parameterized components, such as controllers.

In a closed-loop continuous-time feedback system consisting of a plant and its controller, the output of the controller is input to the plant and the output of the plant is, in turn, input to the controller.

Both genetic algorithms and genetic programming have been previously used for synthesizing controllers having mutually interacting continuous-time variables and continuous-time signal processing blocks (Koza, Keane, Yu, Mydlowec, and Bennett 2000; Man, Tang, Kwong, and Halang; 1997, 1999; Crawford, Cheng, and Menon 1999; Dewell and Menon 1999; Menon, Yousefpor; Lam, and Steinberg 1995; Sweriduk, Menon, and Steinberg 1998, 1999).



Figure 1 Block diagram of a plant and a PID controller composed of proportional, integrative, and derivative blocks. The plant's output is fed back to the controller where it is compared to the reference signal.

Figure 1 is a block diagram for an illustrative control system containing a controller and a plant. The directed lines in a block diagram represent time-domain signals while the blocks represent signal processing functions that operate in the time domain. The output of the controller 500 is a control variable 590 which is, in turn, the input to the plant 592. The plant has one output (plant response) 594. The plant response is fed back (externally as signal 596) and becomes one of the controller's two inputs. The controller's second input is the reference signal 508. The fed-back plant response 596 and the externally supplied reference signal 508 are compared (by subtraction here). Notice that the takeoff point 520 of figure 1 provides a way to disseminate a particular result (of the subtraction 510) to three places in the block diagram (522, 524, and 526).

The output (i.e., control variable 590) of this controller is the sum of a proportional (P) term (the gain block 530 with an amplification factor of 214.0), an integrating (I) term (the integrator 560 preceded by the gain block 540 with an amplification factor of 1,000.0), and a differentiating (D) term (the derivative block 570 preceded by the gain block 550 with an amplification factor of 15.5). This type of controller is called a PID controller and was invented and patented in 1939 by Albert Callender and Allan Stevenson of Imperial Chemical Limited of Northwich, England.

In this paper, a computer program (i.e., program tree, LISP symbolic expression) will represent the block diagram of a controller. The block diagram consists of signal processing functions linked by directed lines representing the flow of information. There is no "order of evaluation" of the functions and terminals of a program tree representing a controller. Instead, the signal processing blocks of the controller and the to-be-controlled plant interact with one another other as part of a closed system in the manner specified by the topology of the block diagram.



Figure 2 Program tree representation of the PID controller of figure 1. The automatically defined function ADF0 (left) subtracts the plant output from the reference signal and makes the difference available to three points in the result-producing branch (right).

Figure 2 presents the block diagram for the PID controller of figure 1 as a program tree. The internal points of this program tree represent the signal processing blocks contained in the block diagram of figure 1 (i.e., derivative, integrator, gain, subtraction, addition). The external points (leaves) of this program tree represent numerical constants and time-domain signals, such as the reference signal and plant output. Notice that automatically defined function (subroutine) ADF0 in the left branch produces a time-domain signal that equals the result of subtracting the plant output from the reference signal. The three references to ADF0 in the result-producing (right) branch of this program tree disseminate the result of subtracting the plant output from the reference signal and correspond to the takeoff point 520 of figure 1.

In the style of ordinary computer programming, a reference to a subroutine ADF0 from inside the function definition for itself would be considered to be a recursive reference. However, in the context of applying genetic programming to control systems, a subroutine that references itself corresponds to a loop

in the block diagram of the controller (i.e., internal feedback inside the controller).

3 Preparatory Steps

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

3.1 Program Architecture

Since the to-be-synthesized controller has one output (control variable), each program tree in the population has one result-producing branch. Each program tree in the initial random population (generation 0) has no automatically defined functions. However, after generation 0, the architecture-altering operations may insert (and delete) automatically defined functions. Automatically defined functions may be used for takeoff points, internal feedback within the controller, and reuse of portions of the block diagram. The permitted maximum of five automatically defined functions is more than sufficient for this problem.

