Genetic Programming and Evolvable Machines, 1, 121–164 (2000)
© 2000 Kluwer Academic Publishers. Manufactured in The Netherlands.

# Automatic Creation of Human-Competitive Programs and Controllers by Means of Genetic Programming

JOHN R. KOZA koza@stanford.edu, http://www.smi.stanford.edu/people/koza
*Department of Electrical Engineering, School of Engineering, and Department of Medicine, School of Medicine, Stanford University, Stanford, California 94305*

MARTIN A. KEANE makeane@ix.netcom.com
*Econometrics Inc., 111 E. Wacker Dr., Chicago, Illinois 60601*

JESSEN YU jyu@cs.stanford.edu
*Genetic Programming Inc., Box 1669, Los Altos, California 94023*

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

WILLIAM MYDLOWEC myd@cs.stanford.edu
*Genetic Programming Inc., Box 1669, Los Altos, California 94023*

*Received September 14, 1999; Revised October 6, 1999*

**Abstract.** Genetic programming is an automatic method for creating a computer program or other complex structure to solve a problem. This paper first reviews various instances where genetic programming has previously produced human-competitive results. It then presents new human-competitive results involving the automatic synthesis of the design of both the parameter values (i.e., tuning) and the topology of controllers for two illustrative problems. Both genetically evolved controllers are better than controllers designed and published by experts in the field of control using the criteria established by the experts. One of these two controllers infringes on a previously issued patent. Other evolved controllers duplicate the functionality of other previously patented controllers. The results in this paper, in conjunction with previous results, reinforce the prediction that genetic programming is on the threshold of routinely producing human-competitive results and that genetic programming can potentially be used as an "invention machine" to produce patentable new inventions.

**Keywords:** control, network synthesis, genetic programming, human-competitive results

## 1. Introduction

Turing recognized the possibility of employing evolution and natural selection to achieve machine intelligence as early as 1948. In his essay "Intelligent Machines," Turing [83] identified three approaches for creating intelligent computer programs. One approach was logic-driven while a second was knowledge-based. The third approach that Turing specifically identified in 1948 for achieving machine intelligence is

> ... the genetical or evolutionary search by which a combination of genes is looked for, the criterion being the survival value.

In his 1950 paper "Computing Machinery and Intelligence," Turing [84] described how evolution and natural selection might be used to automatically create an intelligent computer program.

> We cannot expect to find a good child-machine at the first attempt. One must experiment with teaching one such machine and see how well it learns. One can then try another and see if it is better or worse. There is an obvious connection between this process and evolution, by the identifications
>
> Structure of the child machine = Hereditary material
>
> Changes of the child machine = Mutations
>
> Natural selection = Judgment of the experimenter

The above features of Turing's third approach to machine intelligence are common to the various forms of evolutionary computation developed over the past four decades, including evolution strategies [60, 61], simulated evolution and evolutionary programming [29], genetic algorithms [31], genetic classifier systems [32], genetic programming [37, 38], and evolvable hardware [30].

Each of these approaches to evolutionary computation addresses the main goal of machine intelligence, which, to paraphrase Arthur Samuel [63], entails

> How can computers be made to do what needs to be done, without being told exactly how to do it?

More particularly, as Samuel [64] also stated,

> The aim [is] ... to get machines to exhibit behavior, which if done by humans, would be assumed to involve the use of intelligence.

What do we mean when we say that an automatically created solution to a problem is competitive with a human-produced result? We think it is fair to say that an automatically created result is competitive with one produced by human engineers, designers, mathematicians, or programmers if it satisfies any one (or more) of the following eight (or any other similarly stringent) criteria [43]:

(A) The result was patented as an invention in the past, is an improvement over a patented invention, or would qualify today as a patentable new invention.
(B) The result is equal to or better than a result that was accepted as a new scientific result at the time when it was published in a peer-reviewed scientific journal.

(C) The result is equal to or better than a result that was placed into a database or archive of results maintained by an internationally recognized panel of scientific experts.
(D) The result is publishable in its own right as a new scientific result—*independent* of the fact that the result was mechanically created.
(E) The result is equal to or better than the most recent human-created solution to a long-standing problem for which there has been a succession of increasingly better human-created solutions.
(F) The result is equal to or better than a result that was considered an achievement in its field at the time it was first discovered.
(G) The result solves a problem of indisputable difficulty in its field.
(H) The result holds its own or wins a regulated competition involving human contestants (in the form of either live human players or human-written computer programs).

Note that the above criteria require a result that is meritorious in light of some standard that is external the fields of machine learning, artificial intelligence, and evolutionary computation. Note also that all the criteria are couched in terms of production of a result, not use of a particular methodology.

Using the above criteria, there are now numerous instances where evolutionary computation has produced a result that is competitive with human performance. Since nature routinely uses evolution and natural selection to create designs for complex structures that are well-adapted to their environments, it is not surprising that many of these examples involve the design of complex structures. Examples of human-competitive results that have been achieved using evolutionary algorithms include (but are not limited to) the automated creation of a sorting network that is superior to the previously known human-created solution to the same problem [34], the automated creation of a checker player created by evolutionary programming [24] that plays checkers as well as "class A" human checker players, and the automated creation of a cellular automata rule that outperforms previous human-written and machine-produced algorithms for the same problem [35]*. Evolutionary methods have the advantage of not being encumbered by preconceptions that limit the search to familiar paths. There are at least two instances where evolutionary computation yielded an invention that was granted a patent, namely a design for a wire antenna created by a genetic algorithm [1] and a patent for the shape of an aircraft wing created by a genetic algorithm with variable-length strings [49].

Genetic programming is an automatic method for creating a computer program or other complex structure to solve a problem. Focusing on genetic programming, Table 1 shows 24 instances of results where genetic programming has produced results that are competitive with the products of human creativity and inventiveness. These examples involve quantum computing, the annual Robo Cup competition, computational molecular biology, cellular automata, sorting networks, the automatic synthesis of the design of analog electrical circuits, and the automatic synthesis of the design of controllers (in this paper). Each claim is accompanied by

* See also paper by Miller et al. in this issue.

*Table 1.* Twenty-four instances where genetic programming has produced human-competitive result

| | Claimed instance | Basis for claim | Reference | Infringed patent |
|---|---|---|---|---|
| 1 | Creation, using genetic programming, of a better-than-classical quantum algorithm for the Deutsch-Jozsa "early promise" problem | B, F | [65] | |
| 2 | Creation, using genetic programming, of a better-than-classical quantum algorithm for the Grover's database search problem | B, F | [66] | |
| 3 | Creation, using genetic programming, of a quantum algorithm for the depth-2 AND/OR query problem that is better than any previously published result | B, D | [67] | |
| 4 | Creation of soccer-playing program that ranked in the middle of the field of 34 human-written programs in the Robo Cup 1998 competition | H | [41] | |
| 5 | Creation of four different algorithms for the transmembrane segment identification problem for proteins | B, E | [43] | |
| 6 | Creation of a sorting network for seven items using only 16 steps | A, D | [55, 43] | |
| 7 | Rediscovery of the Campbell ladder topology for lowpass and highpass filters | A, F | [43] | [20] |
| 8 | Rediscovery of "*M*-derived half section" and "constant *K*" filter sections | A, F | [43] | [86] |
| 9 | Rediscovery of the Cauer (elliptic) topology for filters | A, F | [43] | [21, 22, 23] |
| 10 | Automatic decomposition of the problem of synthesizing a crossover filter | A, F | [43] | [86] |
| 11 | Rediscovery of a recognizable voltage gain stage and a Darlington emitter-follower section of an amplifier and other circuits | A, F | [43] | [26] |
| 12 | Synthesis of 60 and 96 decibel amplifiers | A, F | [43] | |
| 13 | Synthesis of analog computational circuits for squaring, cubing, square root, cube root, logarithm, and Gaussian functions | A, D, G | [43] | |
| 14 | Synthesis of a real-time analog circuit for time-optimal control of a robot | G | [43] | |
| 15 | Synthesis of an electronic thermometer | A, G | [43] | |
| 16 | Synthesis of a voltage reference circuit | A, G | [43] | |
| 17 | Creation of a cellular automata rule for the majority classification problem that is better than the Gaes-Kurdyumov-Levin (GKL) rule and all other known rules written by humans | D, E | [3] | |
| 18 | Creation of motifs that detect the D-E-A-D box family of proteins and the manganese superoxide dismutase family | C | [43] | |
| 19 | Synthesis of analog circuit equivalent to Philbrick circuit | A | [57, 44] | |

*Table 1.* (Continued)

| | Claimed instance | Basis for claim | Reference | Infringed patent |
|---|---|---|---|---|
| 20 | Synthesis of NAND circuit | A | [15] | |
| 21 | Synthesis of digital-to-analog converter (DAC) circuit | A | [15] | |
| 22 | Synthesis of analog-to-digital (ADC) circuit | A | [15] | |
| 23 | Synthesis of topology, sizing, placement, and routing of analog electrical circuits | G | [42] | |
| 24 | Synthesis of topology for a PID type of controller with an added second derivative | A, F | This paper | [19, 33] |

the particular criterion (from the list above) that establishes the basis for the claim of human competitiveness.

Fifteen of the 24 instances in Table 1 involve previously patented inventions. Six of these automatically created results infringe on previously issued patents (indicated in the table by a citation to both the original patent and the genetically produced result). One of the genetically evolved results improves on a previously issued patent. Nine of the genetically evolved results duplicate the functionality of previously patented inventions in novel ways. The fact that genetic programming can evolve entities that infringe on previously patented inventions, improve on previously patented inventions, or duplicate the functionality of previously patented inventions suggests that genetic programming can potentially be used as an "invention machine" to create new and useful patentable inventions.

Section 2 provides background on control systems. Section 3 discusses how genetic programming can be used to automatically synthesize (create) the design of controllers. Section 4 describes two illustrative problems. Section 5 describes the preparatory steps necessary to apply genetic programming to the two illustrative problems. Sections 6 and 7 present the results for the two illustrative problems. Section 8 is the conclusion.

## 2. Control systems

The purpose of a controller is to force, in a meritorious way, the actual response of a system (conventionally called the *plant*) to match a desired response (called the *reference signal*) [10, 17, 18, 56, 28].

Controllers (control systems) are ubiquitous. A thermostat is an example of a controller. The occupant of a chilly room may set a thermostat to request that the room temperature be raised to 70 degrees (the reference signal). The controller causes fuel to flow into the furnace so as to cause the furnace to heat the room to 70 degrees. As the temperature of the room rises, the controller continuously adjusts its behavior based on the difference between the reference signal (the desired temperature) and the room's current actual temperature (the plant response).

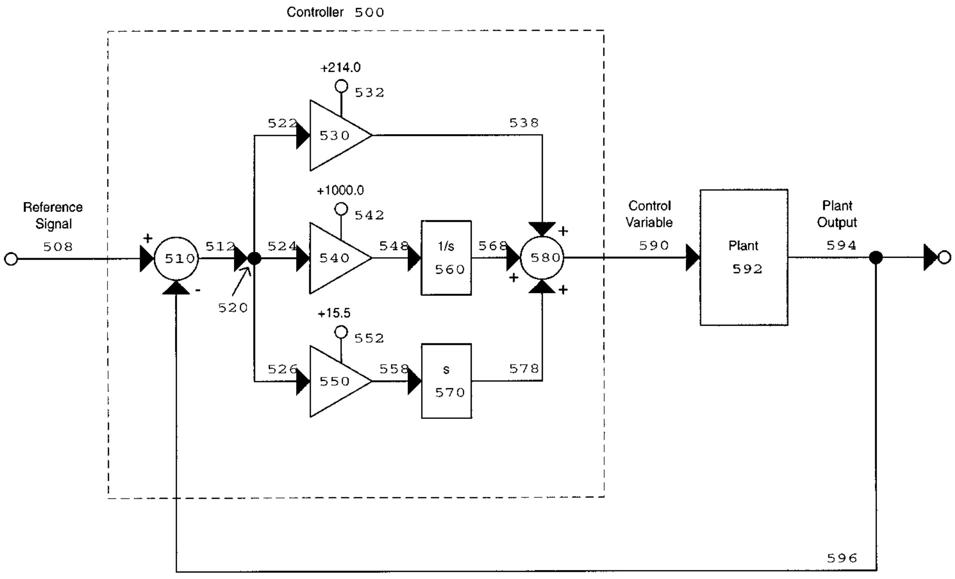

*Figure 1.* Block diagram of a plant and a PID controller composed of proportional, integrative, and derivative blocks. The plant's output is fed back to the controller where it is compared to the reference signal.

## *Illustrative control system*

Figure 1 is a block diagram of an illustrative control system containing a controller and a plant. 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). The system in this figure is called a "closed loop" system because there is external feedback of the plant output back to the controller. Controllers without such feedback are called "open loop" controllers. Such "open loop" controllers are considerably simpler (and generally less useful in real world applications) than "closed loop" controllers.

