Evolution of a Controller with a Free Variable using Genetic Programming

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

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

Martin A. Keane
Econometrics Inc., Chicago, Illinois
makeane@ix.netcom.com

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

ABSTRACT

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 equation's coefficients. Previous work has demonstrated that genetic programming can automatically synthesize the design for a controller consisting of a topological arrangement of signal processing blocks (such as integrators, differentiators, leads, lags, gains, adders, inverters, and multipliers), where each block is further specified ("tuned") by a numerical component value, and where the evolved controller satisfies user-specified requirements. The question arises as to whether it is possible to use genetic programming to automatically create a "generalized" controller for an entire category of such controller design problems — instead of a single instance of the problem. This paper shows, for an illustrative problem, how genetic programming can be used to create the design for both the topology and tuning of controller, where the controller contains a free variable.

1 Introduction

Automatic controllers are ubiquitous in the real world (Astrom and Hagglund 1995; Boyd and Barratt 1991; Dorf and Bishop 1998). The output a controller is fed into the to-be-controlled system (conventionally called the *plant*). The purpose of a controller is to force, in a meritorious way, the plant's response to match a desired response (called the *reference signal* or *setpoint*). For example, the cruise control device in an car continuously adjusts the engine (the plant) based on the difference between the driver-specified speed (reference signal) and the car's actual speed (plant response).

The measure of merit for controllers typically incorporate a variety of interacting and conflicting considerations. Examples are the time required to force the plant response to reach a desired value, the degree of success in avoiding significant overshoot beyond the desired value, the time required for the plant to settle, the ability of the controller to operate robustly in the face of minor variations in the actual characteristics of the plant, the ability of the controller to operate robustly in the face of disturbances that may be introduced between the controller and the plant's input, the ability of the controller to operate robustly in the face of sensor noise that may be added to the plant output, the controller's compliance with various frequency domain constraints (e.g., bandwidth), and the controller's compliance with constraints on the magnitude of the control variable or the plant's internal state variables.

Controllers are often represented by block diagrams. A block diagram is a graphical structure containing (1) directed lines representing the (one-directional) flow of time-domain signals within the controller, (2) time-domain signal processing blocks (e.g., integrator, differentiator, lead, lag, gain, adder, inverter, and multiplier), (3) external input points from which the controller receives signals (e.g., the reference signal and the plant output that is externally fed back from the plant to the controller), and (4) output point(s) for the controller, conventionally called the *control variable*). Some signal processing blocks have multiple inputs (e.g., an adder), but they all have exactly one output. Because of this restriction, block diagrams for controllers usually contain *takeoff points* that enable the output of a block to be disseminated to more than one other point in the block diagram. Many blocks used in controllers (e.g., gain, lead, lag, delay) possess numerical parameters. The determination of these values is called *tuning*. Block diagrams sometimes also contain internal feedback (internal loops in the controller) or feedback of the controller output directly into the controller.

The design (synthesis) of a controller entails specification of both the topology and parameter values (tuning) for the block diagram of the controller such that the controller satisfies user-specified high-level design requirements. Specifically, the design process for a controller entails making decisions concerning the total number of signal processing blocks to be employed in the controller, the type of each block (e.g., integrator, differentiator, lead, lag, gain, adder, inverter, and multiplier), the interconnections between the inputs and outputs of each signal processing block and between the controller's external input and external output points, and the values of all required numerical parameters for the signal processing blocks.

Genetic programming has recently been used to create a block diagram for satisfactory controller for a particular two-lag plant and a particular three-lag plant (Koza, Keane, Yu, Bennett, and Mydlowec 2000). Both of these genetically evolved controllers outperformed the controllers designed by experts in the field of control using the criteria originally specified by the experts. However, both of these controllers were for plants having lags with a particular value of the time constant, τ.

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 question arises as to whether it is possible to evolve a controller for an *entire category* of plants — instead of one particular plant. In other words, is it possible to design a "generalized" controller containing one or more free variables. The fact that genetic programming permits the evolution of mathematical expressions containing free variables suggests that this might be possible. This paper shows, for an illustrative problem, that such a "generalized" controller can indeed be automatically created by means of genetic programming, that the genetically evolved controller has reasonable and interpretable characteristics, and that the genetically evolved controller performs better than the textbook human-designed controller for the illustrative problem.

Section 2 shows how genetic programming can automatically synthesize the design of a controller. Section 3 itemizes the preparatory steps for applying genetic programming to an illustrative problem with a free variable. Section 4 presents results.