3.2 Terminal Set

The numerical parameter value for each signal processing block possessing a parameter is established by an arithmetic-performing subtree containing perturbable numerical terminals, arithmetic operations, and the four parameters for representing the overall characteristics of a plant. Arithmetic-performing subtrees may appear in both result-producing branches and any automatically defined functions that may be created during the run by the architecture-altering operations. The value returned by an entire arithmetic-performing subtree is interpreted as a component value lying in a range of (positive values) between 10^{-3} and 10^{3} . The terminal set for the arithmetic-performing subtrees is

 $\mathsf{T}_{\mathsf{aps}} = \{ \Re, \, \texttt{KU}, \, \texttt{TU}, \, \texttt{L}, \, \texttt{TR} \}.$

Here \Re denotes a perturbable numerical value. In the initial random generation (generation 0) of a run, each perturbable numerical value is set, individually and separately, to a random value in a chosen range (from - 3.0 and +3.0 here). In later generations, a perturbable numerical value may be changed by adding or subtracting a relatively small number determined probabilistically by a Gaussian probability distribution. The standard deviation of the Gaussian distribution is 1.0 here (i.e., one order of magnitude after the value returned by an entire arithmetic-performing subtree is interpreted). The perturbations are implemented by a genetic operation for mutating the perturbable

numerical values. The perturbable numerical values are coded by 30 bits in our system. A constrained syntactic structure maintains one function and terminal set for the arithmetic-performing subtrees and a different function and terminal set (below) for all other parts of the program tree.

The remaining terminals are time-domain signals. The terminal set, T, for the result-producing branch and any automatically defined functions (except the arithmetic-performing subtrees described above) is

 $T = \{ \text{REFERENCE} SIGNAL, \}$

CONTROLLER_OUTPUT, PLANT_OUTPUT}.

Space does not permit a detailed description of the various terminals used herein (although the meaning of the above terminals should be clear from their names). See Koza, Keane, Yu, Bennett, and Mydlowec 2000.

3.3 Function Set

The function set, $\mathsf{F}_{\mathsf{aps}},$ for the arithmetic-performing subtrees is

```
Faps = {ADD_NUMERIC, SUB_NUMERIC,
MUL_NUMERIC, DIV_NUMERIC, REXP,
RLOG}.
```

The two-argument DIV_NUMERIC function divides the first argument by the second argument, except that the quotient is never allowed to exceed 10^5 . The oneargument REXP function is the exponential function and the one-argument RLOG function is the natural logarithm of the absolute value.

The function set, F, for the result-producing branch and any automatically defined functions (except the arithmetic-performing subtrees described above) consists of continuous-time signal processing functions and automatically defined functions.

 $F = \{GAIN, INVERTER, LEAD, LAG, LAG2, \}$

DIFFERENTIAL_INPUT_INTEGRATOR, DIFFERENTIATOR, ADD_SIGNAL, SUB_SIGNAL, ADD_3_SIGNAL, MUL_SIGNAL, DIV_SIGNAL, ULIMIT, ADF0, ADF1, ADF2, ADF3, ADF4}.

The one-argument ULIMIT function limits a signal by constraining it between an upper and lower bound. This function returns the value of its argument (the incoming signal) when its argument lies between -1.0 and +1.0. If the argument is greater than +1.0, the function returns +1.0. If the argument is less than -1.0, the function returns -1.0. ADF0, ..., ADF4 denote automatically defined functions added during the run by the architecture-altering operations. The definitions of the other functions above are suggested by their names. See Koza, Keane, Yu, Bennett, and Mydlowec 2000.

3.4 Fitness Measure

Genetic programming is a probabilistic algorithm that searches the space of compositions of the available functions and terminals under the guidance of a fitness measure. The fitness measure is a mathematical implementation of the problem's high-level requirements. It is couched in terms of "what needs to be done" — not "how to do it." The fitness measure for most problems of controller design is multi-objective in the sense that there are several different (usually conflicting) requirements for the controller.