There are many different (and usually conflicting) measures of merit for controllers. For example, it is often considered desirable to minimize the time required to bring about the desired response of the plant. Meanwhile, it is often considered desirable to simultaneously avoid significantly overshooting the target value for the plant response. Furthermore, since all real world plants, controllers, and their sensors are imperfect, it is also desirable that a controller operate robustly in the face of variations in the actual characteristics of the plant, in the face of disturbances that may be added into the controller's output, and in the face of sensor noise that may be added to the plant output or reference signal. In addition, it is common to constrain the control variable to a particular prespecified range because of physical limitations. Also, the plant's internal state variables are sometimes constrained to a particular prespecified range. It is also desirable to be able to suppress high frequency noise in the reference signal, the control variable, and the actual plant response.

The underlying principles of control systems are broadly the same whether the system is mechanical, electrical, thermodynamic, hydraulic, biological, or economic and whether the variable of interest is temperature, velocity, voltage, water pressure, interest rates, heart rate, or humidity [28].

## *Block diagrams of controllers*

Block diagrams are a useful tool for representing the flow of information in controllers and systems containing controllers. Block diagrams contain signal processing function blocks, external input and output points, and directed lines.

A function block in a block diagram has one or more inputs, but only one output.

Lines in a block diagram represent time-domain signals. Lines are directional in that they represent the flow of information. The lines pointing toward a block represent signals coming into the block. The single line pointing away from a block represents the block's single output. Note that a line in a block diagram differs from a wire in an electrical circuit in that the line is directional and that a function block in a block diagram differs from an electrical component in a circuit in that there is only one output from a function block.

In a block diagram, an external input is represented by an external point with a directed line pointing away from that point. Similarly, an external output point is represented by an external point with a line pointing toward that point.

The topology of a controller entails specification of the total number of processing blocks to be employed in the controller, the type of each block, the connections between the inputs and output of each block in the controller and the external input and external output points of the controller. Many of the signal processing blocks used in controllers possess numerical parameter values.

Adders are conventionally represented by circles in block diagrams. Each input to an adder is labeled with a positive or negative sign (so adders may be used to perform both addition and subtraction). In Figure 1, adder 510 performs the function of subtracting the fed-back plant output 596 from the externally supplied reference signal 508.

Takeoff points (conventionally represented in block diagrams by a large dot) provide a way to disseminate a signal to more than one other function block in a block diagram. Takeoff point 520 receives signal 512 and disseminates this signal to function blocks 530, 540, and 550.

In the figure, the subtractor's output 522 is passed into a `GAIN` block 530. A `GAIN` block (shown in this figure as a triangle) multiplies (amplifies) its input by a specified constant amplification factor (i.e., the numerical constant 214.0). The amplified result 538 becomes the first of the three inputs to addition block 580.

The subtractor's output is also passed into `GAIN` block 540 (with a gain of 1,000.0) and the amplified result 548 is passed into `INTEGRATOR` block 560 (shown in the figure by a rectangle labeled $1/s$, where $s$ is the Laplace transform variable). The result 568 of this integration (with respect to time) becomes the second input to addition block 580. Similarly, the subtractor's output is passed into `GAIN` block 550 (with a gain of 15.5) and the amplified result 558 is passed into `DIFFERENTIATOR` block 570 (shown in the figure by a rectangle labeled $s$) and becomes the third input to addition block 580.

The controller of Figure 1 can be represented as a transfer function as follows.

$$G_c(s) = 214.0 + \frac{1000.0}{s} + 15.5s = \frac{214.0s + 1000.0 + 15.5s^2}{s}.$$

In this representation, the 214.0 is the proportional (P) element of the controller; the 1,000.0/s is the integrating (I) term; and the $15.5s$ is the differentiating (D) term. It is common to write the transfer function as a quotient of two polynomials in the Laplace transform variable (as is done on the right).

## *PID controllers*

Since the output (i.e., control variable 590) of the controller in Figure 1 is the sum of a proportional (P) term, an integrating (I) term, and a differentiating (D) term, this type of controller is called a PID controller. The PID controller was patented in 1939 by Albert Callender and Allan Stevenson of Imperial Chemical Limited of Northwich, England. The PID controller was a significant improvement over previous approaches to control. In discussing the problems of "hunting" and instability, Callender and Stevenson [19] state,

> If the compensating effect $V$ is applied in direct proportion to the magnitude of the deviation $\Theta$, over-compensation will result. To eliminate the consequent hunting and instability of the system, the compensating effect is additionally regulated in accordance with other characteristics of the deviation in order to bring the system back to the desired balanced condition as rapidly as possible. These characteristics include in particular the rate of deviation (which may be indicated mathematically by the time-derivative of the deviation) and also the summation or quantitative total change of the deviation over a given time (which may be indicated mathematically by the time-integral of the deviation).

Callender and Stevenson [19] state,

> A specific object of the invention is to provide a system which will produce a compensating effect governed by factors proportional to the total extent of the deviation, the rate of the deviation, and the summation of the deviation during a given period ...

Claim 1 of Callender and Stevenson [19] covers what is now called the PI controller,

> A system for the automatic control of a variable characteristic comprising means proportionally responsive to deviations of the characteristic from a desired value, compensating means for adjusting the value of the characteristic, and electrical means associated with and actuated by responsive variations in said responsive means, for operating the compensating means to correct such deviations in conformity with the sum of the extent of the deviation and the summation of the deviation.

Claim 3 of Callender and Stevenson [19] covers what is now called the PID controller,

> A system as set forth in claim 1 in which said operation is additionally controlled in conformity with the rate of such deviation.

PD, PI, and PID controllers are in widespread use in industry. As Astrom and Hagglund [10] noted,

> Several studies ... indicate the state of the art of industrial practice of control. The Japan Electric Measuring Instrument Manufacturing Association conducted a survey of the state of process control systems in 1989 ... According to the survey, more than 90% of the control loops were of the PID type.

However, even though PID controllers are in widespread use in industry, the need for better controllers is widely recognized. As Astrom and Hagglund [10] observed,

> ... audits of paper mills in Canada [show] that a typical mill has more than 2,000 control loops and that 97% use PI control. Only 20% of the control loops were found to work well.

It is generally recognized by leading practitioners in the field of control that there are significant limitations on analytical techniques in designing controllers. As Boyd and Barratt stated in *Linear Controller Design: Limits of Performance* [17],

> The challenge for controller design is to productively use the enormous computing power available. Many current methods of computer-aided controller design simply automate procedures developed in the 1930's through the 1950's, for example, plotting root loci or Bode plots, Even the 'modern' state-space and frequency-domain methods (which require the solution of algebraic Riccati equations) greatly underutilize available computing power.

## 3. Genetic programming and control

Design is a major activity of practicing engineers. Engineers are often called upon to design complex structures (such as circuits or controllers) that satisfy certain prespecified high-level design goals. The creation of a design for a complex structure typically involves intricate tradeoffs between competing considerations. Design is ordinarily thought to require human intelligence.

The design (synthesis) of a controller requires specification of both parameter values (tuning) and the topology such that the controller satisfies certain user-specified high-level design requirements. Specifically, 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 inputs and outputs of each block in the controller and the external input and external output points of the controller, and the values of all numerical parameters for the blocks.

The design process for controllers today is generally channeled along lines established by existing analytical techniques (notably those that lead to a PID-type controller).

It would be desirable to have an automatic system for synthesizing (creating) the design of a controller that was open-ended in the sense that it did not require the human user to prespecify the topology of the controller (whether PID or other), but, instead, automatically produced both the overall topology and parameter values directly from a high-level statement of the requirements of the controller. However, there is no preexisting general-purpose analytic method for automatically creating a controller for arbitrary linear and nonlinear plants that can simultaneously optimize prespecified performance metrics (such as minimizing the time required to bring the plant output to the desired value as measured by, say, the integral of the time-weighted absolute error), satisfy time-domain constraints (involving, say, overshoot and disturbance rejection), satisfy frequency domain constraints (e.g., bandwidth), and satisfy additional constraints, such as constraints on the magnitude of the control variable and the plant's internal state variables.

### *Background on genetic programming*

Genetic programming is often used as an automatic method for creating a computer program to solve a problem. Genetic programming is an extension of the genetic algorithm [31]. Genetic programming is described in detail in [38, 48, 39, 40, 43, 45]. Recent work on genetic programming is described in [12, 50, 62, 36, 9, 68, 47, 46, 41, 11, 14, and 58].

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, gene deletion, and certain mechanisms of developmental biology 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 functions and terminals of the problem.
(2) Iteratively perform the following substeps (referred to herein as 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 fitness 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 available repertoire of such operations and create one new offspring program for the new population by applying the selected architecture-altering 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.

The individual programs that are evolved by genetic programming are typically multibranch programs consisting of one or more result-producing branches and zero, one, or more automatically defined functions (subroutines). The *architecture* of such a multibranch program involves

(1) the total number of automatically defined functions,
(2) the number of arguments (if any) possessed by each automatically defined function, and
(3) if there is more than one automatically defined function in a program, the nature of the hierarchical references (including recursive references), if any, allowed among the automatically defined functions.

There are two ways of determining the architecture for a program that is to be evolved using genetic programming.

(1) The human user may prespecify the architecture of the overall program as part of the preparatory steps required for launching the run of genetic programming.
(2) Architecture-altering operations may be used during the run to automatically create the architecture of the program.

Architecture-altering operations 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. Although not used in this paper, certain additional architecture-altering operations enable genetic programming to automatically determine whether and how to use internal memory, iterations, and recursion in evolved programs [43].

### *Previous work involving evolutionary computation and control*

There has been extensive previous work on the problem of automating various aspects of the design of controllers using evolutionary computation in general and genetic programming in particular. Genetic programming has been previously applied to certain simple control problems, including, but not limited to, discrete-time problems where the evolved program receives the system's current state as input, performs an arithmetic or conditional calculation on the inputs, and computes a value for a control variable. These discrete-time problems have included cart centering, broom balancing, backing a tractor-trailer truck to a loading dock, controlling the food foraging strategy of a lizard [38], navigating a robot with a nonzero turning radius to a destination point [43], evolving an analog electrical circuit for a robotic controller navigating a robot with a nonzero turning radius to a destination point [43], using linear genomes for robotic control [2, 13], and using cellular encoding (developmental genetic programming) for the pole-balancing problem involving two poles [85].

In most real-world control systems, the controller continuously interacts with the plant (and reference signal) and the plant interacts with the controller. The behavior of each part of the overall system depends on the behavior of the other parts. Genetic algorithms and genetic programming have also been used for synthesizing controllers having mutually interacting continuous-time variables and continuous-time signal processing blocks [51, 52, 25, 27, 54, 70, 71]. Neural programming and PADO [72 through 82] provide a graphical program structure representation for such multiple independent processes. In 1996, Marenbach, Bettenhausen, and Freyer [53] used automatically defined functions to represent internal feedback in a system used for system identification where the overall multibranch 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" [5 through 8].

*Use of genetic programming to automatically synthesize controllers*

The program trees (consisting of both result-producing branches and automatically defined functions) evolved by genetic programming may be employed in several different ways. In each case, the search mechanism of genetic programming (i.e., the three steps above) is the same.

In the approach where genetic programming is being used to automatically create a computer program to solve a problem, the program tree is simply executed. The result of the execution is 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 individually executed, in time, in accordance with a specified "order of evaluation" such that the result of the execution of one function is available at the time when the next function is going to be executed. In particular, the functions in a subroutine (automatically defined function) in an ordinary computer program are executed at the distinct time when the subroutine is invoked; the subroutine then produces a result that is available at the time when the next function is executed.

The second approach is a developmental approach. In this approach, the program tree is interpreted as a set of instructions for constructing a complex structure, such an electrical circuit [43]. This is accomplished 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 the specified "order of evaluation." Fifteen of the entries in Table 1 are electrical circuits that were evolved by means of developmental genetic programming.

In this paper, program trees are used in a third way. As will be explained in detail below, the program tree represents 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 as part of a closed system in the manner specified by the topology of the block diagram for the overall system.

As will be seen below, genetic programming can be successfully used to automatically synthesize both the topology and parameter values for a controller using this third approach. The automatically created controllers can potentially accommodate one or more externally supplied reference signals; external feedback of one or more plant outputs to the controller; comparisons between one or more reference signals and their corresponding plant outputs; one or more control variables; zero, one, or more internal state variables of the plant; direct feedback of the controller's output back into the controller; and internal feedback of zero, one, or more signals from one part of the controller to another part of the controller. The automatically created controllers can be composed of signal processing blocks such as (but not limited to) gain blocks, lead blocks, lag blocks, inverter blocks, differential input integrators, differentiators, delays, and adders and subtractors and multipliers of time-domain signals. These controllers can also potentially contain conditional operators (switches) that operate on time-domain signals.

## *Repertoire of functions*

The repertoire of functions includes the following time-domain functions (called signal processing blocks when they appear in a block diagram).

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

The one-argument `DIFFERENTIATOR` function differentiates the time-domain signal represented by its argument. That is, this function applies the transfer function $s$, where $s$ is the Laplace transform variable.

