Automatic Synthesis of Both the Topology and Parameters for a Controller for a Three-Lag Plant with a Five-Second Delay using Genetic Programming

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

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

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

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

Forrest H Bennett III
Genetic Programming Inc.
(Currently, FX Palo Alto Laboratory, Palo Alto, California)
forrest@evolute.com

Abstract

This paper describes how the process of synthesizing the design of both the topology and the numerical parameter values (tuning) for a controller can be automated by using genetic programming. Genetic programming can be used to automatically make the decisions concerning the total number of signal processing blocks to be employed in a controller, the type of each block, the topological interconnections between the blocks, and the values of all parameters for all blocks requiring parameters. In synthesizing the design of controllers, genetic programming can simultaneously optimize prespecified performance metrics (such as minimizing the time required to bring the plant output to the desired value), satisfy time-domain constraints (such as overshoot and disturbance rejection), and satisfy frequency domain constraints. Evolutionary methods have the advantage of not being encumbered by preconceptions that limit its search to well-traveled paths. Genetic programming is applied to an illustrative problem involving the design of a controller for a three-lag plant with a significant (five-second) time delay in the external feedback from the plant to the controller. A delay in the feedback makes the design of an effective controller especially difficult.

1 Introduction

The process of creating (synthesizing) the design of a controller entails making decisions concerning the total number of processing blocks to be employed in the controller, the type of each signal processing block (e.g., lead, lag, gain, integrator, differentiator, adder, inverter, subtractor, and multiplier), the values of all parameters for all blocks requiring parameters, and the topological interconnections between the signal processing blocks. The latter includes the question of whether or not to employ internal feedback (i.e., feedback inside the controller).

The problem of synthesizing a controller to satisfy prespecified requirements is sometimes solvable by analytic techniques (often oriented toward producing conventional PID controllers). However, as Boyd and Barratt stated in *Linear Controller Design: Limits of Performance* (1991),

> "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 …"

This paper describes how genetic programming can be used to automatically create both the topology and the numerical parameter values (i.e., the tuning) for a controller directly from a high-level statement of the requirements of the controller. Genetic programming can, if desired, 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), and satisfy frequency domain constraints. Evolutionary methods have the advantage of not being encumbered by preconceptions that limit their search to well-traveled paths.

Section 2 describes an illustrative problem of controller synthesis. Section 3 provides general background on genetic programming. Section 4 describes how genetic programming is applied to control problems. Section 5 describes the preparatory steps necessary to apply genetic programming to the illustrative control problem. Section 6 presents the results.

2 Illustrative Problem

The illustrative problem entails creation of both the topology and parameter values for a controller for a three-lag plant with a significant (five-second) time delay in the external feedback from the plant output to the controller such that plant output reaches the level of the reference signal in minimal time (as measured by the integral of the time-weighted absolute error), 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 disturbance (added into the controller output). The delay in the feedback makes the design of an effective controller especially difficult (Astrom and Hagglund 1995). The transfer function of the plant is

$$G(s) = \frac{Ke^{-5s}}{(1+\tau s)^3}$$

A controller presented in Astrom and Hagglund 1995 (page 225) delivers credible performance on this problem for values of $K = 1$ and $\tau = 1$.

To make the problem more realistic, we added an additional constraint (satisfied by the controller presented in Astrom and Hagglund 1995) that the input to the plant is limited to the range between -40 and +40 volts. The plant in this paper operates over several different combinations of values for K and τ (whereas the controller designed by Astrom and Hagglund was intended only for $K = 1$ and $\tau = 1$).

3 Background on Genetic Programming

Genetic programming is an automatic technique for generating computer programs to solve, or approximately solve, problems.