The fitness of each individual in the population is determined by executing the program tree (i.e., the result-producing branch plus any automatically defined functions that may have been created during the run by the architecture-altering operations). The execution of the program tree produces an interconnected sequence of signal processing blocks — that is, a block diagram for the individual controller. The controller is embedded into a framework containing the (fixed) plant and the (fixed) external feedback loop. A SPICE netlist is then constructed to represent the block diagram of the controller, the (fixed) plant, and the (fixed) external feedback loop. This SPICE netlist is wrapped inside an appropriate set of SPICE commands to carry out various SPICE analyses in the time domain (described below). We also provide SPICE with subcircuit definitions to implement all the signal processing functions in the function set (described above) and all the signal processing functions necessary to represent the plant. The controller is then simulated using our modified version of the original 217,000-line SPICE3 simulator (Quarles, Newton, Pederson, and Sangiovanni-Vincentelli 1994). Our modified version of SPICE is run as a submodule within our genetic programming system. The SPICE simulator returns tabular output (representing the plant output in the time domain). An interface communicates this information to our genetic programming code. See Koza, Keane, Yu, Bennett, and Mydlowec 2000 for details.

The fitness of each controller in the population is measured by means of 48 separate invocations of the SPICE simulator. This 48-part fitness measure attempts to optimize step response and disturbance rejection while simultaneously imposing constraints on maximum sensitivity and sensor noise attenuation. The fitness of an individual controller is the sum of the detrimental contributions of these 48 elements of the fitness measure. The smaller the sum, the better.

Table 1 Six combinations							
Reference signal	Disturbance signal						
1.0	1.0						
10 ⁻³	10 ⁻³						
-10 ⁻⁶	10 ⁻⁶						
1.0	-0.6						
-1.0	0.0						
0.0	1.0						

Table 1 Six combinations

The first 36 elements of this 48-part fitness measure are time-domain-based elements that together represent the six plants from the two families (i.e., n = 3, 4, and 8

and $\alpha = 0.2$, 0.5, and 0.7), in conjunction with six choices of values for the height of the reference signal and disturbance signal (shown in table 1) that sample a range of values. The reference signal is step function that rises from 0 at time t = 0 to the specified height at t = 1 millisecond. The disturbance signal is a step function that rises from 0 at time $t = 10T_u$ to the specified height at $t = 10T_u + 1$ millisecond. The disturbance signal is added to the controller's output.

For each of these first 36 elements of the 48-part fitness measure, a transient analysis is performed in the time domain using the SPICE simulator. e(t) is the difference (error) at time t between the plant output and the reference signal. The contribution to fitness for each of these 36 elements is based on the sum of two integrals of time-weighted absolute error (ITAE). The first term of the integral accounts for the controller's step response while the second term accounts for disturbance rejection.

$$\frac{\int_{t=0}^{10T_u} t |e(t)| Bdt}{T_u^2} + \frac{\int_{t=10T_u}^{20T_u} (t-10T_u) |e(t)| Cdt}{T_u^2}.$$

The factor B in the first term of the integral multiplies each value of e(t) by the reciprocal of the amplitude of the reference signal (so that all reference signals are equally influential). The factor C in the second term of the integral multiplies value of e(t) by the reciprocal of the amplitude of the disturbance signals. When the amplitude of either the reference signal or the disturbance signal is zero, the appropriate factor (B or C) is set to zero. The ITAE component of fitness is such that, all other things being equal, changing the time scale by a factor of F changes the ITAE by F^2 . The division of the integral by T_u^2 is an attempt to eliminate this artifact of the time scale and equalize the influence of each of the plants in the overall fitness measure. For these 36 elements of the fitness measure, the contribution to fitness is multiplied by 20 if the element is greater than for the Astrom and Hagglund (1995).