The one-argument `INTEGRATOR` function integrates the time-domain signal represented by its one argument. That is, this function applies the transfer function $1/s$.

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

The two-argument `LEAD` function applies the transfer function $1 + \tau s$, where $\tau$ is a real-valued numerical parameter. The first argument is the time-domain input signal. The second argument, $\tau$, is a numerical parameter representing the time constant (usually expressed in seconds) of the `LEAD`. The numerical parameter value for this function (and other functions described below) may be represented using one of three different approaches (described below).

The two-argument `LAG` function applies the transfer function $1/(1 + \tau s)$, where $\tau$ is a numerical parameter. The first argument is the time-domain input signal. The second argument, $\tau$, is the time constant.

The three-argument `LAG2` function applies the transfer function

$$\frac{\omega_0^2}{s^2 + 2\zeta\omega_0 s + \omega_0^2},$$

where $\zeta$ is the damping ratio, and $\omega_0$ is the corner frequency.

The two-argument `ADD_SIGNAL, SUB_SIGNAL`, and `MULT_SIGNAL` functions perform addition, subtraction, and multiplication, 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 one-argument `ABS_SIGNAL` function performs the absolute value function on the time-domain signal represented by its argument.

The three-argument `LIMITER` function limits a signal by constraining it between an upper and lower bound. This function returns the value of its first argument (the incoming signal) when its first argument lies between its second and third arguments (the two bounds). If the first argument is greater than its third argument (the upper bound), the function returns its third argument. If its first argument is less than its second argument (the lower bound), the function returns its second argument.

The four-argument DIV_SIGNAL function divides the time-domain signals represented by their two arguments and constrains the resulting output by passing the quotient through a built-in LIMITER function with a specified upper and lower bound. The LIMITER function that is built into the DIV_SIGNAL function protects against the effect of dividing by zero (or by a near-zero value) by returning a specified bound.

The two-argument GAIN function multiplies the time-domain signal represented by its first argument by a constant numerical value represented by its second argument. This numerical value is constant in the sense that it is not a time-domain signal (like the first argument to the GAIN function) and in the sense that this numerical value does not vary when the controller is operating. Note that the GAIN function differs from the MULT_SIGNAL function (described above) in that the second argument of a GAIN function is a constant numerical value whose value does not vary when the controller is operating.

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

The three-argument IF_POSITIVE function is a switching function that operates on three time-domain signals and produces a particular time-domain signal depending on whether its first argument is positive. If, at a given time, the value of the time-domain function in the first argument of the IF_POSITIVE function is positive, the value of the IF_POSITIVE function is the value of the time-domain function in the second argument of the IF_POSITIVE function. If, at a given time, the value of the time-domain function in the first argument of the IF_POSITIVE function is negative or exactly zero, the value of the IF_POSITIVE function is the value of the time-domain function in the third argument of the IF_POSITIVE function.

The one-argument DELAY function has one numerical parameter, its time delay, and applies the transfer function $e^{-sT}$ where $T$ is the time delay.

Automatically defined functions (e.g., ADF0, ADF1) possessing one or more arguments may also be included in the function set of a particular problem (described below).

## *Repertoire of terminals*

The repertoire of terminals includes (but is not limited to) the following terminals.

The REFERENCE_SIGNAL is the time-domain signal representing the desired plant response. If there are multiple reference signals, they are named REFERENCE_SIGNAL_0, REFERENCE_SIGNAL_1, and so forth.

The PLANT_OUTPUT is the plant output. Inclusion of the terminal PLANT_OUTPUT in the terminal set of the problem provides a means to have external feedback of the plant's output into the to-be-evolved controller. If the plant has multiple outputs, the plant outputs are named PLANT_OUTPUT_0, PLANT_OUTPUT_1, and so forth.

The CONTROLLER_OUTPUT is the time-domain signal representing the output of the controller (i.e., the control variable). Inclusion of the terminal CONTROLLER_OUTPUT in the terminal set of the problem provides a means to have direct feedback of the controller's output back into the to-be-evolved controller. If the controller has multiple control variables, the control variables are named CONTROLLER_OUTPUT_0, CONTROLLER_OUTPUT_1, and so forth.

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

The CONSTANT_0 terminal is the constant time-domain signal whose value is always 0. Similar terminals may be defined, if desired, for other particular constant valued time-domain signals.

Additional terminals may also be included in the terminal set to represent constant numerical values.

Zero-argument automatically defined functions may also be included in the terminal set of a particular problem (as described below).

### *Representing the plant*

Use of genetic programming to evolve a satisfactory controller requires determining the behavior of the overall system (composed of both the controller and plant). This behavior is typically determined by a single run of a simulator that simulates the controller and plant together. In practice, many plants can be modeled with the same repertoire of functions and terminals (described above) as a controller. The technique described in this paper assumes that the behavior of the plant is available. If a model of the plant is not available, genetic programming can be used in conjunction with the representational scheme in this paper to create a model for the plant (i.e., perform the task of system identification).

### *Automatically defined functions*

An automatically defined function (ADF) is a function whose body is dynamically evolved during the run and which may be invoked by the main result-producing branch(es) or by other automatically defined function(s). When automatically defined functions are being used, an individual program tree consists of one or more reusable automatically defined functions (function-defining branches) along with the main result-producing branch(es). Automatically defined functions may possess zero, one, or more dummy arguments (formal parameters) and may be reused with different instantiations of their dummy arguments.

Automatically defined functions provide a convenient mechanism for representing takeoff points. Once an automatically defined function is defined, it may be referenced repeatedly by other parts of the overall program tree representing the controller. Thus, an automatically defined function may be used to disseminate the output of a particular processing block within a controller to two or more other points in the block diagram of the controller.

In addition, automatically defined functions provide a convenient mechanism for enabling internal feedback within a controller. In control problems, the function set for each automatically defined function and each result-producing branch includes each existing automatically defined function (including itself). Specifically, each automatically defined function is a composition of the above functions and terminals, all existing automatically defined functions, and (possibly) dummy variables (formal parameters) that parameterize the automatically defined functions. Note that in the style of ordinary computer programming, a reference to `ADF0` from inside the function definition for `ADF0` would be considered to be a recursive reference. However, in the context of control structures, an automatically defined function that references itself represents a cycle in the block diagram of the controller (i.e., internal feedback inside the controller).

In this paper, programs trees in the initial random generation (generation 0) consist only of result-producing branches. Automatically defined functions are introduced sparingly on subsequent generations of the run by means of the architecture-altering operations. Alternatively, automatically defined functions may be present in the individuals in generation 0.

The to-be-evolved controller can accommodate one or more externally supplied reference 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 passed between the controller and the plant. 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 amount of internal feedback is automatically determined during the run.

## *Numerical parameter values*

Many signal processing block functions (such as the `GAIN, BIAS, LEAD, LAG`) possess a numerical parameter value and others (such as `LAG2` and `LIMITER`) possess more than one numerical parameter values.

The following three approaches (each employing a constrained syntactic structure) can be used to establish the numerical parameter values for signal processing blocks:

(1) an arithmetic-performing subtree consisting of one (and usually more than one) arithmetic functions and one (and usually more than one) constant numerical terminals,
(2) a single perturbable numerical value,
(3) an arithmetic-performing subtree consisting of one (and usually more than one) arithmetic functions and one (and usually more than one) perturbable numerical values.

The first approach has been extensively employed in connection with the automatic synthesis of analog electrical circuits [43].

In second approach, the numerical parameter value of a signal processing block function is established by a single perturbable numerical value (coded by 30 bits in our system). These perturbable numerical values are changed during the run (unlike the constant numerical terminals of the first approach). In the initial random generation, each perturbable numerical value is set, individually and separately, to a random value in a chosen range (e.g., between $+5.0$ and $-5.0$). In later generations, the perturbable numerical value may be perturbed by a relatively small amount determined probabilistically by a Gaussian probability distribution. The existing to-be-perturbed value is considered to be the mean of the Gaussian distribution. A relatively small preset parameter establishes the standard deviation of the Gaussian distribution. The standard deviation of the Gaussian perturbation may be, for example, 1.0 (i.e., corresponding to one order of magnitude if the number between $+5.0$ and $-5.0$ is later interpreted, as described below, on a logarithmic scale). This second approach has the advantage (over the first approach) of changing numerical parameter values by a relatively small amount and therefore searching the space of possible parameter values most thoroughly in the immediate neighbor of the value of the existing value (which is, because of Darwinian selection, is necessarily part of a relatively fit individual). These perturbations are implemented by a genetic operation for mutating the perturbable numerical values. It is also possible to perform a special crossover operation in which a copy of a perturbable numerical value is inserted in lieu of a chosen other perturbable numerical value. Our experience (albeit limited) is that this second approach, patterned after the Gaussian mutation operation used in evolution strategies [60, 61] and evolutionary programming [29], appears to work better than the first approach.

The third approach is more general than the second approach and employs arithmetic-performing subtrees in conjunction with perturbable numerical values. This approach differs from the second approach in that a full subtree is used, instead of only a single perturbable numerical value. This approach may be advantageous when there are external global variables or when automatically defined functions, such as `ADF0`, and dummy variables (formal parameters), such as `ARG0`, are involved in establishing numerical parameter values for signal processing blocks.

Regardless of which of the above approaches is used to represent the numerical parameter values, it is advantageous to interpret the value returned by the arithmetic-performing subtree or perturbable numerical value on a logarithmic scale (e.g., converting numbers ranging between $-5.0$ and $+5.0$ into numbers ranging over 10 orders of magnitude) as described in detail in [43].

## *Constrained syntactic structure of the program trees*

The program trees in the initial generation 0 as well as any trees created in later generations of the run by the mutation, crossover, and architecture-altering operations are created in compliance with a constrained syntactic structure (strong typing) that limits the particular functions and terminals that may appear at particular points in each particular branch of the overall program tree. An individual tree consists of main result-producing branch(es) and zero, one, or more automatically defined functions (function-defining branches) and each of these branches is composed of a composition of various functions and terminals.

The functions and terminals in the trees are divided into three categories:

(1) signal processing block functions,
(2) automatically defined functions that appear in the function-defining branches and that enable both internal feedback within a controller and the dissemination of the output from a particular signal processing block within a controller to two or more other points in the block diagram of the controller, and
(3) terminals and functions that may appear in arithmetic-performing subtrees for the purpose of establishing the numerical parameter value for certain signal processing block functions.

*Program tree representations of controllers*

The PID controller of Figure 1 may be represented by a composition of functions and terminals in the style of symbolic expressions (S-expressions) of the LISP programming language. The block diagram for the PID controller of Figure 1 can be represented as the following S-expression:

```
1  (PROGN
2   (DEFUN ADF0 ()
3    (VALUES
4     (- REFERENCE_SIGNAL PLANT_OUTPUT)))
5   (VALUES
6    (+
7     (GAIN 214.0 ADF0)
8     (DERIVATIVE (GAIN 1000.0 ADF0))
9     (INTEGRATOR (GAIN 15.5 ADF0))))
10 )
```

Notice that the automatically defined function ADF0 provides the mechanism for disseminating a particular signal (the difference taken on line 4) to three places (lines 7, 8, and 9) and corresponds to the takeoff point 520 in Figure 1.

A controller may also be represented as a point-labeled tree with ordered branches (i.e., a program tree) that corresponds directly with the above S-expression representation. The terminals of such program trees correspond to inputs to the controller, constant numerical terminals, perturbable constant values, or externally supplied (global) variables. The functions in such program trees correspond to the signal processing blocks in a block diagram representing the controller. The value returned by an entire result-producing branch of the program tree corre-

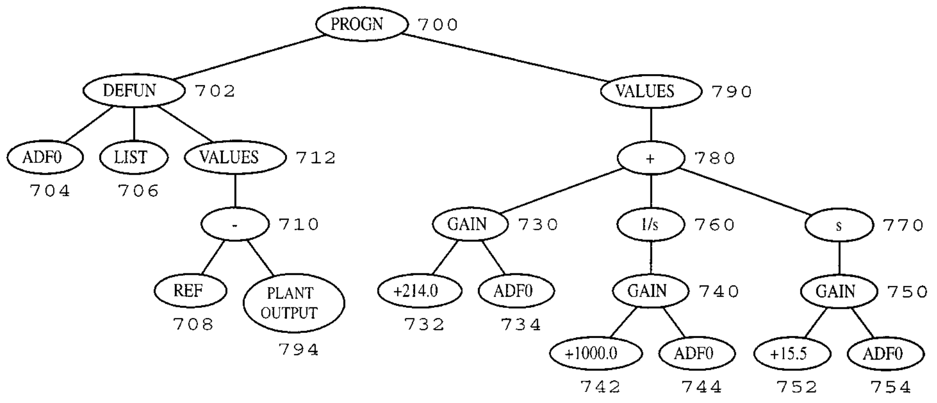

*Figure 2.* Program tree representation of the PID controller of Figure 1. The automatically defined function ADF0 (left) subtracts the plant output from the reference signal and makes the difference available to three points (734, 744, and 754) in the result-producing branch (right).

sponds to an output of the controller (i.e., control variable) that is to be passed from the controller to the plant. If the controller has more than one control variable, the program tree has one result-producing branch for each control variable. Figure 2 presents the block diagram for the PID controller of Figure 1 as a program tree. All block diagrams for controllers may be represented in this manner.