Genetic programming (Koza 1992; Koza and Rice 1992) is an extension of the genetic algorithm (Holland 1975). Genetic programming is capable (Koza 1994a, 1994b) of evolving reusable, parametrized, hierarchically-called automatically defined functions (ADFs) so that an overall program consists of a main result-producing branch and one or more reusable and parameterizable automatically defined functions (function-defining branches). In addition, architecture-altering operations (Koza, Bennett, Andre, and Keane 1999; Koza, Bennett, Andre, Keane, and Brave 1999) enable genetic programming to automatically determine the number of automatically defined functions, the number of arguments that each possesses, and the nature of the hierarchical references, if any, among such automatically defined functions.

Genetic programming often creates novel designs because it is a probabilistic process that is not encumbered by the preconceptions that often channel human thinking down familiar paths. For example, genetic programming is capable of synthesizing the design of both the topology and sizing for a wide variety of analog electrical circuits from a high-level statement of the circuit's desired behavior and characteristics (Koza, Bennett, Andre, and Keane 1999; Koza, Bennett, Andre, Keane, and Brave 1999). Five of the evolved analog circuits in that book infringe on previously issued patents while five others deliver the same functionality as previously patented inventions in a novel way.

Additional information on current research in genetic programming can be found in Banzhaf, Nordin, Keller, and Francone 1998; Langdon 1998; Ryan 1999; Kinnear 1994; Angeline and Kinnear 1996; Spector, Langdon, O'Reilly, and Angeline 1999; Koza, Goldberg, Fogel, and Riolo 1996; Koza, Deb, Dorigo, Fogel, Garzon, Iba, and Riolo 1997; Koza, Banzhaf, Chellapilla, Deb, Dorigo, Fogel, Garzon, Goldberg, Iba, and Riolo 1998; Banzhaf, Poli, Schoenauer, and Fogarty 1998; Banzhaf, Daida, Eiben, Garzon, Honavar, Jakiela, and Smith 1999; Poli, Nordin, Langdon, and Fogarty 1999; at web sites such as `www.genetic-programming.org`; and in the *Genetic Programming and Evolvable Machines* journal (from Kluwer Academic Publishers).

4 Genetic Programming and Control

Controllers can be represented by block diagrams in which the blocks represent signal processing functions, in which external points represent the controller's input(s) and output(s), and in which cycles in the block diagram correspond to internal feedback

inside the controller. Genetic programming can be extended to the problem of creating both the topology and parameter values for a controller by establishing a mapping between the program trees used in genetic programming and the block diagrams germane to controllers.

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

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

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

Genetic programming has recently been used to create a controller for a particular two-lag plant and a three-lag plant (Koza, Keane, Yu, Bennett, and Mydlowec 2000). Both of these genetically evolved controllers outperformed the controllers designed by experts in the field of control using the criteria originally specified by the experts.

5 Preparatory Steps

Six major preparatory steps are required before applying genetic programming: (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.

5.1 Program Architecture

Since there is one result-producing branch in the program tree for each output from the controller and this problem involves a one-output controller, each program tree has one result-producing branch. Each program tree in the initial random generation (generation 0) has no automatically defined functions. However, in subsequent generations, architecture-altering operations may insert and delete automatically defined functions (up to a maximum of five per program tree).

5.2 Terminal Set

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 requiring a parameter value. These numerical values initially range from -5.0 to +5.0. These numerical values are perturbed during the run by a Gaussian mutation operation that operates only on numerical values. Numerical

constants are later interpreted on a logarithmic scale so that they represent values in a range of 10 orders of magnitude (Koza, Bennett, Andre, and Keane 1999).

The remaining terminals are time-domain signals. The terminal set, T, for the result-producing branch and any automatically defined functions (except for the perturbable numerical values mentioned above) is

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

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

5.3 Function Set

The functions are signal processing functions that operate on time-domain signals (the terminals in T). The function set, F, for the result-producing branch and any automatically defined functions is

F = {GAIN, INVERTER, LEAD, LAG, LAG2, DIFFERENTIAL_INPUT_INTEGRATOR, DIFFERENTIATOR, ADD_SIGNAL, SUB_SIGNAL, ADD_3_SIGNAL, DELAY, ADF0, …, ADF4}.