2 Genetic Programming and Control

Genetic programming is commonly used in several different ways. As one example, genetic programming is often used as an automatic method for creating a program tree to solve a problem (Koza 1992). The individual programs that are evolved by genetic programming are typically multi-branch programs consisting of result-producing branches, automatically defined functions (subroutines), and other types of branches (e.g., iterations, loops, recursions). In this approach, the program tree is simply executed. The result of the execution may be a set of returned values, a set of side effects on some other entity (e.g., an external entity such as a robot or an internal entity such as computer memory), or a combination of returned values and side effects. In this approach, the functions in the program are sequentially executed, in time, in accordance with a specified order of evaluation such that the result of executing one function is available at the time when the next function is to be executed. Early work on the problem of automatically creating controllers used this conventional approach to genetic programming. This work includes, but is not limited to, Andersson, Svensson, Nordin, and Nordin, and Mats 1999; Banzhaf, Nordin, Keller, and Olmer 1997; Whitley, Gruau, and Preatt 1995; and Koza 1992.

As a second example, genetic programming is often also used to automatically create program trees which can be used in conjunction with a developmental process to design complex structures, such as neural networks (Gruau 1992) and analog electrical circuits (Koza, Bennett, Andre, and Keane 1996; Koza, Bennett, Andre, and Keane 1999; Koza, Bennett, Andre, Keane, and Brave 1999). In this approach, the program tree is interpreted as a set of instructions for constructing the desired structure. The construction process is implemented by applying the functions of a program tree to an embryonic structure so as to develop the embryo into a fully developed structure. As in the first approach, the functions in the program are executed separately, in time, in accordance with a specified order of evaluation.

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. Because of these continuous-time interactions, the sequential order of evaluation inherent in the two above approaches to genetic programming not a suitable. In the past, both genetic algorithms and genetic programming have been used for synthesizing controllers having mutually interacting continuous-time variables and continuous-time signal processing blocks (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). In addition, the PADO system and neural programming (Teller 1996a, 1996b; Teller and Veloso 1995a, 1995b, 1995c, 1995d, 1995e, 1996) provide a graphical program structure representation for such multiple independent processes. Also, Marenbach, Bettenhausen, and Freyer (1996) used automatically defined functions to represent internal feedback in a system (used for a system identification problem) where the overall multi-branch program represented a continuous-time system with continuously interacting processes. The essential features of their approach has been subsequently referred to as "multiple interacting programs" (Angeline 1997, 1998a, 1998b; Angeline and Fogel 1997).

In this paper, a program tree will constitute a description of the block diagram of a controller. The block diagram consists of time-domain signal processing functions linked by directed lines representing the flow of information. There is no order of evaluation of the functions and terminals of the program tree. Instead, the signal processing blocks of the controller and the plant interact with one another other as part of a closed system in the manner specified by the topology of the block diagram.

Figure 1 is a block diagram for an illustrative system containing a controller and a plant. The directed lines in a block diagram represent continuous-time 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 the a particular result (the result of the subtraction 510) to three places in the block diagram. The system in this figure is called a “closed loop” system because there is external feedback of the plant output back to the controller.

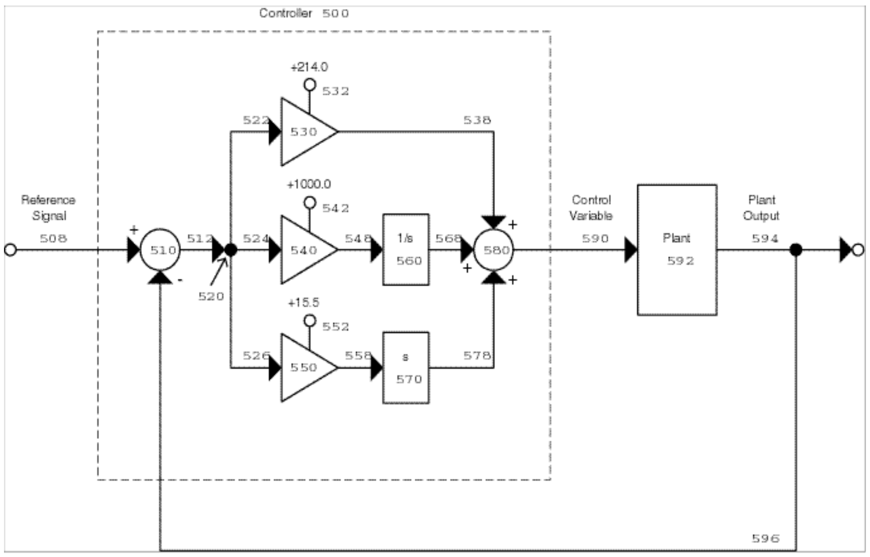

Figure 1 Block diagram of a plant and a PID controller.