A controller may also be represented as a SPICE netlist. The SPICE simulator can be used for simulating controllers and plants. SPICE is a large family of programs written over several decades at the University of California at Berkeley for the simulation of analog, digital, and mixed analog/digital electrical circuits. SPICE3 [59] is currently the most recent version of Berkeley SPICE and consists of about 217,000 lines of C source code residing in 878 separate files. The required input to the SPICE simulator consists of a netlist along with certain commands and other commands required by the SPICE simulator. The netlist describes the topology and parameter values of a controller and plant and can be derived directly from the program tree for the controller and the block diagram for the plant. The SPICE simulator was originally designed for simulating electrical circuits. Circuit diagrams differ from block diagrams in several important ways. In particular, the leads of a circuit's electrical components are joined so there is no directionality in circuits (as there is in block diagrams). In addition, SPICE does not ordinarily handle most of the signal processing functions typically contained in a controller and plant (e.g., derivative, integral, lead, and lag). Nonetheless, behavior of the signal processing functions (such as derivative, integral, lead, lag) can be realized by using the facility of SPICE to create subcircuit definitions involving appropriate combinations of electrical components and the facility of SPICE to implement user-defined continuous-time mathematical calculations.

## 4. Two illustrative problems

We use two example problems for illustrating the use of genetic programming for automatically synthesizing controllers.

### *Two-lag plant*

The first illustrative problem calls for the design of a robust controller for a two-lag plant. The problem (described by Dorf and Bishop in [28, page 707]) is to create both the topology and parameter values for a controller for a two-lag plant such that plant output reaches the level of the reference signal so as to minimize the integral of the time-weighted absolute error (ITAE), such that the overshoot in response to a step input is less than 2%, and such that the controller is robust in the face of significant variation in the plant's internal gain, $K$, and the plant's time constant, $\tau$. The transfer function of the plant is

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

The plants internal gain, $K$, is varied from 1 to 2 and the plant's time constant, $\tau$, is varied from 0.5 to 1.0.

To make the problem more realistic, we added two additional constraints to the problem. Both of these added constraints are, in fact, satisfied by the Dorf and Bishop controller. These two constraints are of the type that are often implicit in work in the field of control. The first constraint is that the input to the plant is limited to the range between $-40$ and $+40$ volts. Limiting the control variable passing into the plant reflects the limitations of real world actuators: a motor has a maximum armature current; a furnace a maximum rate of heat production, etc. 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. This bandwidth limitation reflects the desirability of limiting the effect of high frequency noise in the reference input. Note that the 2% overshoot requirement in the above statement of the problem is more stringent than the 4% overshoot requirement used by Dorf and Bishop.

Figure 3 shows an illustrative two-lag plant consisting of a series composition of a `LIMITER` block (with a range $-40.0$ volts to $+40.0$ volts) and two `LAG` blocks (each with a lag of 1.0). In the figure, the input to the plant 600 is control variable 610 (the output of some controller). This signal is first passed into `LIMITER` function block 620 whose upper limit is established by the numerical parameter $+40.0$ (at 622 of the figure) and whose lower limit is established by the numerical parameter $-40.0$ (at 624). The output of `LIMITER` function block 620 is signal 626. This signal then passes into first `LAG` block 630 whose time constant is established by the numerical parameter 1.0 (at 632). The output of this first `LAG` block 630 is signal 636. This signal then passes into second `LAG` block 640 whose time constant

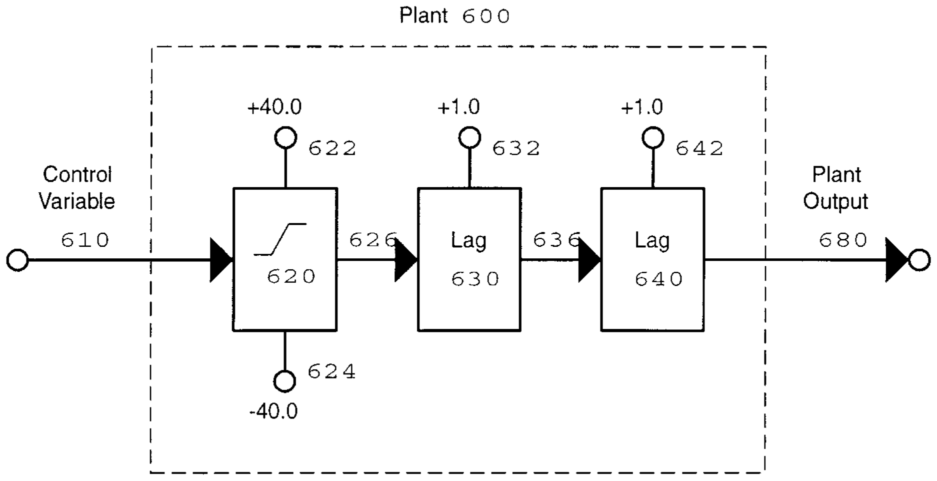

*Figure 3.* Block diagram of an illustrative two-lag plant consisting of a limiter block that constrains the plant's input (the control variable) between −40 and +40 volts and two consecutive lag blocks (each with a time constant of 1.0).

is established by the numerical parameter 1.0 (at 642). The output of this second LAG block 640 is plant output 680.

A textbook PID compensator preceded by a lowpass prefilter delivers credible performance on this problem. Dorf and Bishop [28] say that they "obtain the optimum ITAE transfer function." By this, they mean they obtained the optimum given that they had already decided to employ a PID compensator.

As will be seen below, the result produced by genetic programming differs from a conventional PID controller in that the genetically evolved controller employs a second derivative processing block. As will be seen, the genetically evolved controller is 2.42 times better than the Dorf and Bishop [28] controller as measured by the criterion used by Dorf and Bishop (namely, the integral of the time-weighted absolute error). In addition, the genetically evolved controller has only 56% of the rise time in response to the reference input, has only 32% of the settling time, and is 8.97 times better in terms of suppressing the effects of a step disturbance at the plant input.

## *Three-lag plant*

The technique for automatically synthesizing a controller will be further illustrated by a problem calling for the design of a similarly robust controller for a three-lag plant (described by Astrom and Hagglund in [10, page 225]). The three-lag plant is identical to the plant of Figure 3 except that there is one additional lag block with a time constant of $\tau$. The transfer function of the plant is

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

The plant's internal gain, $K$, is varied from 1 to 2 and the plant's time constant, $\tau$, is varied from 0.5 to 1.0. Since Astrom and Hagglund do not explicitly mention a limit on the controller output, we added an additional constraint (i.e., the control variable is limited to the range between $-10$ and $+10$ volts) that is both reasonable and, in fact, satisfied by the Astrom and Hagglund controller.

A PID compensator designed by Astrom and Hagglund [10] delivers credible performance on this problem.

As will be seen below, the controller produced by genetic programming is better than 7.2 times as effective as the textbook controller as measured by the integral of the time-weighted absolute error, has only 50% of the rise time in response to the reference input, has only 35% of the settling time, and is 92.7 dB better in terms of suppressing the effects of a step disturbance at the plant input.

## 5. 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.

### *Program architecture*

Since both problems involve one-output controllers, 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, in subsequent generations, the architecture-altering operations may insert (and delete) automatically defined functions to particular individual program trees in the population. We decided to permit each program tree to acquire a maximum of five automatically defined functions.

### *Terminal set*

For the two-lag plant problem, arithmetic-performing subtrees involving constant numerical terminals are used for establishing the values of the numerical parameter(s) for the signal processing blocks in the overall program tree. A constrained syntactic structure enforces a different function and terminal set for the arithmetic-performing subtrees (as opposed to all other parts of the program tree). For the two-lag plant problem, the terminal set, $\mathsf{T}_{\mathsf{aps}}$, for the arithmetic-performing subtrees is

$$\mathsf{T}_{\mathsf{aps}} = \{\Re\},$$

where $\Re$ denotes constant numerical terminals in the range from $-1.0$ to $+1.0$.

For the three-lag plant problem, a constrained syntactic structure permits only a single perturbable numerical value to appear as the argument for establishing each numerical parameter value for each signal processing block. These two approaches are used in this paper for purposes of illustration, not because either problem is more suited to a particular approach than the other.

For both problems, the terminal set, T, for every part of the result-producing branch and any automatically defined functions except the arithmetic-performing subtrees is

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

*Function set*

For the two-lag plant problem, the function set, $\mathsf{F}_{\mathsf{aps}}$, for the arithmetic-performing subtrees is

$\mathsf{F}_{\mathsf{aps}}$ = {ADD_NUMERIC, SUB_NUMERIC}.

For both problems, the function set, F, for every part of the result-producing branch and any automatically defined functions except the arithmetic-performing subtrees is

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

Here ADF0, ADF1, ... denote the automatically defined functions added during the run by the architecture-altering operations.

*Fitness measure*

Genetic programming is a probabilistic search algorithm that searches 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."

The fitness measure may incorporate any measurable, observable, or calculable behavior or characteristic or combination of behaviors or characteristics. Construction of the fitness measure requires translating the high-level requirements of the problem into a mathematically precise computation.

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 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, a block diagram for the controller. The netlist for the resulting controlled system (i.e., the controller and the to-be-controlled plant) is wrapped inside an appropriate set of SPICE commands. The controller is then simulated using our modified version of the SPICE simulator [59]. The SPICE simulator returns tabular and other information from which the fitness of the individual can be computed.

The fitness measures below for the two illustrative problems are suggestive of the many different considerations that may be incorporated into a fitness measure that is used to guide the evolutionary process in synthesizing a controller. The fitness measures below illustrate five different types of considerations, including

(1) an optimization requirement (e.g., minimizing the integral of the time-weighted absolute error),
(2) time-domain constraints (e.g., the overshoot penalty, disturbance rejection, and the penalty based on the response to an extreme spiked reference signal),
(3) a frequency-domain constraint (e.g., an AC sweep over the frequencies),
(4) robustness requirements (e.g., significant variations in the values of the plant's internal gain and the plant's time constant), and
(5) consistency of treatment in the face of variations in the step size of the reference signal.

Intermixing of different types of considerations is often difficult and sometimes impossible when conventional analytical techniques are used to design controllers.

***Fitness measure for the two-lag plant problem.*** For the-two-lag plant problem, the fitness of a controller is measured using 10 elements, including

(1) eight time-domain-based elements based on a modified integral of time-weighted absolute error measuring the achievement of the desired value of the plant response, the controller's robustness, and the controllers avoidance of overshoot,
(2) one time-domain-based element measuring the controller's stability when faced with an extreme spiked reference signal, and
(3) one frequency-domain-based element measuring the reasonableness of the controller's frequency response.

The fitness of an individual controller is the sum (i.e., a simple linear combination) of the detrimental contributions of these 10 elements of the fitness measure. The smaller the sum, the better.

The first eight elements of the fitness measure together evaluate (i) how quickly the controller causes the plant to reach the reference signal, (ii) the robustness of the controller in face of significant variations in the plant's internal gain and the plant's time constant, and (iii) the success of the controller in avoiding overshoot. These eight elements of the fitness measure represent the eight choices of a particular one of two different values of the plant's internal gain, $K$, in conjunction with a particular one of two different values of the plant's time constant $\tau$, in conjunction with a particular one of two different values for the height of the reference signal. The two values of $K$ are 1.0 and 2.0. The two values of $\tau$ 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. The second reference signal rises from 0 to 1 microvolts at $t = 100$ milliseconds. The two values of $K$ and $\tau$ are used in order to obtain a robust controller. The two step functions are used to deal with the nonlinearity caused by the limiter. For each of these eight fitness cases, a transient analysis is performed in the time domain using the SPICE simulator. The contribution to fitness for each of these eight elements of the fitness measure is based on the integral of time-weighted absolute error

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

Here $e(t)$ is the difference (error) at time $t$ between the plant output and the reference signal. An integration between $t = 0$ to $t = 9.6$ seconds is sufficient to capture all interesting behavior of any reasonable controller for this problem. The multiplication of each value of $e(t)$ by $B$ makes both reference signals 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 $10^6$. The integral also contains an additional weighting function, $A$, that heavily penalizes noncompliant amounts of overshoot. Specifically, the function $A$ depends on $e(t)$ and weights all variations up to 2% above the reference signal by a factor of 1.0, while $A$ weights overshoots above 2% by a factor 10.0. A discrete approximation to the integral employing 120 80-millisecond time steps is sufficient to yield a solution to this problem.