The 37th through 42nd elements of the 48-part fitness measure are frequency-domain-based elements that measure stability margin. Figure 3 presents a model for the entire system containing the given plant and the tobe-evolved controller. In this figure, R(s) is the reference signal; Y(s) is the plant output; and U(s) is the controller's output (control variable). Disturbance D(s)may be added to the controller's output U(s). Sensor noise N(s) may be added to the plant's output Y(s)yielding Q(s). Here N(s) is an AC signal. For each of these six elements of the fitness measure, an AC sweep is performed using the SPICE simulator from $1/(1000T_{\rm u})$ to $1000/T_{\rm u}$ while holding the reference signal R(s) and the disturbance signal D(s) at zero. The maximum sensitivity, M_s, is a measure of the stability margin. It is desirable to minimize the maximum sensitivity (and therefore maximize the stability margin). The quantity $1/M_s$ is the minimum distance between the Nyquist plot and the point (-1,0) and is the stability margin incorporating both gain and phase margin. The maximum sensitivity is the maximum amplitude of Q(s). The contribution to fitness is 0 if $M_s < 1.5$; $2(M_s - 1.5)$ for $1.5 \le M_s \le 2.0$; and $20(M_s - 2.0) + 1$ for $M_s > 2.0$. For these six elements of the fitness measure (as well as the six elements below), the contribution to fitness is multiplied by 10 if the element is greater than for the Astrom and Hagglund controller.



Figure 3 Overall model.

The 43rd through 48th elements of this 48-part fitness measure are frequency-domain-based elements measuring the sensor noise attenuation. Achieving favorable sensor noise attenuation is often in direct conflict with the goal of achieving a rapid response to setpoint changes and rejection of plant disturbances. For each of these six elements of the fitness measure, an AC sweep is performed using the SPICE simulator from $10/T_u$ to $1000/T_u$ while holding the reference signal R(s) and the disturbance signal, D(s) at zero. The attenuation of the sensor noise is measured at plant output at Y(s). A_{\min} is the minimum attenuation in decibels within this frequency range. It is desirable to maximize the minimum attenuation. The contribution to fitness for sensor noise attenuation is 0 if $A_{\min} > 40$ dB; $(40 - A_{\min})/10$ if 20 dB $\leq A_{\min} \leq 40$ dB; and 2 + (20 - A_{\min}) if $A_{\min} < 20 \text{ dB}$.

The SPICE simulator cannot simulate many of the controllers that are randomly created during a run of genetic programming. A controller that cannot be simulated by SPICE is assigned a high penalty value of fitness (10^8).

3.5 Control Parameters

The population size, M, was 100,000. A (generous) maximum size of 150 points (for functions and terminals) was established for each result-producing branch and a (generous) maximum size of 100 points was established for each automatically defined function. The percentages of the genetic operations for each generation are 46% one-offspring crossover on internal points of the program tree other than numerical constant terminals, 9% one-offspring crossover on points of the program tree other than numerical constant terminals, 9% one-offspring crossover on numerical constant terminals, 1% mutation on points of

the program tree other than numerical constant terminals, 20% mutation on numerical constant terminals, 9% reproduction, 2% subroutine creation, 2% subroutine duplication, and 2% subroutine deletion. The other parameters are the same default values that we have used on many other problems (Koza, Bennett, Andre, Keane 1999).

3.6 Termination

The run was manually monitored and manually terminated when the fitness of many successive best-ofgeneration individuals appeared to have reached a plateau. The best-so-far individual was harvested and designated as the result of the run.

3.7 Parallel Implementation

This problem was run on a home-built Beowulf-style (Sterling, Salmon, Becker, and Savarese 1999; Bennett. Koza, Shipman, and Stiffelman 1999) parallel cluster computer system consisting of 1,000 350 MHz Pentium II processors (each accompanied by 64 megabytes of RAM). The system has a 350 MHz Pentium II 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 with unsynchronized generations and semi-isolated subpopulations was used with a subpopulation size of Q = 100 at each of D = 1,000 demes. As each processor (asynchronously) completes a generation, four boatloads of emigrants from each subpopulation are dispatched to each of the four toroidally adjacent processors. The 1,000 processors are hierarchically organized. There are $5 \times 5 = 25$ high-level groups (each containing 40 processors). If the adjacent node belongs to a different group, the migration rate is 2% and emigrants are selected based on fitness. If the adjacent node belongs to the same group, the migration rate is 5% (10% if in the same physical box) and emigrants are selected randomly.