Since this controller's output (control variable 590) is the sum of a proportional (P) term (gain block 530, with an amplification factor of 214.0), an integrating (I) term (integrator 560 preceded by the gain block 540, with an amplification factor of 1,000.0), and a differentiating (D) term (derivative block 570 preceded by the gain block 550, with an amplification factor of 15.5), this controller is called a PID controller.

Figure 2 presents the block diagram for the PID controller of figure 1 as a program tree. The internal points of this tree represent the signal processing blocks contained in the block diagram of figure 1 (i.e., derivative, integrator, gain, subtraction, addition) and other functions. The external points represent numerical constants and time-domain signals, such as the reference signal and plant output (and other terminals). Notice that automatically defined function `ADF0` (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 disseminate the result of this subtraction and correspond to the takeoff point 520 of

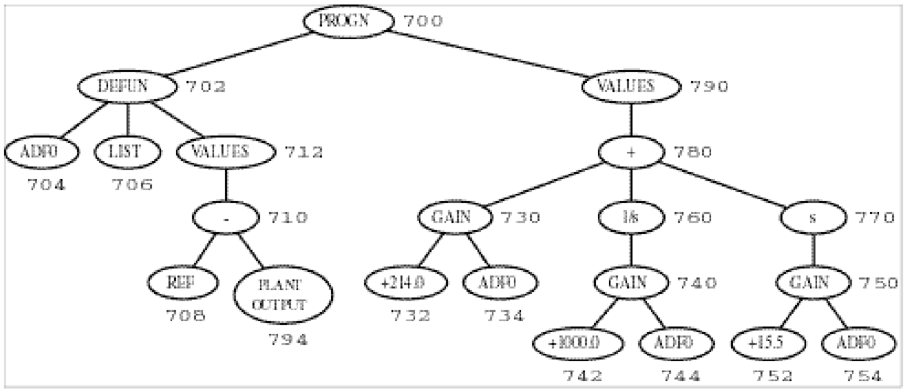

Figure 2 Program tree representation of the PID controller of figure 1. The automatically defined function `ADF0` (left) subtracts the plant output from the reference signal and makes this difference available at three points in the result-producing branch (right).

figure 1.

In the style of ordinary computer programming, a reference to an automatically defined function (subroutine) such as `ADF0` from inside the function definition for `ADF0` would be considered to be a recursive reference. However, in the context of control systems, an automatically defined function that references itself represents a loop in the controller's block diagram (i.e., internal feedback inside the controller).

3 Illustrative Problem

The problem is to create both the topology and parameter values for a robust "generalized" controller for a three-lag plant, where the controller contains a free variable representing the plant's time constant, τ. This free variable, τ, is allowed to range over two orders of magnitude. The transfer function of the three-lag plant is

$$G(s) = \frac{K}{(1+\tau s)^3}$$

The controller is to be designed so that plant output reaches the level of the reference signal so as to minimize the integral of the time-weighted error (ITAE), such that the overshoot in response to a step input of the reference signal (setpoint) is less than 2%, and such that the controller is robust in the face of variation in the plant's internal gain, *K* (tested by values of 1.0 and 2.0). The input to the plant is limited to the range between -40 and +40 volts.

The techniques in Astrom and Hagglund 1995 yield a credible controller for this problem.

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

An arithmetic-performing subtree containing perturbable numerical terminals, arithmetic operations, and a free variable representing the plant's time constant, τ, is used to establish the numerical parameter value for each signal processing block possessing a parameter.

The terminal set, T_{aps}, for such arithmetic-performing subtrees is

$T_{aps} = \{\Re, \tau\}$,

where $\Re$ denotes perturbable numerical values in the range from -3.0 and +3.0. In the initial random generation of a run, each perturbable numerical value is set, individually and separately, to a random value in this range. In later generations, this perturbable numerical value is perturbed by a Gaussian probability distribution (with standard deviation of 1.0). A constrained syntactic structure maintains a 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 = {REFERENCE_SIGNAL, CONTROLLER_OUTPUT, PLANT_OUTPUT,
CONSTANT_0}.

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 for details.