The ninth element of the fitness measure evaluates the stability of the controller when faced with an extreme spiked reference signal. The spiked reference signal rises to $10^{-9}$ volts at time $t = 0$ and persists for 10-nanoseconds. The reference signal is then 0 for all other times. A transient analysis is performed using the SPICE simulator for 121 fitness cases representing times $t = 0$ to $t = 120$ microseconds. If the plant output never exceeds a fixed limit of $10^{-8}$ volts (i.e., a order of magnitude greater than the pulse's magnitude) for any of these 121 fitness cases, then this element of the fitness measure is zero. However, if the absolute value of plant output goes above $10^{-8}$ volts for any time $t$, then the contribution to fitness is $500(0.000120 - t)$, where $t$ is first time (in seconds) at which the absolute value of plant output goes above $10^{-8}$ volts. This penalty is a ramp starting at the point $(0, 0.06)$ and ending at the point $(1.2, 0)$, so that 0.06 seconds is the maximum penalty and 0 is the minimum penalty.

The tenth element of the fitness measure for the two-lag plant problem 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. If the closed loop frequency response is acceptable, this element of the fitness measure will be zero. 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 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 maximum acceptable 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 SPICE simulator cannot simulate some of the controllers that are randomly created for the initial random generation and some of the controllers that are created by the mutation, crossover, and architecture-altering operations in later generations of the run. A controller that cannot be simulated by SPICE is assigned a high penalty value of fitness ($10^8$). Such controllers become worst-of-generation individuals for their generation.

If an individual controller in the population takes more than a specified amount of time (e.g., 20 seconds of computer time), the simulation is terminated and the individual is assigned a fitness of $10^8$.

***Fitness measure for the three-lag plant problem.*** For the three-lag plant problem, the fitness of a controller is measured using 10 elements. The first nine elements are the same as for the two-lag plant problem.

The tenth element of the fitness measure for the three-lag plant problem is based on disturbance rejection. This tenth element is computed based on a time-domain analysis for 9.6 seconds. In this analysis, the reference signal is held at a value of 0. A disturbance signal consisting of a unit step is added to the controller variable (plant input) at time $t = 0$ and the resulting disturbed signal is provided as input to the plant. The detrimental contribution to fitness is the absolute value of the largest single difference between the plant output and the reference signal (which is invariant at 0 throughout).

The number of hits for the three-lag plant problem is defined as the number of elements (0 to 6) for which the individual controller has a smaller (better) contribution to fitness than the controller of Astrom and Hagglund [10].

Different fitness measures are used for the two problems in this paper for the purpose of demonstrating the ease of combining optimization requirements, time-domain constraints, frequency-domain constraints, and robustness requirements into a fitness measure (and not because the three-lag problem specifically requires a different fitness measure than the two-lag problem).

Note that many alternative approaches could have been used in constructing the fitness measures for the two illustrative problems. Different optimization metrics might have been used including, for example, the integral of the squared error, the settling time (defined below), and the rise time (defined below). There are numerous alternative time-domain constraints that may be included in a fitness measure (including, for example, measuring stability in ways other than the response to a spiked reference signal). Similarly, there are numerous other frequency-domain constraints that may be included as elements of a fitness measure. Robustness may be included in a fitness measure with respect to any aspect of the plant that might potentially vary. In addition, the fitness measure may be constructed to include elements measuring the robustness of the behavior of the plant in the face of sensor noise (added to the plant output, the reference signal, or the plant's internal states, if any are being made available to the controller). Also, the fitness measure may be constructed to impose constraints on the plant's internal states or the control variable (the controller's output) by, for example, penalizing extreme values of the plant's internal states or the control variable. The fitness measure may also be constructed to include elements measuring the robustness of the plant's behavior with respect to changes of some external variable that affects the plant's operation (such as temperature, the plant's production rate, line speed, flow rate, or the like, or other free variable characterizing the operation of the plant).

### *Control parameters for the run*

For both problems, the population size, $M$, was 66,000. A maximum size of 150 points (for functions and terminals) was established for each result-producing branch and a maximum size of 100 points was established for each automatically defined function.

For the three-lag plant, the percentages of the genetic operations for each generation on and after generation 5 are 47% 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, 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 by using an increased percentage for the architecture-altering operations that add subroutines (i.e., subroutine creation and subroutine duplication) prior to generation 5. Specifically, the percentages for the genetic operations for each generation up to generation 5 are 45% 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, 5% 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, 5% subroutine creation, 5% subroutine duplication, and 1% subroutine deletion.

The percentages of operations for the two-lag plant problem are similar, except that no distinction is made between numerical constant terminals and other terminals.

The other parameters for controlling the runs are the default values that we apply to a broad range of problems [43].

### *Termination*

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.

### *Parallel implementation*

Both problem were run on a home-built Beowulf-style [69, 43, 16] parallel cluster computer system consisting of 66 processors (each containing a 533-MHz DEC Alpha microprocessor and 64 megabytes of RAM) arranged in a two-dimensional $6 \times 11$ toroidal mesh. The system has a DEC Alpha type computer as host. The processors are connected with a 100 megabit-per-second Ethernet. The processors and the host use the Linux operating system. The distributed genetic algorithm 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 processors.

## 6. Results for the two-lag plant

A run of genetic programming starts with the creation of an initial population of individual trees (each consisting of one result-producing branch) composed of the functions and terminals identified above and in accordance with the constrained syntactic structure described above.

The initial random population of a run of genetic programming is a blind random parallel search of the search space of the problem. As such, it provides a baseline for comparing the results of subsequent generations.

The best individual from generation 0 of our one (and only) run of this problem has a fitness of 8.26. The S-expression for this individual is shown below (except that a 29-point arithmetic-performing subtree establishing the amplification factor for the GAIN function has been replaced by the equivalent numerical value of 62.8637):

```
(GAIN
 (DIFFERENTIATOR
  (DIFFERENTIAL_INPUT_INTEGRATOR
   (LAG REFERENCE_SIGNAL 0.708707)
   PLANT_OUTPUT
  )
 )
62.8637)
```

The differentiation offsets the integration in the above best-of-generation individual, so that the action of the controller is based on the difference between the REFERENCE_SIGNAL and the PLANT_OUTPUT (amplified by some factor). Thus, this best-of-generation individual, like many of the other individuals from early generations of the run, resembles the proportionate (P) portion of a PID controller. A P-type controller is a poor controller for this problem; however, it is a rudimentary starting point for the evolution of a better controller.

Generation 1 (and each subsequent generation of a run of genetic programming) is created from the population at the preceding generation by performing reproduction, crossover, mutation, and architecture-altering operations on individuals (or pairs of individuals in the case of crossover) selected probabilistically from the population on the basis of fitness.

Both the average fitness of all individuals in the population as a whole and the fitness of the best individual in the population tend to improve over successive generations.

Sixty percent of the programs of generation 0 for this run produce controllers that cannot be simulated by SPICE. The unsimulatable programs are the worst-of-generation programs for each generation and receive the high penalty value of fitness ($10^8$). However, the percentage of unsimulatable programs drops to 14% by generation 1 and 8% by generation 10. The vast majority of the offspring created by genetic programming are simulatable after just a few generations.

Controllers resembling PI, PD, and PID controllers (typically deviating from the canonical form only by a LEAD or LAG block with a small time constant) are common throughout the intermediate generations of the run. That is, runs of genetic programming routinely rediscover the utility of the PI and PID controller topology invented by Callender and Stevenson [19].

The best-of-run individual emerges in generation 32 and has a near-zero fitness of 0.1639. Table 2 shows the contribution of each of the 10 elements of the fitness measure for the best-of-run individual of generation 32 for the two-lag plant problem.

Most of the computer time was consumed by the fitness evaluation of candidate individuals in the population. The fitness evaluation (involving 10 separate SPICE

*Table 2.* Fitness of best-of-run individual of generation 32 for the two-lag plant problem

| | Step size (volts) | Internal gain, $K$ | Time constant, $\tau$ | Fitness |
|---|---|---|---|---|
| 0 | 1 | 1 | 1.0 | 0.0220 |
| 1 | 1 | 1 | 0.5 | 0.0205 |
| 2 | 1 | 2 | 1.0 | 0.0201 |
| 3 | 1 | 2 | 0.5 | 0.0206 |
| 4 | $10^{-6}$ | 1 | 1.0 | 0.0196 |
| 5 | $10^{-6}$ | 1 | 0.5 | 0.0204 |
| 6 | $10^{-6}$ | 2 | 1.0 | 0.0210 |
| 7 | $10^{-6}$ | 2 | 0.5 | 0.0206 |
| 8 | Spiked reference signal | | | 0.0000 |
| 9 | AC sweep | | | 0.0000 |
| TOTAL FITNESS | | | | 0.1639 |

simulations) averaged $2.57 \times 10^9$ computer cycles (4.8 seconds) per individual. The best-of-run individual from generation 32 was produced after evaluating $2.178 \times 10^6$ individuals (66,000 times 33). This required 44.5 hours on our 66-node parallel computer system—that is, the expenditure of $5.6 \times 10^{15}$ computer cycles (5 petacycles of computing effort).

Figure 4 shows this individual controller in the form of a block diagram. 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).

Figure 5 compares the time-domain response of the best-of-run genetically evolved controller (triangles in the figure) from generation 32 for a 1 volt unit step with $K = 1$ and $\tau = 1$ with the time-domain response (circles) of the Dorf and Bishop controller. The faster rising curve in the figure shows the performance of the genetically evolved controller.

The rise time is the time required for the plant output to first reach a specified percentage (90% here) of the reference signal. As can be seen in the figure, the rise time for the best-of-run controller from generation 32 is 296 milliseconds. This is 56% of the 465-millisecond rise time for the Dorf and Bishop controller.

The settling time is the first time for which the plant response reaches and stays within a specified percentage (2% here) of the reference signal. The settling time

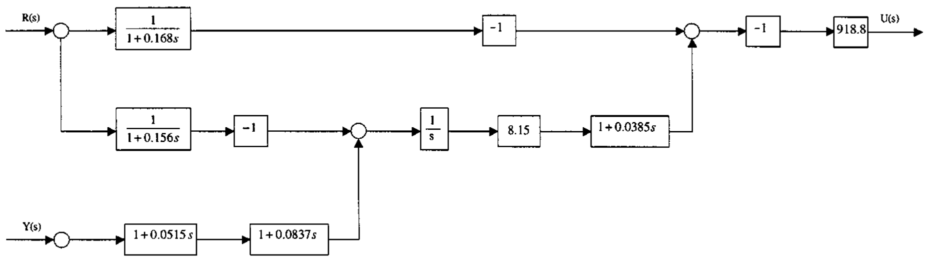

*Figure 4.* Best-of-run genetically evolved controller from generation 32 for the two-lag plant problem.

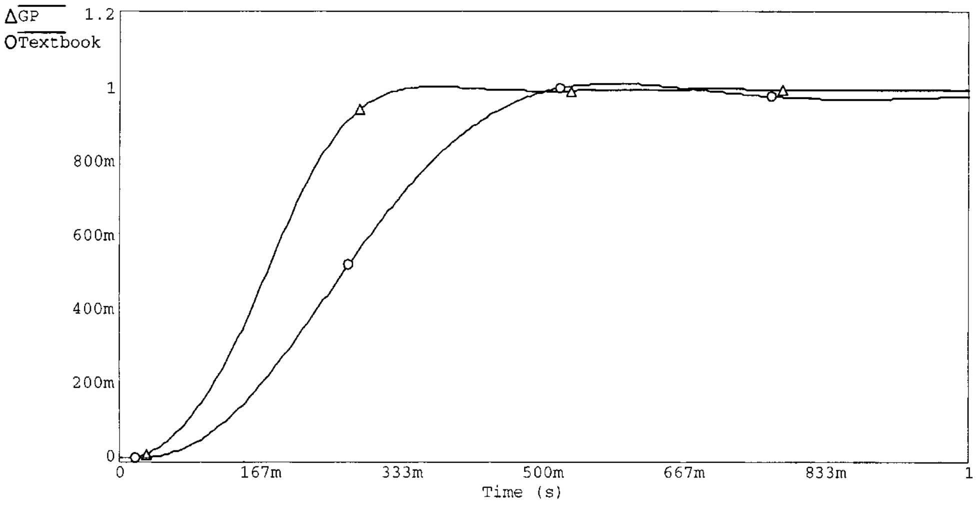

*Figure 5.* Comparison of the time-domain response for the genetically evolved controller (triangles) and the Bishop and Dorf controller (circles) to 1-volt step input with $K = 1$ and $\tau = 1$ for the two-lag plant.

for the best-of-run controller from generation 32 is 304 milliseconds. The Dorf and Bishop controller first reaches the 98% level at 477 milliseconds; however, it rings and subsequently falls below the 98% level. It does not settle until 944 milliseconds. That is, the genetically evolved controller settles in 32% of the settling time for the Dorf and Bishop controller.

The overshoot is the percentage by which the plant response exceeds the reference signal. The best-of-run controller from generation 32 reaches a maximum value of 1.0106 at 369 milliseconds (i.e., has a 1.06% overshoot). The Dorf and Bishop controller reaches a maximum value of 1.020054 at 577 milliseconds (i.e., has an overshoot of slightly above 2%).