ADF0, …, ADF4 denote automatically defined functions added during the run by architecture-altering operations.

The functionality of each of the above signal processing functions is suggested by their names and is described in detail in Koza, Keane, Yu, Bennett, and Mydlowec 2000.

5.4 Fitness

Genetic programming is a probabilistic 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. The fitness measure 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. 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 invoked) to produce an interconnected sequence of signal processing blocks — that is, a block diagram for the controller. A SPICE netlist is then constructed from the block diagram. The SPICE netlist for the resulting controller is wrapped inside an appropriate set of SPICE commands. The controller is then simulated using our modified version of the SPICE simulator. The 217,000-line SPICE3 simulator (Quarles, Newton, Pederson, and Sangiovanni-Vincentelli 1994) is an industrial-strength simulator. It is run as a submodule within our genetic programming system. The SPICE simulator returns tabular output and other information from which the fitness of the individual is then computed.

The fitness of a controller is measured using 13 elements consisting of 12 time-domain-based elements based on a modified integral of time-weighted absolute error (ITAE) and one time-domain-based element measuring disturbance rejection.

The fitness of an individual controller is the sum (i.e., linear combination) of the detrimental contributions of these 13 elements of the fitness measure. The smaller the sum, the better.

The first 12 elements of the fitness measure evaluate how quickly the controller causes the plant to reach the reference signal and the controller's success in avoiding overshoot. Two reference signals are used. The first reference signal is a step function that rises from 0 to 1 volts at t = 100 milliseconds while the second rises from 0 to 1 microvolts at t = 100 milliseconds. The two step functions are used to deal with the non-linearity caused by the limiter. Two values of the time constant, τ, are used (namely 0.5 and 1.0). Three values of K are used, namely 0.9, 1.0, and 1.1. Exposing genetic programming to different combinations of values of step size, K, and τ produces a robust controllers and also prevents genetic programming from engaging in pole elimination. For each of these 12 fitness cases, a transient analysis is performed in the time domain using the SPICE simulator. Table 1 shows the elements of the fitness measure in its left-most four columns.

The contribution to fitness for each of these 12 elements of the fitness measure is based on the integral of time-weighted absolute error (ITAE)

$$\int_{t=5}^{36} (t-5)\left|e(t)\right| A(e(t))BCdt \ .$$

Because of the built-in five-second time delay, the integration runs from time t = 5 seconds to t = 36 seconds. Here $e(t)$ is the difference (error) at time t between the delayed plant output and the reference signal. The integral of time-weighted absolute error penalizes differences that occur later more heavily than differences that occur earlier.

We modified the integral of time-weighted absolute error in four ways. First, we used a discrete approximation to the integral by considering 120 300-millisecond time steps between t = 5 to t = 36 seconds. Second, we multiplied each fitness case by the reciprocal of the amplitude of the reference signals so that both reference signals (1 microvolt and 1 volt) are equally influential. Specifically, B is a factor that is used to normalize the contributions associated with the two step functions. 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. Third, the integral contains an additional weight, A, that varies with $e(t)$. The function A weights all variation up to 102% of the reference signal by a factor of 1.0, and heavily penalizes overshoots over 2% by a factor 10.0. Fourth, the integral contains a special weight, C, which is 5.0 for the two fitness cases for which K = 1 and τ = 1, and 1.0 otherwise.

The 13th element of the fitness measure is based on disturbance rejection. The penalty is computed based on a time-domain analysis for 36.0 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_OUTPUT` at time t = 0 and the resulting disturbed signal is provided as input to the plant. The detrimental contribution to fitness is 500/36 times the time required to bring the plant output to within 20 millivolts of the reference signal of 0 volts (i.e., to reduce the effect to within 2% of

the 1-volt disturbance signal) assuming that the plant settles to within this range within 36 seconds. If the plant does not settle to within this range within 36 seconds, the detrimental contribution to fitness is 500 plus the absolute value of the plant output in volts times 500. For example, if the effect of the disturbance was never reduced below 1 volts, the detrimental contribution to fitness would be 1000.