3.3 Function Set

The function set, F_{aps}, for the arithmetic-performing subtrees is

F_{aps} = {ADD_NUMERIC, SUB_NUMERIC, MUL_NUMERIC, DIV_NUMERIC}.

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 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`, `ADF0`, `ADF1`, `ADF2`,
`ADF3`, `ADF4`}.

The definition of the above signal processing functions is suggested by their names. See Koza, Keane, Yu, Bennett, and Mydlowec 2000 for details. `ADF0`, …, `ADF4` denote automatically defined functions added during the run by the architecture-altering operations.

3.4 Fitness Measure

Genetic programming conducts 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 a mathematical implementation of the high-level requirements of the problem and is couched in terms of "what needs to be done" — not "how to do it." Construction of the fitness measure requires translating the high-level requirements of the problem into a mathematically precise computation. The fitness measure may incorporate any measurable, observable, or calculable behavior or characteristic or combination of behaviors or characteristics. The fitness measure for most problems of controller design is multi-objective in the sense that there are several different (generally conflicting and interacting) requirements.

The fitness of each individual is determined by using the program tree (i.e., the result-producing branch and any automatically defined functions that may have been created during the run by the architecture-altering operations) to generate an interconnected sequence of signal processing blocks — that is, a block diagram for the controller. We use our modified version of the SPICE simulator (Quarles, Newton, Pederson, and Sangiovanni-Vincentelli 1994) from the University of California at Berkeley to simulate the continuous-time behavior of each controller. The various signal processing functions in the controller are each represented by means of electrical subcircuits (or mathematical expressions that can be handled by SPICE). A SPICE netlist (itemizing the subcircuits and the topological connections between them) is generated from the block diagram. The netlist is wrapped inside an appropriate set of SPICE commands and the overall behavior of the controller is then simulated. See Koza, Keane, Yu, Bennett, and Mydlowec 2000 and Koza, Bennett, Andre, and Keane 1999 for details.

The fitness of each controller in the population is measured by means of 42 separate invocations of the SPICE simulator. This 42-part fitness measure consists of

(1) 40 time-domain-based elements based on a modified integral of time-weighted absolute error (ITAE) measuring how quickly the controller causes the plant to reach the reference signal, the robustness of the controller in face of significant variations in the plant's internal gain, K, the success of the controller in avoiding overshoot, and the effect of the externally supplied value of the time constant of τ,

(2) one frequency-domain-based element measuring bandwidth, and

(3) one frequency-domain-based element measuring the effect of sensor noise.

The fitness of an individual controller is the sum (i.e., linear combination) of the detrimental contributions of these 42 elements of the fitness measure. The smaller the sum, the better.

The first 40 elements of the 42-part fitness measure together represent

- five different externally supplied values for the plant's time constant τ,
- in conjunction with two choices of values for the plant's internal gain, K,

• in conjunction with two choices of values for the height of the reference signal, and

• in conjunction with two choices of values of the variation in time constant τ, around its externally supplied value.

The five values of the externally supplied values for the plant's time constant τ, span a range of two orders of magnitude. They are 0.1, 0.3, 1.0, 3.0, and 10.0. The five externally supplied values of the plant's time constant τ, force generalization in the sense that the to-be-evolved controller becomes a function of the externally supplied value of the plant's time constant τ (the free variable).

The two values of the plant's internal gain, K, are1.0 and 2.0. These two values of the plant's internal gain, K, are used in order to ascertain robustness in the face of variation in K.

The reference signal is step function that rises from 0 to a specified height at t = 100 milliseconds. The two values for the height are 1 volt and 1 microvolts. The two values for the height of the step function are used to deal with the non-linearity caused by the limiter.

For each of these first 40 elements of the 42-part fitness measure, a transient analysis is performed in the time domain using the SPICE simulator. The contribution to fitness for each of these 40 elements of the fitness measure is based on the integral of time-weighted absolute error

$$\frac{\int_{t=0}^{10\tau} t\left|e(t)\right| A(e(t)) B dt}{\tau^2}.$$

Here $e(t)$ is the difference (error) at time t between the plant output and the reference signal. The integral is approximated from $t = 0$ to $t = 10\tau$, where τ is the externally supplied value of the time constant, τ. We multiply each value of $e(t)$ in the body of the integral by the reciprocal of the amplitude of the reference signals (so that both reference signals are equally influential). Specifically, B multiplies the difference $e(t)$ associated with the 1-volt step function by 1 and multiplies the difference $e(t)$ associated with the 1-microvolt step function by 10^6. In addition, we multiply each value of $e(t)$ in the body of the integral by an additional weight, A, that varies depending on $e(t)$ and that heavily penalizes non-compliant amounts of overshoot. The function A weights all variations below the reference signal and all variations up to 2% above the reference signal by a factor of 1.0, but A penalizes overshoots above 2% by a factor 10.0. Finally, we divide each value of the integral by τ^2 so as to equalize the influence of each of the five externally supplied values of τ.