The curves for all the other values of $K$ and $\tau$ similarly favor the genetically evolved controller.

Figure 6 compares the effect of disturbance on the best-of-run genetically evolved controller (triangles) from generation 32 and the controller (circles) presented in Dorf and Bishop. The upper curve in the figure is the time-domain response to a 1-volt disturbance signal with $K = 1$ and $\tau = 1$ for the Dorf and Bishop controller. The peak value of response to the 1-volt disturbance signal is 5,775 microvolts. The lower curve (whose peak is 644 microvolts) applies to the best-of-run controller from generation 32.

The system bandwidth is the frequency of the reference signal above which the plant's output is attenuated by at least a specified degree (3 dB here) in comparison to the plant's output at a specified lower frequency (e.g., DC or very low frequencies).

Table 3 compares the average performance (over the eight combinations of values for $K$, $\tau$, and the step size of the reference signal) of the best-of-run

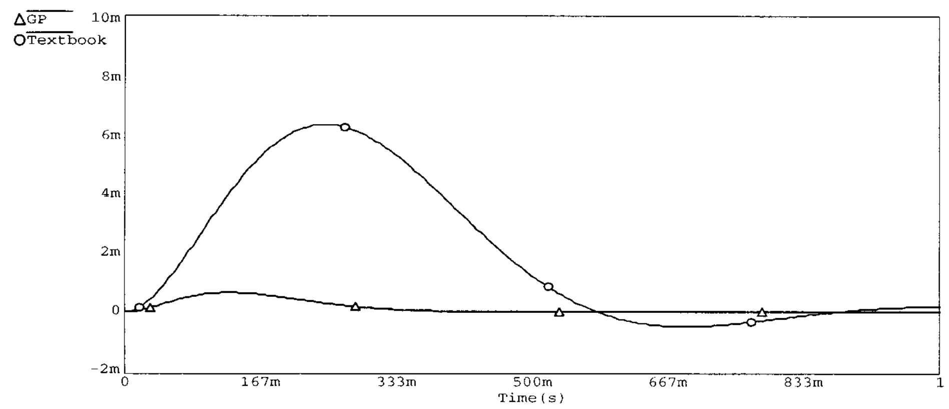

*Figure 6.* Comparison of the time-domain response of the genetically evolved controller (triangles) and the Bishop and Dorf controller (circles) to a 1-volt disturbance signal with $K = 1$ and $\tau = 1$ for the two-lag plant.

controller from generation 32 and the Dorf and Bishop controller. As can be seen in the table, the best-of-run controller from generation 32 is 2.42 times better than the textbook controller as measured by the integral of the time-weighted absolute error, has only 64% of the rise time in response to the reference input, has only 32% of the settling time, and is 8.97 times better in terms of suppressing the effects of disturbance at the plant input. Both controllers have approximately the same bandwidth (i.e., around 1 Hz), with the genetically evolved controller being inferior.

The above results can be compared to the results for the Dorf and Bishop controller by structuring the entire system as a prefilter and compensator. Figure 7 presents a model for the entire system that is helpful in making such comparisons with previous work. In this figure, the reference signal $R(s)$ is fed through prefilter $G_p(s)$. The plant output $Y(s)$ is passed through $H(s)$ and then subtracted from the prefiltered reference signal and the difference (error) is fed into the compensator $G_c(s)$. The plant $G(s)$ has one input and one output $Y(s)$. $G_c(s)$ has one input (the difference) and one output $U(s)$. Disturbance $D(s)$ may be added to the output $U(s)$ of $G_c(s)$. The resulting sum is subjected to a limiter (in the range between $-40$ and $+40$ volts for this problem).

*Table 3.* Comparison for the two-lag plant

| | Units | Genetically evolved controller | Dorf and Bishop [28] |
|---|---|---|---|
| Disturbance sensitivity | $\mu$Volts/Volt | 644 | 5,775 |
| ITAE | millivolt sec$^2$ | 19 | 46 |
| Bandwidth (3 dB) | Hz | 1.5 | 1 |
| Rise time | milliseconds | 296 | 465 |
| Settling time | milliseconds | 304 | 944 |

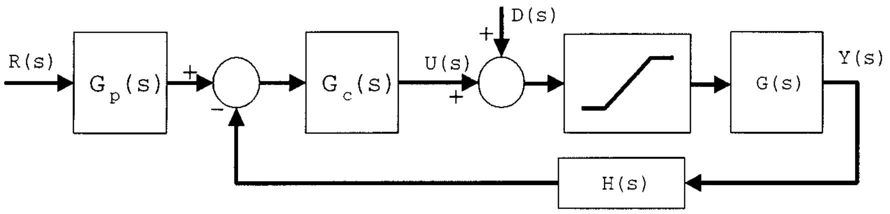

*Figure 7.* Overall model used for comparing genetically evolved results with results of Dorf and Bishop.

The transfer function for the prefilter, $G_{p\text{-}dorf}(s)$, of the Dorf and Bishop controller is

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

and the transfer function for their PID compensator, $G_{c\text{-}dorf}(s)$, is

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

After applying standard block diagram manipulations, the transfer function for the best-of-run controller from generation 32 for the two-lag plant can be expressed as a transfer function for a prefilter and a transfer function for a compensator. The transfer function for the prefilter, $G_{p32}(s)$, for the best-of-run individual from generation 32 for the two-lag plant is

$$G_{p32}(s) = \frac{1(1 + .1262s)(1 + .2029s)}{(1 + .03851s)(1 + .05146s)(1 + .08375)(1 + .1561s)(1 + .1680s)}.$$

The transfer function for the compensator, $G_{c32}(s)$, for the best-of-run individual from generation 32 for the two-lag plant is

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

$$= \frac{7487.05 + 1300.63s + 71.2511s^2 + 1.2426s^3}{s}.$$

The $s^3$ term (in conjunction with the $s$ in the denominator) indicates a second derivative. Although derivatives may not be useful in control systems (where they may amplify high frequency effects such as noise), their use here is appropriate since there are no such effects in this problem. Thus, the compensator consists of a second derivative in addition to proportional, integrative, and derivative functions.

Harry Jones of The Brown Instrument Company of Philadelphia patented this same kind of controller topology in 1942 [33]. As Jones states,

> A ... specific object of the invention is to provide electrical control apparatus ... wherein the rate of application of the controlling medium may be effected in accordance with or in response to the first, second, and high derivatives of the magnitude of the condition with respect to time, as desired.

Claim 38 of the Jones 1942 patent [33] states,

> In a control system, an electrical network, means to adjust said network in response to changes in a variable condition to be controlled, control means responsive to network adjustments to control said condition, reset means including a reactance in said network adapted following an adjustment of said network by said first means to initiate an additional network adjustment in the same sense, and rate control means included in said network adapted to control the effect of the first mentioned adjustment in accordance with the second or higher derivative of the magnitude of the condition with respect to time.

Because the best-of-run individual from generation 32 has proportional, integrative, derivative, and second derivative blocks, it infringes on the (now-expired) 1942 Jones patent [33].

The legal criteria for obtaining a U.S. patent are that the proposed invention be "new" and "useful" and

> ... the differences, between the subject matter sought to be patented and the prior art are such that the subject matter as a whole would [not] have been obvious at the time the invention was made to a person having ordinary skill in the art to which said subject matter pertains. (35 *United States Code* 103a).

The criteria for obtaining patents in Great Britain and elsewhere are broadly similar. Since filing for a patent entails the expenditure of a considerable amount of time and money, patents are generally sought only if an individual or business believes the inventions are potentially useful in the real world. Patents are only issued if an arms-length examiner is convinced that the proposed invention is novel, useful, and satisfies the statutory test for unobviousness.

Note that the user of genetic programming did not preordain, prior to the run (as part of the preparatory steps for genetic programming), that a second derivative should be used in the controller. The evolutionary process discovered that a second derivative was helpful in producing a better value of fitness in the automatically created controller. That is, necessity was the mother of invention. Similarly, the user did not preordain any particular topological arrangement of proportional, integrative, derivative, second derivative, or other functions within the automatically created controller. Instead, genetic programming automatically created a robust controller for a two-lag plant without the benefit of user-supplied information concerning the total number of processing blocks to be employed in the

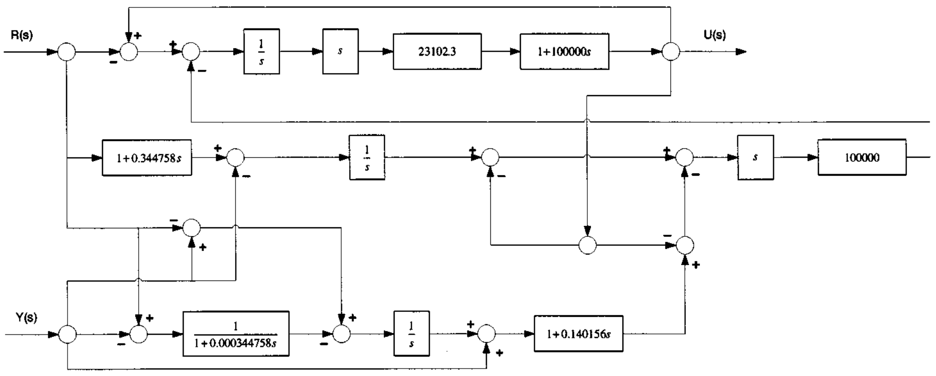

*Figure 8.* Best-of-run genetically evolved controller from generation 31 for the three-lag plant.

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 (none in this instance) within the controller.

## 7. Results for the three-lag plant

The best individual from generation 0 of our one (and only) run of this problem has a fitness of 14.35.

The best-of-run individual emerges in generation 31 and has a near-zero fitness of 1.14.

Figure 8 shows this individual controller in the form of a block diagram. $R(s)$ is the reference signal, $Y(s)$ is the plant output, and $U(s)$ is the controller's output (control variable).

Table 4 compares the average performance for five metrics of the genetically evolved best-of-run controller from generation 31 for the three-lag plant and the PID controller presented in Astrom and Hagglund [10] over the eight combinations of values for $K$, $\tau$, and the step size of the reference signal. The system bandwidth is the frequency of the reference signal above which the plant's output is attenuated by at least a specified degree (3 dB here) in comparison to the plant's output

*Table 4.* Comparison of average characteristics for the three-lag plant

| | Units | Genetically evolved controller | PID controller |
|---|---|---|---|
| Disturbance sensitivity | $\mu$volts/volt | 4.3 | 180,750 |
| ITAE | millivolt seconds$^2$ | 0.142 | 2.2 |
| Bandwidth (3 dB) | Hertz | 0.72 | 0.21 |
| Rise time | milliseconds | 0.77 | 2.8 |
| Settling time | milliseconds | 1.23 | 5.5 |

at a specified lower frequency (e.g., DC or very low frequencies). As can be seen, the genetically evolved best-of-run controller from generation 31 has superior average values for ITAE, disturbance sensitivity, rise time, and settling time (and an acceptable, although inferior, value for bandwidth).

Astrom and Hagglund [10] did not consider seven of the eight combinations of values for $K$, $\tau$, and the step size used in computing the averages, whereas we used all eight combinations of values in our run. Accordingly, Table 5 compares the performance of the best-of-run controller from generation 31 for the three-lag plant and the Astrom and Hagglund PID controller for the specific value of plant internal gain, $K$, of 1.0 used by Astrom and Hagglund, the specific value of the plant time constant $\tau$, of 1.0 used by Astrom and Hagglund, and the specific step size of the reference signal (1.0 volts) used by Astrom and Hagglund.

As can be seen in Table 5, the best-of-run genetically evolved controller from generation 31 is 7.2 times better than the textbook controller as measured by the integral of the time-weighted absolute error, has only 50% of the rise time in response to the reference input, has only 35% of the settling time, and is 92.7 dB better in terms of suppressing the effects of disturbance at the plant input. The genetically evolved controller has 2.9 times the bandwidth of the PID controller.

For all eight combinations of values for $K$, $\tau$, and the step size of the reference signal, the genetically evolved best-of-run controller from generation 31 has superior values for ITAE, disturbance sensitivity, rise time, and settling time.

Figure 9 compares the time-domain response of the best-of-run genetically evolved controller (squares) from generation 31 for a 1 volt unit step with $K = 1$ and $\tau = 1$ with the time-domain response of the controller (circles) presented in Astrom and Hagglund [10]. The faster rising curve in the figure shows the performance of the genetically evolved controller.

The rise time is the time required for the plant output to first reach a specified percentage (90% here) of the reference signal. As can be seen in the figure, the rise time for the best-of-run genetically evolved controller from generation 31 is 1.25 seconds. This is 50% of the 2.49-second rise time for the Astrom and Hagglund [10] controller.