4 **Results**

The initial random generation is a blind random search of the search space of the problem. The best-of-generation circuit from generation 0 has a fitness of 14,530.8.

The best-of-run controller (figure 4) appears in generation 217. This genetically evolved controller has an overall fitness of 14.996. The program tree has one result-producing branch with 10 points and five automatically defined functions (with 22, 38, 3, 19, and 3 points, respectively). The result-producing branch refers to ADF0. Also, ADF0 hierarchically refers to ADF1. The other three automatically defined functions are not referenced. Note that the controller's output is fed back internally into the controller.

Table 2 presents the control signal, U(s), for the best-of-run controller from generation 217. Note that all four parameters (K_u , T_u , T_r , and L) appear.

When simplified, it can be seen that the best-of-run controller from generation 217 is a PID controller whose three coefficients are as shown in table 3.



Figure 4 Block diagram of best-of-run controller from generation 217.

Table 2 C	ontrol signal.	U(s), for the	best-of-run	controller from	generation 217
	Unti Ul Signala		DUST-01-1 un	contronce in our	

$U(s) = \frac{1}{2}$	$(1 + T_r s)(1 + T_u s)\ln(K_u) + (1 + T_r s)(\ln(K_u))^2 + \ln(K_u)$	$\left(K_{u}\left(1+e^{T_{r}-K_{u}}\right)\right)$	$(\ln(K_u + 1.334419L) - 1)$
O(3) =	$T_{\mu}s$		

Table 3 Three coefficients of PID controller equivalent to the best-of-run controller from generation 217 $K = \frac{\ln(K_u)(T_u + T_r(1 + \ln(K_u)))}{T_u}$ $K_d = \ln(K_u)T_r$ $K_i = \frac{\ln(K_u)(1 + \ln(K_u)) + (\ln(K_u) + \ln(1 + e^{T_r - K_u}))(\ln(K_u + 1.33419L) - 1)}{T_u}$

Plant	Plant	Genetically evolved Controller					Astrom and Hagglund Controller			
		ITAE Step	ITAE Disturb	M _s	A_{\min}	ITAE Step	ITAE Disturb	M _s	A_{\min}	
1	$\frac{1}{(1+s)^3}$	1.93	1.15	1.66	41.26	2.75	1.28	2.11	43	
2	$\frac{1}{(1+s)^4}$	4.60	6.64	1.80	54.17	8.5	7.6	2.08	58	
3	$\frac{1}{\left(1+s\right)^8}$	27.43	74.20	1.69	85.8	49.7	78	1.87	90	
4	$\frac{1}{(1+s)(1+0.2s)(1+0.2^2s)(1+0.2^3s)}$	0.037	0.0055	1.59	35.23	0.051	0.006	1.9	40	
5	$\frac{1}{(1+s)(1+0.5s)(1+0.5^2s)(1+0.5^3s)}$	0.436	0.498	1.72	47.5	0.945	0.522	2.07	50	

Table 4 Comparison of characteristics of the controller and the Astrom and Hagglund controller for all six plants

6	1	1.40	1.99	1.77	52.2	2.9	2.23	2.07	55
	$\overline{(1+s)(1+0.7s)(1+0.7^2s)(1+0.7^3s)}$								

Table 4 compares the characteristics of the best-ofrun controller from generation 217 with those of the Astrom and Hagglund (1995) controller for all six plants. As can be seen, the genetically evolved controller is superior to the Astrom and Hagglund controller for all six plants for the integral of the timeweighted absolute error (ITAE) for the step input, the ITAE for disturbance rejection, and the maximum sensitivity, $M_{\rm s}$. All values of $A_{\rm min}$ are above the required minimum 40 (except for plant 4). Averaged over the six plants, the ITAE for the step input for the genetically evolved controller is only 58% of the value for the Astrom and Hagglund controller; the ITAE for disturbance rejection is 91% of the value for the Astrom and Hagglund controller; and the maximum sensitivity, $M_{\rm s}$. for the genetically evolved controller is only 85% of the value for the Astrom and Hagglund controller.