The 41st element of the 42-part fitness measure is designed to constrain the frequency of the control variable so as to avoid extreme high frequencies in the demands placed upon the plant. This term reflects an unspoken constraint that is typically observed in real-world systems in order to prevent damage to delicate components of plants. This element of the fitness measure is based on 121 fitness cases representing 121 frequencies. Specifically, SPICE is instructed to perform an AC sweep of the reference signal 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 less than 6 dB is acceptable for frequencies between 0.01 Hz and 60 Hz and a gain of less than -80 dB is acceptable for frequencies between 60 Hz and 10,000 Hz. For each of the 121 frequencies, the contribution to fitness is 0 if the gain is acceptable, but 1,000/121 otherwise.

The 42nd element of the fitness measure is based on 121 fitness cases representing 121 frequencies. Specifically, SPICE is instructed to perform an AC sweep of the signal resulting by adding the plant response to a noise source ranging 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 less than 6 dB is acceptable for frequencies between 0.01 Hz and 60 Hz and a gain of less than -80 dB is acceptable for frequencies between 60 Hz and 10,000 Hz. For each of the 121 frequencies, the contribution to fitness is 0 if the gain is acceptable but 1,000/121 otherwise.

A controller that cannot be simulated by SPICE is assigned a fitness of 10^8.

3.5 Control Parameters

The population size, *M*, was 500,000. 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 for the genetic operations were the same as in Koza, Keane, Yu, Bennett, and Mydlowec 2000**.** The other parameters are the same default values that we have used previously on a broad range of problems (Koza, Bennett, Andre, Keane 1999).

3.6 Termination

The run was manually monitored and manually terminated when the fitness of the 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.

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 with 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 $Q = 500$ at each of $D = 1{,}000$ demes. Two processors are housed in each of the 500 physical boxes of the system. As each processor (asynchronously) completes a generation, four groups of emigrants from each subpopulation (selected probabilistically based on fitness) are dispatched to each of four toroidally adjacent processors. The migration rate is 2% (but 10% if the toroidally adjacent node is in the same physical box).

4 Results

The best-of-generation circuit from generation 0 has a fitness of 3,448.8. The best-of-run controller (figure 3) appears in generation 42. It has an overall fitness of 38.72. Notice that the free variable, τ, appears in five blocks of this controller.

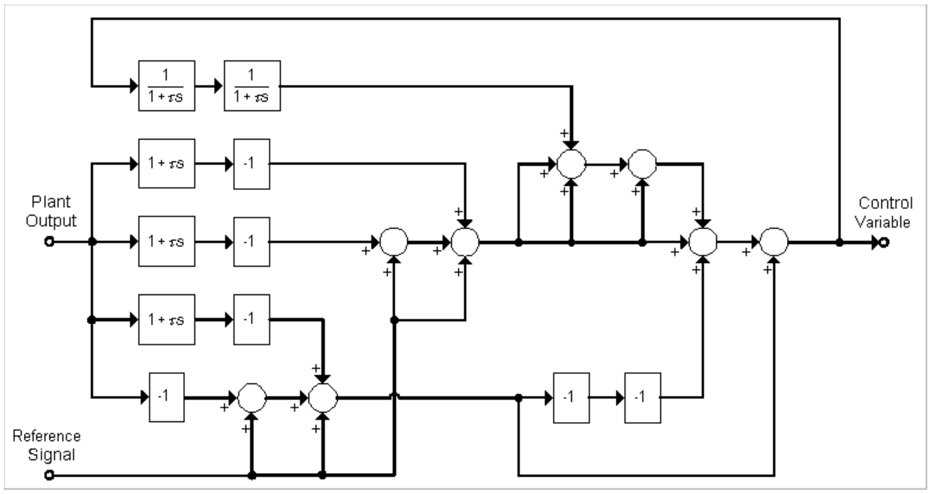

Figure 3 Block diagram of best-of-run genetically evolved controller from generation 42.