The settling time is the first time for which the plant response reaches and stays within a specified percentage (2% here) of the reference signal. The settling time for the best-of-run genetically evolved controller from generation 31 is 1.87

*Table 5.* Comparison of characteristics for $K = 1.0$, $\tau = 1.0$, and step size of 1.0 for the three-lag plant

| | Units | Genetically evolved controller | PID controller |
|---|---|---|---|
| Disturbance sensitivity | $\mu$volts/volt | 4.3 | 186,000 |
| ITAE | volt seconds$^2$ | 0.360 | 2.6 |
| Bandwidth (3 dB) | Hertz | 0.72 | 0.248 |
| Rise time | seconds | 1.25 | 2.49 |
| Settling time | seconds | 1.87 | 6.46 |

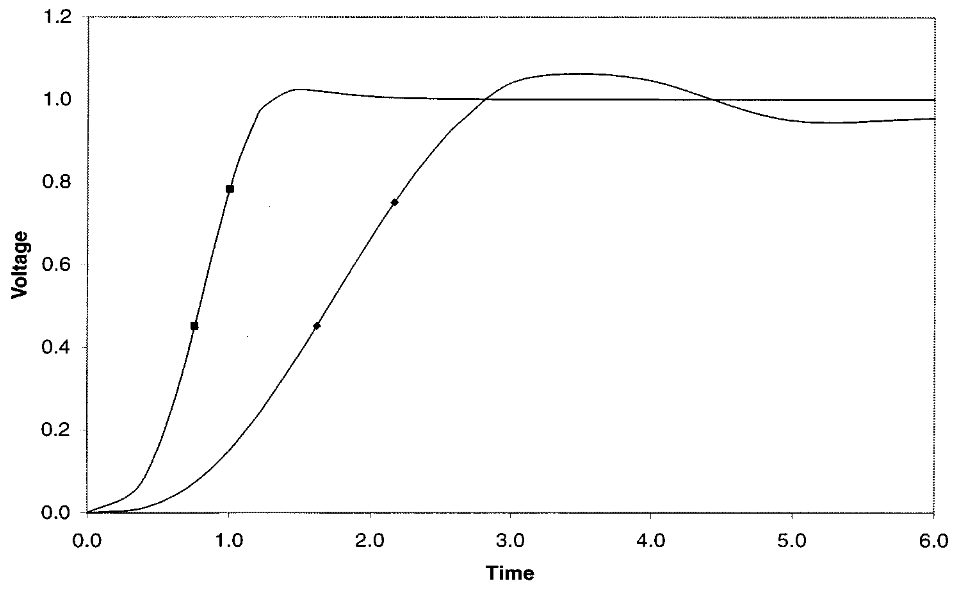

*Figure 9.* Comparison of the time-domain response of the best-of-run genetically evolved controller (squares) from generation 31 and the Astrom and Hagglund controller (circles) for a 1 volt unit step with $K = 1$ and $\tau = 1$ for the three-lag plant problem.

seconds. That is, the best-of-run genetically evolved controller from generation 31 settles in 35% of the 6.46-second settling time for the Astrom and Hagglund [10] controller.

The overshoot is the percentage by which the plant response exceeds the reference signal. The best-of-run genetically evolved controller from generation 31 reaches a maximum value of 1.023 at 1.48 seconds (i.e., has a 2.3% overshoot). The Astrom and Hagglund [10] controller reaches a maximum value of 1.074 at 3.47 seconds (i.e., a 7.4% overshoot).

After applying standard block diagram manipulations, the transfer function for the best-of-run controller from generation 31 for the three-lag plant can be expressed as a transfer function for a prefilter and a transfer function for a compensator. The transfer function for the prefilter, $G_{p31}(s)$, for the best-of-run individual from generation 31 for the three-lag plant is

$$G_{p31}(s) = \frac{(1 + 0.2083s)(1 + 0.0002677s)}{(1 + 0.000345s)}.$$

The transfer function for the compensator, $G_{c31}(s)$, for the best-of-run individual from generation 31 for the three-lag plant is

$$G_{c31}(s) = 300{,}000.$$

The feedback transfer function, $H_{31}(s)$, for the best-of-run individual from generation 31 for the three-lag plant is

$$H_{31}(s) = 1 + 0.42666s + 0.046703s^2.$$

## 8. Conclusion

This paper has demonstrated that genetic programming can be used to automatically create both the parameter values (tuning) and the topology for controllers for illustrative problems involving a two-lag plant and a three-lag plant.

For both problems, genetic programming made the decisions concerning the total number of signal processing blocks to be employed in the controller, the type of each signal processing block, the topological interconnections between the signal processing blocks and external input and output points of the controller, the values of all parameters for the signal processing blocks, and the existence, if any, of internal feedback between the signal processing blocks within the controller. Genetic programming simultaneously optimized a prespecified performance metric (i.e., minimizing the time required to bring the plant outputs to the desired values as measured by the integral of the time-weighted absolute error), satisfied time-domain constraints (involving, variously, overshoot, disturbance rejection, and stability), incorporated frequency-domain constraints (e.g., bandwidth), and satisfied robustness requirements.

The two genetically evolved controllers are better than controllers designed and published by experts in the field of control using their own criteria.

The run of the two-lag plant problem rediscovered a previously patented controller topology involving a second derivative. Intermediate results from the runs of both illustrative problems approximately duplicated the functionality of previously patented controller topologies (e.g., PI and PID controllers).

The preparatory steps for both problems were straightforward and uncomplicated translations of the statement of the problem into the requirements for the input to a run of genetic programming. Both genetically evolved controllers were evolved on the first and only run of the problem. These two facts suggest that there is considerable future potential for applying genetic programming to problems of automatically synthesizing the design of more demanding controllers.

More broadly, the results in this paper (and the other human-competitive results in Table 1) suggest that genetic programming is on the threshold of routinely producing human-competitive results. We expect that the rapidly decreasing cost of computing power will enable genetic programming to deliver additional human-competitive results on increasingly difficult problems and, in particular, that genetic programming will be routinely used as an “invention machine” for producing patentable new inventions.

## References

1. E. E. Altshuler and D. S. Linden, Process for the Design of Antennas using Genetic Algorithm, United States Patent 5,719,794, 1998. Applied for on July 19, 1995. Issued on February 17, 1998.
2. B. Andersson, P. Svensson, P. Nordin, and M. Nordahl, "Reactive and memory-based genetic programming for robot control," in Genetic Programming: Second European Workshop. EuroGP'99. Proceedings, Lecture Notes in Computer Science, R. Poli, P. Nordin, W. B. Langdon, and T. C. Fogarty (eds.), Springer-Verlag, Berlin, Germany, 1991, vol. 1598, pp. 161-172.
3. D. Andre, F. H. Bennett, III, and J. R. Koza, "Discovery by genetic programming of a cellular automata rule that is better than any known rule for the majority classification problem," in Genetic Programming 1996: Proceedings of the First Annual Conference, Stanford University, July 28–31, 1996, J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo (eds.), MIT Press, Cambridge, MA, 1996, pp. 3-11.
4. D. Andre and A. Teller, "Evolving team Darwin United," in RoboCup-98: Robot Soccer World Cup II, Lecture Notes in Computer Science, M. Asada, and H. Kitano (eds.), Springer-Verlag, Berlin, 1999, vol. 1604, pp. 346-352.
5. P. J. Angeline, "An alternative to indexed memory for evolving programs with explicit state representations," in Genetic Programming 1997: Proceedings of the Second Annual Conference, July 13–16, 1997, Stanford University, J. R. Koza, K. Deb, M. Dorigo, D. B. Fogel, M. Garzon, H. Iba, and R. L. Riolo (eds.), Morgan Kaufmann, San Francisco, CA, 1997, pp. 423-430.
6. P. J. Angeline, "Multiple interacting programs: A representation for evolving complex behaviors," Cybernetics and Systems vol. 29(8) pp. 779-806, 1998.
7. P. J. Angeline, "Evolving predictors for chaotic time series," in Proceedings of SPIE (Volume 3390): Application and Science of Computational Intelligence, S. Rogers, D. Fogel, J. Bezdek, and B. Bosacchi (eds.), SPIE—The International Society for Optical Engineering, Bellingham, WA, 1998, pp. 170-180.
8. P. J. Angeline and D. B. Fogel, "An evolutionary program for the identification of dynamical systems," in Proceedings of SPIE (Volume 3077): Application and Science of Artificial Neural Networks III, S. Rogers (ed.), SPIE—The International Society for Optical Engineering, Bellingham, WA, 1997, pp. 409-417.
9. P. J. Angeline and K. E. Kinnear, Jr. (eds.). Advances in Genetic Programming 2, The MIT Press: Cambridge, MA, 1996.
10. K. J. Astrom and T. Hagglund, PID Controllers: Theory, Design, and Tuning, second edition, Instrument Society of America: Research Triangle Park, NC, 1995.
11. W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith (eds.), GECCO-99: Proceedings of the Genetic and Evolutionary Computation Conference, July 13–17, 1999, Orlando, Florida, Morgan Kaufmann, San Francisco, CA, 1999.
12. W. Wolfgang, P. Nordin, R. E. Keller, and F. D. Francone, Genetic Programming—An Introduction, Morgan Kaufmann, San Francisco, CA and dpunkt, Heidelberg, 1998.
13. W. Banzhaf, P. Nordin, R. Keller, and M. Olmer, "Generating adaptive behavior for a real robot using function regression with genetic programming," in Genetic Programming 1997: Proceedings of the Second Annual Conference, July 13–16, 1997, Stanford University, J. R. Koza, K. Deb, M. Dorigo, D. B. Fogel, M. Garzon, H. Iba, and R. L. Riolo (eds.), Morgan Kaufmann, San Francisco, CA, 1997, pp. 35-43.
14. W. Banzhaf, R. Poli, M. Schoenauer, and T. C. Fogarty, Genetic Programming: First European Workshop, EuroGP'98, Paris, France, April 1998 Proceedings, April 1998, Paris, France, Lecture Notes in Computer Science, Springer-Verlag, Berlin, Germany, 1998, vol. 1391.
15. F. H Bennett, III, J. R. Koza, M. A. Keane, J. Yu, W. Mydlowec, and O. Stiffelman, "Evolution by means of genetic programming of analog circuits that perform digital functions," in GECCO-99: Proceedings of the Genetic and Evolutionary Computation Conference, July 13–17, 1999, Orlando, Florida, W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith (eds.), Morgan Kaufmann, San Francisco, CA, 1999, pp. 1477-1483.
16. F. H Bennett, III, J. R. Koza, J. Shipman, and O. Stiffelman, "Building a parallel computer system for $18,000 that performs a half peta-flop per day," in GECCO-99: Proceedings of the Genetic and

Evolutionary Computation Conference, July 13–17, 1999, Orlando, Florida, W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith (eds.), Morgan Kaufmann, San Francisco, CA, 1999, pp. 1484-1490.

17. S. P. Boyd, and C. H. Barratt, Linear Controller Design: Limits of Performance, Prentice Hall: Englewood Cliffs, NJ, 1991.
18. A. E. Bryson and Y-C. Ho, Applied Optimal Control, Hemisphere Publishing: New York, 1975.
19. A. Callender, and A. B. Stevenson, 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, 1939.
20. G. A. Campbell, Electric Wave Filter. U.S. Patent 1,227,113. Filed July 15, 1915. Issued May 22, 1917, 1917.
21. W. Cauer, Artificial Network. U.S. Patent 1,958,742. Filed June 8, 1928 in Germany. Filed December 1, 1930 in United States. Issued May 15, 1934, 1934.
22. W. Cauer, Electric Wave Filter. U.S. Patent 1,989,545. Filed June 8, 1928. Filed December 6, 1930 in United States. Issued January 29, 1935, 1935.
23. W. Cauer, Unsymmetrical Electric Wave Filter. Filed November 10, 1932 in Germany. Filed November 23, 1933 in United States. Issued July 21, 1936, 1936.
24. K. Chellapilla, and D. B. Fogel, "Evolution, neural networks, games, and intelligence," Proceedings of the IEEE vol. 87(9) pp. 1471-1496, 1999.
25. L. S. Crawford, V. H. L. Cheng, and P. K. Menon, "Synthesis of flight vehicle guidance and control laws using genetic search methods," in Proceedings of 1999 Conference on Guidance, Navigation, and Control, American Institute of Aeronautics and Astronautics, Reston, VA, Paper AIAA-99-4153, 1999.
26. S. Darlington, Semiconductor Signal Translating Device, U.S. Patent 2,663,806. Filed May 9, 1952. Issued December 22, 1953, 1953.
27. L. D. Dewell, and P. K. Menon, "Low-thrust orbit transfer optimization using genetic search," in Proceedings of 1999 Conference on Guidance, Navigation, and Control, American Institute of Aeronautics and Astronautics, Reston, VA, Paper AIAA-99-4151, 1999.
28. R. C. Dorf, and R. H. Bishop, Modern Control Systems, eighth edition, Addison-Wesley: Menlo Park, CA, 1998.
29. L. J. Fogel, A. J. Owens, and M. J. Walsh, Artificial Intelligence through Simulated Evolution, John Wiley: New York, 1966.
30. T. Higuchi, T. Niwa, T. Tanaka, H. Iba, H. de Garis, and T. Furuya, "Evolving hardware with genetic learning: A first step towards building a Darwin machine," in From Animals to Animats 2: Proceedings of the Second International Conference on Simulation of Adaptive Behavior, J.-A. Meyer, H. L. Roitblat, and S. W. Wilson (eds.), The MIT Press, Cambridge, MA, 1993, pp. 417-424.
31. J. H. Holland, Adaptation in Natural and Artificial Systems, University of Michigan Press: Ann Arbor, MI, 1975.
32. J. H. Holland, K. J. Holyoak, R. E. Nisbett, and P. A. Thagard, Induction: Processes of Inference, Learning, and Discovery, The MIT Press: Cambridge, MA, 1986.
33. H. S. Jones, Control Apparatus. United States Patent 2,282,726. Filed October 25, 1939. Issued May 12, 1942, 1942.
34. H. Juille, "Evolution of non-deterministic incremental algorithms as a new approach for search in state spaces," in Proceedings of the Sixth International Conference on Genetic Algorithms, L. J. Eshelman (ed.), Morgan Kaufmann, San Francisco, CA, 1995, pp. 351-358.
35. H. Juille, and J. B. Pollack, "Coevolving the "ideal" trainer: Application to the discovery of cellular automata rules," in Genetic Programming 1998: Proceedings of the Third Annual Conference, July 22–25, 1998, University of Wisconsin, Madison, Wisconsin, J. R. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D. B. Fogel, M. H. Garzon, D. E. Goldberg, H. Iba, and R. L. Riolo (eds.), Morgan Kaufmann, San Francisco, CA, 1998, pp. 519-527.
36. K. E. Kinnear, Jr. (ed.), Advances in Genetic Programming, The MIT Press: Cambridge, MA, 1994.
37. J. R. Koza, "Hierarchical genetic algorithms operating on populations of computer programs," in Proceedings of the 11th International Joint Conference on Artificial Intelligence, Morgan Kaufmann: San Mateo, CA, vol. 1, 1989, pp. 768-774.

