A Self-Learning Rule Base for Command Following in Dynamical Systems[1]

Wei K. Tsai, Hon-Mun Lee
Department of Electrical and Computer Engineering
University of California at Irvine
Irvine, CA 92717
and
Alexander Parlos
Department of Nuclear Engineering
Texas A&M University
College Station, TX 77843

Abstract

In this paper, a self-learning Rule Base for command following in dynamical systems is presented. The learning is accomplished though reinforcement learning using an associative memory called SAM. The main advantage of SAM is that it is a function approximator with explicit storage of training samples. A learning algorithm patterned after the dynamic programming is proposed. Two unstable dynamical systems artificially created are used for testing and the Rule Base was used to generate a feedback control to improve command following ability of the otherwise uncontrolled systems. The numerical results are very encouraging. The controlled systems exhibit a more stable behavior and a better capability to follow reference commands. The rules resulting from the reinforcement learning are explicitly stored and they can be modified or augmented by human experts. Due to the overlapping storage scheme of SAM, the stored rules are similar to fuzzy rules.

[1]This research research is funded by the U.S. Department of Energy Idaho Operations Office under the Grant DE-FG07-89ER12893.

I. Introduction

Expert systems (or knowledge-based systems) technology has many uses, see for example [1]. In this paper we will focus on automatic generation of the rule-base for controlling nonlinear dynamical systems often encountered in engineering endeavors. For nonlinear dynamical systems, there are two basic problems: estimation and control. Estimation refers to the the problem of reconstructing dynamic (as well as static) behaviors of partially unknown systems from input-output sample pairs. Control refers to the problem of generating desired system behaviors by exerting some control efforts to the system. In this paper, we will focus on one sub-problem of the control problem: command following. The problem is to generate feedback control rules to force the response of the controlled system to follow a reference command input. Our approach is to generate needed control by reinforcement learning [2] using an associative memory.

In this paper, we will focus only on the heart of an expert system, the Rule Base. A rule is of the form: "if $x_1 = x_1^j$, $x_2 = x_2^j$, ..., $x_d = x_d^j$, then the control is $u = u^j$." We propose a self-learning Rule Base in which learning is accomplished though reinforcement learning using an associative memory called SAM (Self-organizing Associative Memory). Self-learning is one of the main feature of the proposed Rule Base. Self-learning is especially useful for situations in which the dynamical system has a highly complex behavior such that even experienced engineers have difficulties in designing control laws. For such systems, additional rules are needed to supplement the experts' knowledge. An obvious way to gain additional knowledge is to experiment with an approximately realistic model of the system by feeding the model with various reference command inputs and feedback control efforts and observing the output behaviors. Such experimenting is known as *reinforcement learning* in the artificial neural network (ANN) literature.

The main advantage of using SAM is that it is an associative memory with explicit storage of training samples. Each training sample can be interpreted as a rule. The rules obtained through reinforcement learning are explicitly stored and they can be modified or augmented by human experts. Such modification or augmentation is useful when the model of the dynamical system is not entirely accurate and a human expert can modify a rule learned from the approximate model to incorporate dynamics not described by the model. Due to the fact that different rules can fire over overlapping regions, the rules base resembles a fuzzy rule base.

A learning algorithm patterned after the dynamic programming is also proposed. Two unstable dynamical systems artificially created are used for testing and the Rule Base was used to generate a feedback control to improve command following ability of the otherwise uncontrolled systems. The numerical results are very encouraging. The controlled systems exhibit a more stable behavior and a better capability to follow reference commands.

Since SAM is the essential part of the Rule Base, we now briefly describe the distinctive features of SAM and basic motivation behind the creation of SAM. SAM can be considered as a nonlinear function approximator. To obtain the best approximation,

techniques of the classical approximation theory, regression theory, and system identification theory, which include curve-fitting, Volterra and other basis function approximation methods, spline methods and others could be applied. In designing SAM, we use *local linear approximation* or *piecewise linear approximation* to represent the identification model. Similar to the classical spline technique, we employ linear interpolation to generate a recalled output for an input which is never seen before.