After simplification, it can be seen that the best-of-run controller from generation 42 has recognizable characteristics that are interpretable in the context of previously patented inventions or previously known control engineering techniques. The genetically evolved controller has the topology of a "PID-D2 controller" (i.e., proportional integrative, derivative, and second derivative), followed by a first-order lag block.. The PID controller was patented in 1939 by Albert Callender and Allan Stevenson of Imperial Chemical Limited of Northwich, England (Callender and Stevenson 1939) and the use of the second derivative was patented in 1942 by Harry Jones of the Brown Instrument Company of Philadelphia (Jones 1942). Also, the genetically evolved controller employs the technique called "setpoint weighting" in which the reference signal is weighted before the plant output is subtracted from it.

The control signal, $U(s)$, for this controller is given by the transfer function

$$U(s) = \frac{K_p(bR(s)-Y(s)) + K_i(R(s)-Y(s))/s + K_d(cR(s)-Y(s))s + K_{d2}(eR(s)-Y(s))s^2}{1+ns}$$

Here the four PID-D2 coefficients in the numerator are K_p = 17.0 for the proportional (P) term, $K_i = 6/\tau$ for the integrative (I) term (with the $1/s$ in the numerator), K_d = 16τ for the derivative (D) term (with the s), and $K_{d2} = 5\tau^2$ for the second derivative (D2) term (with the s^2). The coefficient for the setpoint weighting of the proportional term is $b = 0.706$. That is, the plant output, $Y(s)$, is subtracted from b multiplied by the reference signal, $R(s)$. Similarly, the coefficient for the setpoint weighting of the derivative term is $c = 0.375$. The coefficient for the setpoint weighting of the second derivative term is $e = 0.0$. The time-constant of the first-order lag (shown here in the denominator of the transfer function) is $n = 0.5\tau$.

Figure 4 compares the time-domain response of the best-of-run genetically evolved controller (triangles) from generation 42 and the Astrom and Hagglund controller (squares) with $K = 1$ and $\tau = 1$. This (and other comparisons below) are substantially similar for other values of K and τ. Figure 5 compares the time-domain response of the best-of-run controller (triangles) from generation 42 and the Astrom and Hagglund

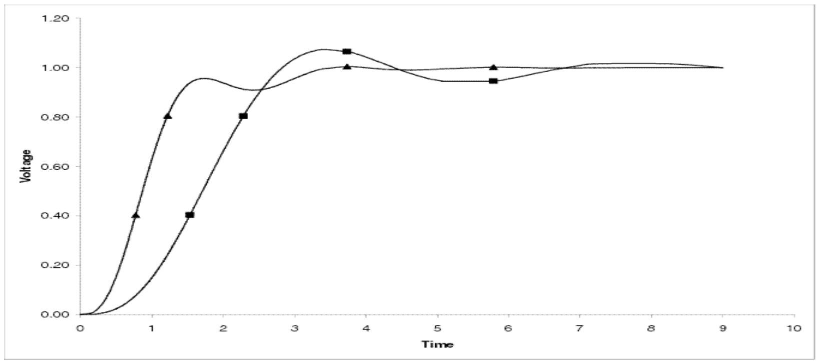

Figure 4 Comparison of the time-domain response of the best-of-run controller (triangles) from generation 42 and the Astrom and Hagglund controller (squares) with $K = 1$ and $\tau = 1$.

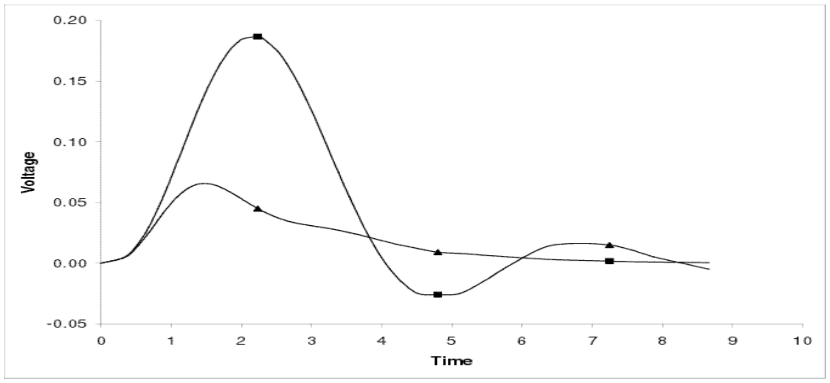

Figure 5 Comparison of the time-domain response of the genetically evolved controller (triangles) from generation 42 and the Astrom and Hagglund controller (squares) to a 1-volt disturbance signal with $K = 1$ and $\tau = 1$.

controller (squares) to a 1-volt disturbance signal with $K = 1$ and $\tau = 1$.