The values of the PID coefficients of the controller created by genetic programming are very close to those of the Astrom and Hagglund (1995) controller.

The best-of-run controller from generation 217 is similarly superior for other reference signals, other disturbance signals, and other plants from the two families (but are not shown for reasons of space).





Figure 5 compares the time-domain response of the best-of-run controller (triangles) from generation 217 and the Astrom and Hagglund controller (squares) to a 1-volt reference signal for the three-lag plant. Note that the genetically evolved controller (triangles) is superior based on the fitness measure used and the measures contained in table 4.

Figure 6 compares the time-domain response of the best-of-run controller (triangles) from generation 217 and the Astrom and Hagglund controller (squares) to a 1-volt disturbance signal for the three-lag plant.

Most of the computer time in runs of genetic programming involving complex simulations is consumed by the fitness evaluation of candidate individuals in the population. The fitness evaluation for each individual in the population of 100,000 on each of the 218 generations (generation 0 plus 217 additional generations) in this run entailed 36 very timeconsuming time-domain SPICE simulations and 12 relatively fast frequency-domain SPICE simulations. The fitness evaluation for each individual averaged about 6.7 seconds per individual (using a 350 MHz Pentium II processor). The best-of-run individual from generation 217 was produced after evaluating 2.18 × 10^7 individuals. This required 40.58 hours on our 1,000-node parallel computer system — that is, the expenditure of 5.11 × 10^{16} computer cycles (about 51 peta-cycles of computer time).



Figure 6 Comparison of the disturbance responses.

5 Conclusion

This paper demonstrated that genetic programming can be used to automatically create the design for both the topology and parameter values (tuning) for a common parameterized controller (containing various parameters representing the overall characteristics of the plant) for two families of plants. The automatically designed controller outperforms the controller designed with conventional techniques.

A mathematical formula containing one or more free variables is "general" in the sense that it provides a solution to an entire category of problems. For example, the familiar formula for solving a quadratic equation contains free variables representing the coefficients of the equation.

The above result created by genetic programming contains four free variables (K_u , T_u , T_r , and L). That is, genetic programming automatically created a "general" solution to an entire category of problems (i.e., all the plants in the two families) — not merely a solution to single instance of the problem (i.e., a particular single plant).

Moreover, genetic programming did not just automatically create formulae for the controller's parameter values (tuning) — it automatically created the topology of the controller.

Thus, genetic programming can be viewed as a new kind of mathematics in which the result consists of not just general formulae, but, instead, a combination of a graphical structure (i.e., the controller's topology) and general formulae for the parameter values of each block of the controller.