A controller that cannot be simulated by SPICE is assigned a high penalty value of fitness (10^8).

5.5 Control Parameters

The population size, M, was 500,000. A maximum size of 150 points (functions and terminals) was established for each result-producing branch and a maximum size of 100 points was established for each automatically defined function. The other parameters for controlling the runs are the default values that we apply to a broad range of problems (Koza, Bennett, Andre, and Keane 1999).

5.6 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.

5.7 Parallel Implementation

This problem was run on a home-built Beowulf-style (Sterling, Salmon, Becker, and Savarese 1999; Bennett, Koza, Shipman, and Stiffelman 1999) parallel cluster computer system consisting of 1,000 350 MHz Pentium II processors (each accompanied by 64 megabytes of RAM). The system has a 350 MHz Pentium II computer as host. The processing nodes are connected with a 100 megabit-per-second Ethernet. The processing nodes and the host use the Linux operating system. The distributed genetic algorithm with unsynchronized generations and semi-isolated subpopulations was used with a subpopulation size of Q = 500 at each of D = 1,000 demes. Two processors are housed in each of the 500 physical boxes of the system. As each processor (asynchronously) completes a generation, four boatloads of emigrants from each subpopulation (selected probabilistically based on fitness) are dispatched to each of the four toroidally adjacent processors. The migration rate is 2% (but 10% if the toroidally adjacent node is in the same physical box).

6 Results

The best individual in generation 0 has a fitness of 1926.498.

The best-of-run controller emerged in generation 129 (figure 1). This best-of-run controller has a fitness of 522.605. The result-producing branch of this best-of-run individual has 119 points (functions and terminals) and 95, 93, and 70 points, respectively, in its three automatically defined functions. Note that genetic programming employed a 4.8 second delay (comparable to the five-second plant delay) in the transfer function of the evolved pre-filter. This best-of-run controller from generation 129 has a better value of fitness for a step size of 1 volt, an internal gain, K, of 1.0, and a time-constant, τ,of 1.0 (the specific case considered by Astrom and Hagglund 1995).

Figure 2 compares the time-domain response to step input of the best-of-run controller from generation 129 (triangles) with the controller in Astrom and

Hagglund 1995 (squares) for a step size of 1 volt, an internal gain, K, of 1.0, and a time-constant, τ,of 1.0.

Figure 3 compares the disturbance rejection of the best-of-run controller from generation 129 (triangles) with the controller in Astrom and Hagglund 1995 (squares) for a step size of 1 volt, an internal gain, K, of 1.0, and a time-constant, τ,of 1.0.

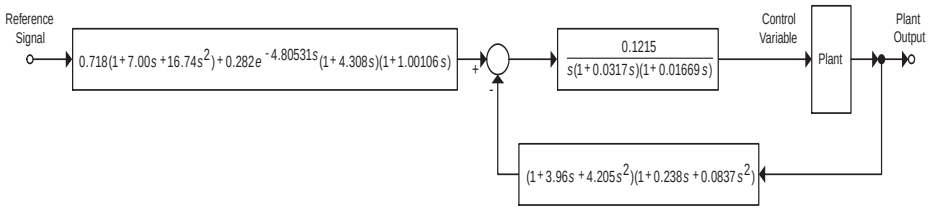

Figure 1 Best-of-run controller from generation 129 for three-lag plant with five-second delay.

Table 1 compares the fitness of the best-of-run controller from generation 129 and the Astrom and Hagglund 1995. Two of the entries are divided by the special weight C = 5.0. All 13 entries are better for the genetically evolved controller than for the Astrom and Hagglund 1995 controller.

Table 1 Fitness of two controllers for three-lag plant with five-second delay.