Table 1 compares the characteristics of the best-of-run genetically evolved controller (triangles) from generation 42 with those of the Astrom and Hagglund 1995 controller for $K = 1.0$, $\tau = 1.0$, and step size of 1.0.

Table 1 Comparison of characteristics for $K = 1.0$, $\tau = 1.0$, and step size of 1.0.

	Units	Genetically evolved controller	Astrom and Hagglund controller
ITAE for unit-step reference	volt sec^2	0.77	2.84
Rise time	seconds	1.43	2.51
Settling time	seconds	4.66	8.50
Disturbance sensitivity	volts/volts	0.07	0.19
IATE for unit-step disturbance	volt sec^2	0.42	1.35

Noise Attenuation Corner Frequency	Hz	0.72	0.34
Noise Attenuation Roll Off	dB/Decade	40	40
Maximum Sensitivity	volts/volts	2.09	2.11

The controller created by genetic programming is better than 3.69 times as effective as the Astrom and Hagglund 1995 controller as measured by the integral of the time-weighted absolute error, has only 57% of the rise time in response to the reference input, and has only 55% of the settling time.

4.1 Interpretation of the Results

Suppose $G(s)$ is the transfer function of a plant and we wanted to change all the time constants of this plant by the same amount, say τ. To do this, we would produce a plant with transfer function, $G_\tau(s) = G(\tau s)$. In this new transfer function $G_\tau(s)$, each coefficient of s would be multiplied by τ; each coefficient of s^2 would be multiplied by τ^2; and each coefficient of $1/s$ would be multiplied by $1/\tau$.

A human designer might approach this problem as follows: Call the plant with $\tau = 1$ the reference plant. Design a controller for this reference plant satisfying the design requirements and call the new controller, H, the reference controller. Call the controlled plant, J, the reference controlled plant. Let $H(s)$ be the transfer function of controller $H(s)$, and let $J(s)$ be the transfer function of the controlled plant. Now, to handle the time scaling, let $H_\tau(s) = H(\tau s)$ be the controller design for the time-scaled plant. This will produce an entire controlled plant, $J_\tau(s)$, scaled by the same factor τ; $J_\tau(\mathrm{s}) = J(\tau{*}s)$. Effectively all the behavior of the time-scaled controlled plant is that of the reference controlled plant with the time scale changed.

In fact, we did exactly this in order to produce a PID controller with set point weighting for comparison purposes above (using the design rules using $M_s = 2$ on page 225 of Astrom and Hagglund 1995).

Referring now to genetically evolved controller shown above, notice that we could have written the following four coefficients in the control signal, $U(s)$, as follows:

$K_i/s = 6/\tau s$

$K_d s = 16\tau \mathrm{s}^2$

$K_{d2} s^2 = 5\tau^2 s^2$

$ns = 0.5\tau s$

This is, in fact, the very same change in time scale that a human designed would have used.

Notice that we did not provide genetic programming with any of the above knowledge and insight about how a human designer might have proceeded on this problem. We did not bias genetic programming toward multiplying s by τ; toward multiplying s^2 by τ^2; and toward multiplying $1/s$ by $1/\tau$. We simply provided the free variable, τ, as a terminal that genetic programming could use as it saw fit.

4.2 Computer Time

Most of the computer time was consumed by the fitness evaluation of candidate individuals in the population. The fitness evaluation (involving 40 time-consuming time-domain SPICE simulations and two relatively fast frequency-domain SPICE simulations) averaged about 3.92 seconds per individual (using a 350 MHz Pentium II processor). The best-of-run individual from generation 42 was produced after evaluating 2.15×10^7 individuals (500,000 times 43). This required 23.43 hours (84,360 seconds) on our 1,000-node parallel computer system — that is, the expenditure of 2.95×10^{16} computer cycles (about 30 peta-cycles of computer time).

5 Conclusion

This paper demonstrated, for an illustrative problem involving a three-lag plant, that genetic programming can be used to create the design for both the topology and parameter values (tuning) of signal processing functions for a controller, where the controller contains a free variable. The genetically evolved controller outperformed the textbook human-designed 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.

Angeline, Peter J. 1997. An alternative to indexed memory for evolving programs with explicit state representations. 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 423 - 430.