References

- Andersson, Bjorn, Svensson, Per, Nordin, Peter, and Nordahl, Mats. 1999. Reactive and memory-based genetic programming for robot control. In Poli, Riccardo, Nordin, Peter, Langdon, William B., and Fogarty, Terence C. 1999. Genetic Programming: Second European Workshop. EuroGP'99. Proceedings. Lecture Notes in Computer Science. Volume 1598. Berlin, Germany: Springer-Verlag. Pages 161 - 172.
- Astrom, Karl J. and Hagglund, Tore. 1995. *PID Controllers: Theory, Design, and Tuning.* Second Edition. Research Triangle Park, NC: Instrument Society of America.
- Banzhaf, Wolfgang, Nordin, Peter, Keller, Richard, and Olmer, Markus. 1997. Generating adaptive behavior for a real robot using function regression with genetic programming. In Koza, John R., Deb, Kalyanmoy, Dorigo, Marco, Fogel, David B., Garzon, Max, Iba, Hitoshi, and Riolo, Rick L. (editors). Genetic Programming 1997: Proceedings of the Second Annual Conference, July 13–16, 1997, Stanford University. San Francisco, CA: Morgan Kaufmann. Pages 35 - 43.
- 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. Pages 1484 - 1490.
- Callender, Albert and Stevenson, Allan Brown. 1939. Automatic Control of Variable Physical Characteristics. United States Patent 2,175,985. Filed February 17, 1936 in United States. Filed February 13, 1935 in Great Britain. Issued October 10, 1939 in United States.
- Crawford, L. S., Cheng, V. H. L., and Menon, P. K. 1999. Synthesis of flight vehicle guidance and control laws using genetic search methods. *Proceedings of 1999 Conference on Guidance, Navigation, and Control.* Reston, VA: American Institute of Aeronautics and Astronautics. Paper AIAA-99-4153.
- Dewell, Larry D. and Menon, P. K. 1999. Low-thrust orbit transfer optimization using genetic search. *Proceedings of* 1999 Conference on Guidance, Navigation, and Control. Reston, VA: American Institute of Aeronautics and Astronautics. Paper AIAA-99-4151.
- 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. 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., Bennett III, Forrest H, Andre, David, and Keane, Martin A. 1999. *Genetic Programming III:* Darwinian Invention and Problem Solving. San Francisco, CA: Morgan Kaufmann.

- Koza, John R., Bennett III, Forrest H, Andre, David, Keane, Martin A., and Brave, Scott. 1999. *Genetic Programming III Videotape: Human-Competitive Machine Intelligence*. San Francisco, CA: Morgan Kaufmann.
- Koza, John R., Keane, Martin A., Yu, Jessen, Bennett, Forrest H III, and Mydlowec, William. 2000. Automatic creation of human-competitive programs and controllers by means of genetic programming. *Genetic Programming* and Evolvable Machines. 1 (1 -2) 121 - 164.
- Koza, John R., Keane, Martin A., Yu, Jessen, Mydlowec, William, and Bennett, Forrest H III. 2000. Automatic synthesis of both the topology and parameters for a controller for a three-lag plant with a five-second delay using genetic programming. In Cagnoni, Stafano et al. (editors). *Real-World Applications of Evolutionary Computing. EvoWorkshops 2000. EvoIASP, Evo SCONDI, EvoTel, EvoSTIM, EvoRob, and EvoFlight, Edinburgh, Scotland, UK, April 2000, Proceedings.* Lecture Notes in Computer Science. Volume 1803. Berlin, Germany: Springer-Verlag. Pages 168 - 177.
- Koza, John R., and Rice, James P. 1992. *Genetic Programming: The Movie.* Cambridge, MA: MIT Press.
- Man, K. F., Tang, K. S., Kwong, S., and Halang, W. A. 1997. Genetic Algorithms for Control and Signal Processing. London: Springer-Verlag.
- Man, K. F., Tang, K. S., Kwong, S., and Halang, W. A. 1999. Genetic Algorithms: Concepts and Designs. London: Springer-Verlag.
- Menon, P. K., Yousefpor, M., Lam, T., and Steinberg, M. L. 1995. Nonlinear flight control system synthesis using genetic programming. *Proceedings of 1995 Conference on Guidance, Navigation, and Control.* Reston, VA: American Institute of Aeronautics and Astronautics. Pages 461 - 470.
- 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, University of California. Berkeley, CA. March 1994.
- 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.
- Sweriduk, G. D., Menon, P. K., and Steinberg, M. L. 1998. Robust command augmentation system design using genetic search methods. *Proceedings of 1998 Conference* on Guidance, Navigation, and Control. Reston, VA: American Institute of Aeronautics and Astronautics. Pages 286 - 294.
- Sweriduk, G. D., Menon, P. K., and Steinberg, M. L. 1999. Design of a pilot-activated recovery system using genetic search methods. *Proceedings of 1998 Conference on Guidance, Navigation, and Control.* Reston, VA: American Institute of Aeronautics and Astronautics.
- Ziegler, J. G. and Nichols, N. B. 1942. Optimum settings for automatic controllers. *Transactions of ASME*. (64) 759-768.