Element	Step size (volts)	Plant internal Gain, K	Time constant, τ	Best-of-run generation 129	Astrom and Hagglund controller
0	1	0.9	1.0	13.7	27.4
1	1	0.9	0.5	25.6	38.2
2	1	1.0	1.0	34.0 / 5 = 6.8	22.9
3	1	1.0	0.5	18.6	29.3
4	1	1.1	1.0	4.4	25.4
5	1	1.1	0.5	16.3	22.7
6	10^{-6}	0.9	1.0	13.2	27.4
7	10^{-6}	0.9	0.5	25.5	38.2
8	10^{-6}	1.0	1.0	30.7 / 5 = 6.1	22.9
9	10^{-6}	1.0	0.5	18.5	29.3
10	10^{-6}	1.1	1.0	4.3	25.4
11	10^{-6}	1.1	0.5	16.2	22.7
Disturbance	1	1	1	302	373

References

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

Astrom, Karl J. and Hagglund, Tore. 1995. *PID Controllers: Theory, Design, and Tuning*. 2nd Edition. Research Triangle Park, NC: Instrument Society of America.

Banzhaf, Wolfgang, Daida, Jason, Eiben, A. E., Garzon, Max H., Honavar, Vasant, Jakiela, Mark, and Smith, Robert E. (editors). 1999. *GECCO-99: Proceedings of*

the Genetic and Evolutionary Computation Conference, July 13-17, 1999, Orlando, Florida USA. San Francisco, CA: Morgan Kaufmann.

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

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

Bennett, Forrest H III, Koza, John R., Shipman, James, and Stiffelman, Oscar. 1999. Building a parallel computer system for $18,000 that performs a half peta-flop per day. In Banzhaf, Wolfgang, Daida, Jason, Eiben, A. E., Garzon, Max H., Honavar, Vasant, Jakiela, Mark, and Smith, Robert E. (editors). 1999. *GECCO-99: Proceedings of the Genetic and Evolutionary Computation Conference, July 13-17, 1999, Orlando, Florida USA*. San Francisco, CA: Morgan Kaufmann. 1484 - 1490.

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

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

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

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

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

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

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

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

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

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

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

Koza, John R., Keane, Martin A., Yu, Jessen, Bennett, Forrest H III, and Mydlowec, William. 2000. Automatic creation of human-competitive programs and controllers by means of genetic programming. *Genetic Programming and Evolvable Machines*. (1) 121 - 164.

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

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

Poli, Riccardo, Nordin, Peter, Langdon, William B., and Fogarty, Terence C. 1999. *Genetic Programming: Second European Workshop. EuroGP'99. Proceedings*. Lecture Notes in Computer Science. Volume 1598. Berlin: Springer-Verlag.

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

Ryan, Conor. 1999. *Automatic Re-engineering of Software Using Genetic Programming*. Amsterdam: Kluwer Academic Publishers.

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

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

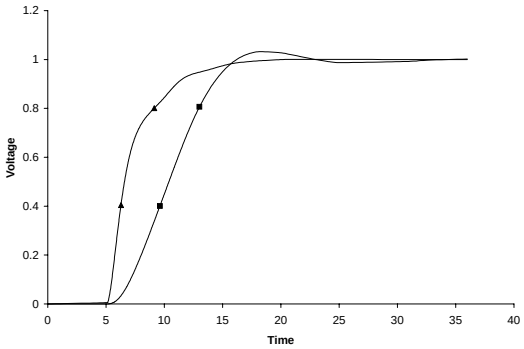

Figure 2 Comparison for step input.

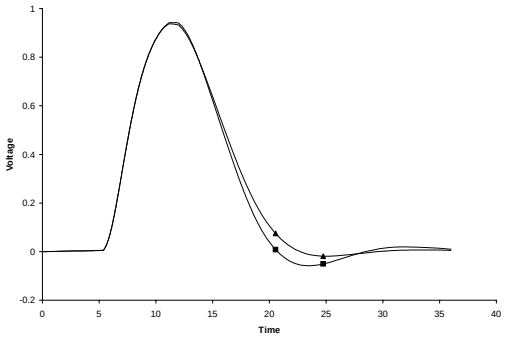

Figure 3 Comparison for disturbance rejection.