This paper is organized as follows. Section II motivates the selection of a model class for the dynamical systems under consideration. Section III provides a description of SAM. Section IV describes the reinforcement learning and section V presents simulation results. The paper concludes in section VI.

II. The Nonlinear Input-Output Model

In this paper, we assume the nonlinear dynamical system is described in the following Nonlinear MIMO (Multi-Input and Multi-Output) Input-Output form:

$$y(k) = f(y(k-1), y(k-2), \ldots, y(k-n), u(k), u(k-1), \ldots, u(k-q)), \qquad (1)$$

where $y(k) \in \Re^p$, $u(k) \in \Re^m$, k is a discrete time index, and $f(\cdot)$ is a general vector-valued nonlinear function of multiple variables. The above system could represent either a genuine discrete-time system or a sampled continuous-time system.

The above input-output model is also known as the Nonlinear Auto-Regression with eXogenous inputs (NARX)[4]. The above model also includes dynamical systems with noise and disturbance, either at the input or at the output, or at both places. The overall input vector $u(k)$ could be decomposed into three parts: the control input components, the disturbance input components (i.e., the un-intended inputs either due to noise or exogenous disturbances), and the measurement noise components.

III. A brief Description of SAM

A. The Overlapping Local Linear Approximation

The approximation method adopted for SAM is an *overlapping local linear approximation* (OLLA). Consider the generic scalar function approximation problem:

$$y = f(x) \qquad , y \in \Re^1,\ x \in \Re^d, \qquad (2)$$

For each x of interest, we assume that there exist a neighborhood of x, $N(x)$, such that, for all $\bar{x} \in N(x)$, $f(\bar{x})$ is well approximated by a linear functional:

$$f(\bar{x}) = a^T \bar{x} + b, \qquad (3)$$

where a is a d-dimensional weight vector and b is a scalar. The function can be viewed in the $\Re^{d+1}$ space as a linear hyperplane by defining the augmented state vector $z = [1, x^T]^T$,

and the augmented weight vector $w = [b, a^T]^T$. The hyperplane is then described by the equation $f(x) = w^T z$.

To determine the local hyperplane, only $d+1$ linearly independent prototypes - a prototype is defined to be a vector of the form $[\bar{x}^T, f(\bar{x})]^T$ - from the neighborhood $N(x)$ will be needed. If there are exactly $d+1$ linearly independent prototypes available, one can solve the following linear equation to obtain the local parameters w:

$$\begin{bmatrix} 1 & \bar{x}_1^1 & \dots & \bar{x}_d^1 \\ 1 & \bar{x}_1^2 & \dots & \bar{x}_d^2 \\ & & \vdots & \\ 1 & \bar{x}_1^2 & \dots & \bar{x}_d^2 \end{bmatrix} \begin{bmatrix} b \\ a_1 \\ \vdots \\ a_d \end{bmatrix} = \begin{bmatrix} f(\bar{x}^1) \\ f(\bar{x}^2) \\ \vdots \\ f(\bar{x}^{d+1}) \end{bmatrix} \tag{4}$$

Once the local $w(x)$ is determined, the recalled value $f(x)$ can be computed simply via the formula $f(x) = a^T x + b$.

Now suppose there are less than $d+1$ linearly independent prototypes available, i.e., there are less equations to determine uniquely the local weight $w(x)$. There are many options here, and we decided to use the minimum norm solution to (4). The minimum norm solution is equivalent to a least square minimization problem:

$$\begin{aligned} \min \quad & \|w\|^2 \\ s.t. \quad & Aw = \bar{f} \end{aligned} \tag{5}$$

where A is matrix in the left hand side of equation (4), $\bar{f}$ is the vector in the right hand side of (4), and the $\|\cdot\|$ is the usual Euclidean norm (ℓ^2 norm). The solution to (5) is well-known: a pseudo-inverse solution described by the following equation:

$$w = A^T(AA^T)^{-1}\bar{f}. \tag{6}$$

We now briefly describe the *storing* and *retrieval* mechanism of the OLLA method. In storing, a new sample $[x^T, f(x)]^T$ will be stored in its entirety, if $f(x)$ cannot be adequately linearly approximated by the already stored prototypes in $N(x)$. Let $\tilde{f}(x)$ denote the value recalled from the present memory, i.e., with no more than $d+1$ prototypes stored in the memory in the neighborhood $N(x)$, $\tilde{f}(x)$ is computed based on (3) with the weights computed using either (4) or (6). The value $\tilde{f}(x)$ is said to be *recalled* from the memory. The user of the linear SAM then chooses a tolerance ε_2 such that if

$$|f(x) - \tilde{f}(x)| > \varepsilon_2, \tag{7}$$

then the sample $[x^T, f(x)]^T$ is stored into the memory.