38. J. R. Koza, Genetic Programming: On the Programming of Computers by Means of Natural Selection, MIT Press: Cambridge, MA, 1992.
39. J. R. Koza, Genetic Programming II: Automatic Discovery of Reusable Programs, MIT Press: Cambridge, MA, 1994.
40. J. R. Koza, Genetic Programming II Videotape: The Next Generation, MIT Press: Cambridge, MA, 1994.
41. J. R. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D. B. Fogel, M. H. Garzon, D. E. Goldberg, H. Iba, and R. Riolo (eds.), Genetic Programming 1998: Proceedings of the Third Annual Conference, Morgan Kaufmann, San Francisco, CA, 1998.
42. J. R. Koza and F. H Bennett, III, Automatic Synthesis, Placement, and Routing of Electrical Circuits by Means of Genetic Programming," in Advances in Genetic Programming 3, L. Spector, W. B. Langdon, U.-M. O'Reilly, and P. Angeline (eds.), The MIT Press, Cambridge, MA, 1999, chap. 6, pp. 105-134.
43. J. R. Koza, F. H Bennett, III, D. Andre, and M. A. Keane, Genetic Programming III: Darwinian Invention and Problem Solving, Morgan Kaufmann: San Francisco, CA, 1999.
44. J. R. Koza, F. H Bennett, III, M. A. Keane, J. Yu, W. Mydlowec, and O. Stiffelman, "Searching for the impossible using genetic programming," in GECCO-99: Proceedings of the Genetic and Evolutionary Computation Conference, July 13–17, 1999, Orlando, Florida, W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith (eds.), Morgan Kaufmann, San Francisco, CA, 1999, pp. 1083-1091.
45. J. R. Koza, F. H Bennett, III, D. Andre, M. A. Keane, and S. Brave, Genetic Programming III Videotape: Human-Competitive Machine Intelligence, Morgan Kaufmann: San Francisco, CA, 1999, forthcoming.
46. J. R. Koza, K. Deb, M. Dorigo, D. B. Fogel, M. Garzon, H. Iba, and R. L. Riolo (eds.), Genetic Programming 1997: Proceedings of the Second Annual Conference, Morgan Kaufmann, San Francisco, CA, 1997.
47. J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo (eds.), Genetic Programming 1996: Proceedings of the First Annual Conference, The MIT Press, Cambridge, MA, 1996.
48. J. R. Koza and J. P. Rice, Genetic Programming: The Movie, MIT Press: Cambridge, MA, 1992.
49. I. M. Kroo, J. H. McMasters, and R. J. Pavek, Large Airplane with Nonplanar Wing. U.S. Design Patent number USD0363696. Applied for on June 23, 1993. Issued October 31, 1995, 1995.
50. W. B. Langdon, Genetic Programming and Data Structures: Genetic Programming + Data Structures = Automatic Programming!, Kluwer Academic Publishers: Amsterdam, 1998.
51. K. F. Man, K. S. Tang, S. Kwong, and W. A. Halang, Genetic Algorithms for Control and Signal Processing, Springer-Verlag: London, 1997.
52. K. F. Man, K. S. Tang, S. Kwong, and W. A. Halang, Genetic Algorithms: Concepts and Designs, Springer-Verlag: London, 1999.
53. P. Marenbach, K. D. Bettenhausen, and S. Freyer, "Signal path oriented approach for generation of dynamic process models," in Genetic Programming 1996: Proceedings of the First Annual Conference, July 28–31, 1996, Stanford University, J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo (eds.), MIT Press, Cambridge, MA, 1996, pp. 327-332.
54. P. K. Menon, M. Yousefpor, T. Lam, and M. L. Steinberg, "Nonlinear flight control system synthesis using genetic programming. Proceedings of 1995 Conference on Guidance, Navigation, and Control," in American Institute of Aeronautics and Astronautics, Reston, VA, 1995, pp. 461-470.
55. D. G. O'Connor and R. J. Nelson, Sorting System with N-Line Sorting Switch. United States Patent number 3,029,413. Issued April 10, 1962, 1962.
56. K. Ogata, Modern Control Engineering, third edition. Prentice Hall: Upper Saddle River, NJ, 1997.
57. G. A. Philbrick, Delayed Recovery Electric Filter Network. Filed May 18, 1951. U.S. Patent 2,730,679. Issued January 10, 1956, 1956.
58. R. Poli, P. Nordin, W. B. Langdon, and T. C. Fogarty, Genetic Programming: Second European Workshop, EuroGP99, Goteborg, Sweden, May 1999, Proceedings, Lecture Notes in Computer Science, Springer-Verlag, Berlin, Germany, 1999, vol. 1598.

59. T. Quarles, A. R. Newton, D. O. Pederson, and A. Sangiovanni-Vincentelli, SPICE 3 Version 3F5 User's Manual, Department of Electrical Engineering and Computer Science, University of California, Berkeley, CA, March 1994.
60. I. Rechenberg, Cybernetic solution path of an experimental problem, Royal Aircraft Establishment, Ministry of Aviation, Library Translation 1112, Farnborough, 1965.
61. I. Rechenberg, Evolutionsstrategie: Optimierung Technischer Systeme nach Prinzipien der Biolgischen Evolution, Verlag Frommann-Holzboog: Stuttgart-Bad Cannstatt, 1973.
62. C. Ryan, Automatic Re-engineering of Software Using Genetic Programming, Kluwer Academic Publishers: Amsterdam, 1999.
63. A. L. Samuel, "Some studies in machine learning using the game of checkers," IBM Journal of Research and Development vol. 3(3) pp. 210-229, 1959.
64. A. L. Samuel, "AI: Where it has been and where it is going," in Proceedings of the Eighth International Joint Conference on Artificial Intelligence, Morgan Kaufmann, Los Altos, CA, 1983, pp. 1152-1157.
65. L. Spector, H. Barnum, and H. J. Bernstein, "Genetic programming for quantum computers," in Genetic Programming 1998: Proceedings of the Third Annual Conference, J. R. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D. B. Fogel, M. H. Garzon, D. E. Goldberg, H. Iba, and R. Riolo (eds.), Morgan Kaufmann, San Francisco, CA, 1998, pp. 365-373.
66. L. Spector, H. Barnum, and H. J. Bernstein, "Quantum computing applications of genetic programming," in Advances in Genetic Programming 3, L. Spector, W. B. Langdon, U.-M. O'Reilly, and P. Angeline (eds.), The MIT Press, Cambridge, MA, 1999, pp. 135-160.
67. L. Spector, H. Barnum, H. J. Bernstein, and N. Swamy, "Finding a better-than-classical quantum AND/OR algorithm using genetic programming," in IEEE. Proceedings of 1999 Congress on Evolutionary Computation, IEEE Press, Piscataway, NJ, 1999, pp. 2239-2246.
68. L. Spector, W. B. Langdon, U.-M. O'Reilly, and P. Angeline (eds.), Advances in Genetic Programming 3, The MIT Press: Cambridge, MA, 1999.
69. T. L. Sterling, J. Salmon, D. J. Becker, and D. F. Savarese, How to Build a Beowulf: A Guide to Implementation and Application of PC Clusters, MIT Press: Cambridge, MA, 1999.
70, G. D. Sweriduk, P. K. Menon, and M. L. Steinberg, "Robust command augmentation system design using genetic search methods," in Proceedings of 1998 Conference on Guidance, Navigation, and Control, American Institute of Aeronautics and Astronautics, Reston, VA, 1998, pp. 286-294.
71. G. D. Sweriduk, P. K. Menon, and M. L. Steinberg, "Design of a pilot-activated recovery system using genetic search methods," in Proceedings of 1998 Conference on Guidance, Navigation, and Control, American Institute of Aeronautics and Astronautics, Reston, VA, 1999.
72. A. Teller, Evolving Programmers: SMART Mutation, Computer Science Department, Carnegie Mellon University, Technical Report CMU-CS-96, 1996.
73. A. Teller, "Evolving programmers: The co-evolution of intelligent recombination operators," Advances in Genetic Programming 2, P. J. Angeline and K. E. Kinnear, Jr. (eds.), The MIT Press: Cambridge, MA, 1996.
74. A. Teller, "Algorithm Evolution with Internal Reinforcement for Signal Understanding," Computer Science Department, Carnegie Mellon University, Pittsburgh, Pennsylvania, PhD Thesis, 1998.
75. A. Teller, "The internal reinforcement of evolving algorithms," Advances in Genetic Programming 3, L. Spector, W. B. Langdon, U.-M. O'Reilly, and P. Angeline (eds.), The MIT Press: Cambridge, MA, 1999.
76. A. Teller and M. Veloso, Learning Tree Structured Algorithms for Orchestration into an Object Recognition System, Computer Science Department, Carnegie Mellon University, Technical Report CMU-CS-95-101, 1995.
77. A. Teller and M. Veloso, "Program evolution for data mining," in Special Issue on Genetic Algorithms and Knowledge Bases, S. Louis (ed.), The International Journal of Expert Systems, JAI Press (3) pp. 216-236, 1995.
78. A. Teller and M. Veloso, "A controlled experiment: evolution for learning difficult problems," in Proceedings of Seventh Portuguese Conference on Artificial Intelligence, Springer-Verlag, 1995, pp. 165-176.

79. A. Teller and M. Veloso, “Algorithm Evolution for Face Recognition: What Makes a Picture Difficult?,” in Proceedings of the IEEE International Conference on Evolutionary Computation, IEEE Press, 1995.
80. A. Teller and M. Veloso, “Language Representation Progression in PADO,” in Proceedings of AAAI Fall Symposium on Artificial Intelligence, AAAI Press, Menlo Park, CA, 1995.
81. A. Teller and M. Veloso, “PADO: A new learning architecture for object recognition,” Symbolic Visual Learning, K. Ikeuchi, and M. Veloso (eds.), Oxford University Press, 1996, pp. 81-116.
82. A. Teller and M. Veloso, “Neural programming and an internal reinforcement policy,” Simulated Evolution and Learning, Lecture Notes in Artificial Intelligence, X. Yao, J.-H. Kim, and T. Furuhashi (eds.), Springer-Verlag: Heidelberg, Germany, 1997, vol. 1285, pp. 279-286.
83. A. M. Turing, “Intelligent machines. Pages 21-23,” in Mechanical Intelligence: Collected Works of A. M. Turing, D. C. Ince (ed.), North-Holland: Amsterdam, 1948, pp. 107-128.
84. A. M. Turing, “Computing machinery and intelligence,” Mind vol. 59(236) pp. 433-460. Reprinted in Mechanical Intelligence: Collected Works of A. M. Turing, D. C. Ince (ed.), Amsterdam, North-Holland, 1992, pp. 133-160.
85. D. Whitley, F. Gruau, and L. Preatt, “Cellular encoding applied to neurocontrol,” in Proceedings of the Sixth International Conference on Genetic Algorithms, L. J. Eshelman (ed.), Morgan Kaufmann, San Francisco, CA, 1995, pp. 460-467.
86. O. J. Zobel, Wave Filter. U.S. Patent 1,538,964. Filed January 15, 1921. Issued May 26, 1925, 1925.