Angeline, Peter J. 1998. Multiple interacting programs: A representation for evolving complex behaviors. *Cybernetics and Systems*. 29 (8) 779 - 806.

Angeline, Peter J. 1998. Evolving predictors for chaotic time series. In Rogers, S., Fogel, D., Bezdek, J., and Bosacchi, B. (editors). *Proceedings of SPIE (Volume 3390): Application and Science of Computational Intelligence,* Bellingham, WA: SPIE - The International Society for Optical Engineering. Pages 170-180.

Angeline, Peter J. and Fogel, David B. 1997. An evolutionary program for the identification of dynamical systems. In Rogers, S. (editor). *Proceedings of SPIE (Volume 3077): Application and Science of Artificial Neural Networks III*. Bellingham, WA: SPIE - The International Society for Optical Engineering. Pages 409-417.

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.

Boyd, S. P. and Barratt, C. H. 1991. *Linear Controller Design: Limits of Performance*. Englewood Cliffs, NJ: Prentice Hall.

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.

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

Gruau, Frederic. 1992. Genetic synthesis of Boolean neural networks with a cell rewriting developmental process. In Schaffer, J. D. and Whitley, Darrell (editors). *Proceedings of the Workshop on Combinations of Genetic Algorithms and Neural Networks 1992*. Los Alamitos, CA: The IEEE Computer Society Press.

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

Jones, Harry S. 1942. *Control Apparatus*. United States Patent 2,282,726. Filed October 25, 1939. Issued May 12, 1942.

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., Bennett III, Forrest H, Andre, David, and Keane, Martin A. 1996. Automated design of both the topology and sizing of analog electrical circuits using genetic programming. In Gero, John S. and Sudweeks, Fay (editors). *Artificial Intelligence in Design '96*. Dordrecht: Kluwer Academic Publishers. 151-170.

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.

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.

Marenbach, Peter, Bettenhausen, Kurt D., and Freyer, Stephan. 1996. Signal path oriented approach for generation of dynamic process models. In Koza, John R., Goldberg, David E., Fogel, David B., and Riolo, Rick L. (editors). *Genetic Programming 1996: Proceedings of the First Annual Conference, July 28-31, 1996, Stanford University*. Cambridge, MA: MIT Press. Pages 327 - 332.

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.

Teller, Astro. 1996a. *Evolving Programmers: SMART Mutation*. Technical Report CMU-CS-96. Computer Science Department, Carnegie Mellon University.

Teller, Astro. 1996b. Evolving programmers: The co-evolution of intelligent recombination operators. In Angeline, Peter J. and Kinnear, Kenneth E. Jr. (editors). *Advances in Genetic Programming 2*. Cambridge, MA: The MIT Press.

Teller, Astro, and Veloso, Manuela. 1995a. *Learning Tree Structured Algorithms for Orchestration into an Object Recognition System*. Technical Report CMU-CS-95-101. Computer Science Department, Carnegie Mellon University.

Teller, Astro, and Veloso, Manuela. 1995b. Program evolution for data mining. In Louis, Sushil (editor). Special Issue on Genetic Algorithms and Knowledge Bases. *The International Journal of Expert Systems*. JAI Press. (3) 216 - 236.

Teller, Astro, and Veloso, Manuela. 1995c. A controlled experiment: evolution for learning difficult problems. *Proceedings of Seventh Portuguese Conference on Artificial Intelligence*. Springer-Verlag. Pages 165-76.

Teller, Astro, and Veloso, Manuela. 1995d. Algorithm Evolution for Face Recognition: What Makes a Picture Difficult? *Proceedings of the IEEE International Conference on Evolutionary Computation*. IEEE Press.

Teller, Astro, and Veloso, Manuela. 1995e. Language Representation Progression in PADO. *Proceedings of AAAI Fall Symposium on Artificial Intelligence*. Menlo Park, CA: AAAI Press.

Teller, Astro and Veloso, Manuela. 1996. PADO: A new learning architecture for object recognition. In Ikeuchi, Katsushi and Veloso, Manuela (editors). *Symbolic Visual Learning*. Oxford University Press. Pages 81-116.

Whitley, Darrell, Gruau, Frederic, and Preatt, Larry. 1995. Cellular encoding applied to neurocontrol. In Eshelman, Larry J. (editor). *Proceedings of the Sixth International Conference on Genetic Algorithms*. San Francisco, CA: Morgan Kaufmann. Pages 460 - 467.