The reason that the approximation method described above is called an *overlapping* method is that, in a small neighborhood, the function could be approximated by several linear hyperplanes computed based on several overlapping (intersecting) sets of prototypes. This overlapping property is the main difference between the linear SAM approximation approach and the classical local linear parametric regression method [5].

B. *The Architecture of SAM*

C. *The Storage and Retrieval Scheme of SAM*

We now describe the detailed computation scheme to implement the storing and retrieval schemes aiming to minimize searching time for both storing (learning) and recall. There are many ways to implement these computation structures. The description here is most conveniently interpreted as a sequential algorithm. However, the algorithm can be easily parallelized given a proper hardware architecture.

We have developed three storing schemes: *tree scheme*, *mesh scheme*, and the *hybrid scheme*. In this paper, we will only described the mesh scheme. A simple mesh storing scheme is described as follows. In the following description, let the current training sample be x. Let $\varepsilon_1 > 0$ be a user specified scalar such that a linear interpolation of x by a set of $d+1$ closest vectors to x $\{\bar{x}^i : i = 1, \ldots, d+1\}$ will be allowed only if

$$\|x - \bar{x}^i\| \leq \varepsilon_1. \tag{8}$$

The condition(8) will be referred to as the *ε-neighborhood condition*. Define the interpolation index:

$$I(x) = |f(x) - \bar{f}(x)|, \tag{9}$$

where $\bar{f}(x)$ is the recalled value generated by SAM for x. ε_4 is another user-specified parameter which is used by the algorithm to define a hypercube neighborhood. The only requirement is that the hypercube region defined by

$$\{\bar{x} : \bar{x}_i - \varepsilon_4 < x_i < \bar{x}_i + \varepsilon_4, \forall i = 1, \ldots, d, \}, \tag{10}$$

contains the ε-neighborhood defined by (8). The mesh will be called the *SAM mesh*.

1. Initialization: Let the first training sample be x. Then let the entry node to the mesh represent the vector x and each node that will be added to the mesh represent a particular prototype. The node storing x will have $2d$ pointers pointing to the set of mesh neighbors:

$$\begin{aligned}
x^1 &= [x_1^H, x_2, \ldots, x_d]^T, \\
x^2 &= [x_1^L, x_2, \ldots, x_d]^T, \\
x^3 &= [x_1, x_2^H, x_3, \ldots, x_d]^T, \\
x^4 &= [x_1, x_2^L, x_3, \ldots, x_d]^T, \\
&\cdots \qquad\qquad \cdots, \\
x^{2d-1} &= [x_1, x_2, \ldots, x_{d-1}, x_d^H]^T \\
x^{2d} &= [x_1, x_2, \ldots, x_{d-1}, x_d^L]^T,
\end{aligned} \tag{11}$$

where

$$x_i^L < x_i < x_i^H, \quad x_i^H - x_i^L \leq 2\varepsilon_4, \ \forall i = 1, \ldots, d, \tag{12}$$

are components of either *genuine* or *pseudo* prototypes – if no genuine prototype vectors satisfying (12) are found, then create artificial (pseudo) prototypes to make up the mesh and to mark boundaries of the mesh. A node storing a pseudo prototype x does not carry actual value of $f(x)$.

2. For the current training prototype x, compute the interpolation set and the interpolation index as follows: search in an ε-neighborhood of x to find a set of $d+1$ closest vectors to x, denoted by $\tilde{N}(x)$. Compute the interpolation index of x as in (9).

3. For the current training prototype x, check if $f(x)$ can be well interpolated from previously stored prototypes according to (7). If this is so, the current training sample is discarded.

4. Else, extend the SAM mesh by adding x and $f(x)$ to the SAM mesh.

The retrieval scheme for the mesh scheme is trivial: Suppose the cue vector is x and SAM is asked to supply an approximate $f(x)$.

1. Retrieve the ε-neighborhood set $\tilde{N}(x)$ of x.

2. If there is a almost matching prototype, say $\bar{x}$, then return the value $f(\bar{x})$. Otherwise compute the recalled values based on (10).

IV. The Reinforcement Learning Algorithm

The reinforcement learning algorithm proposed is described below. First we describe the feedback structure. Let k be the discrete time index. Let $u_f(k)$ be the reference command input at time k. For each k the algorithm iterates through the index i to generate a desirable incremental feedback control $u_e^i(k)$. The overall input $u(k)$ is the difference between reference input $u_f(k)$ and feedback control $u_c(k)$: $u(k) = u_f(k) - u_c(k)$. For training, the overall feedback control $u_c(k)$ is decomposed into two parts: the current control $u_c(k)$ recalled from SAM, and the i-th trial incremental control $u_e^i(k)$: $u_c^i(k) = u_e^i(k) + u_c(k)$. Let $J^i(k) = (y^i(k) - u_f(k))^2$ be the error at time k using the i-th trial incremental control. At time index k, the following is done:

1. Set $u_c(k) = SAM(y(k))$.

2. Set $i \leftarrow 1$.

3. Generate a trial incremental control $u_e^i(k)$;

4. Set $u(k) = u_f(k) - (u_e^i(k) + u_c(k))$ and generate the output $y^i(k+1)$ with $u(k)$;

5. If $J^i(k) < J^{i-1}(k)$, store the relation $y^i(k) \rightarrow u_e^i(k) + u_c(k)$ into SAM; else set $i \leftarrow i+1$ and go to step 3.

6. Set $k \leftarrow k+1$

V. Numerical Simulation Results

We tested two artificial SISO (Single-Input-Single-Output) systems. The error measure we use to gauge the overall performance of the predictor is a normalized:

$$E = \frac{\frac{1}{N}\sum_{i=1}^{N} |y_d(k) - y(k)|}{\sqrt{\frac{1}{N}\sum_{i=1}^{N}(y_d(k))^2}} \tag{13}$$

where $y_d(k)$ is the actual output at time k, $y(k)$ is predicted output at time k, N is the total number of samples taken in time.

A. Example 1:

The SISO nonlinear system is described by the following equation:

$$y(k) = u(k) + 0.2y(k-1) - 0.3(y(k-1)y(k-3))^{\frac{2}{3}} - 0.5(y(k-2)y(k-4))^2,$$

where $y(k)$ is the output, and $u(k)$ is the input. We made seven different tests. In the first test, we trained the system with two separate ramp inputs with slopes 0.01 and 0.009 and tested the system with a ramp input of slope 0.0095. The second test is similar to the first test except that there is a output white noise of magnitude .01. In the third test, we trained the system with two separate step inputs with magnitudes 1.0 and 0.9 and tested the system with a step input of magnitude .095. In the fourth test, we trained the system with two separate step inputs with magnitudes 1.2 and 1.3 and tested the system with a step input of magnitude 1.25. In the fifth test, we trained the system with two sinusoids inputs and tested the system with an input which is added to a sinusoid. The sixth test is similar to the first test except that there is a output white noise of magnitude .01. In the seventh test we trained the system with two ramp-with-step inputs with step magnitudes at 1.1 and 1.0, and we tested with a ramp-with-step input at 1.05.

From the Figures attached, it is clear that the controlled system has a better commdan-following capability than the uncontrolled. The only drawback is that the learning algorithm does not seems to perform well when there is an output noise.

B. Example 2:

The nonlinear system is described by the following equations:

$$y(k) = cos[\frac{1}{2}u(k)y(k-1)] + (y(k-1)y(k-3))^2.$$

This system is highly unstable, and in this example we demonstrate the stabilizing ability of the Rule Base feedback control. We trained the systems with two step inputs with magnitudes 0.55 and 0.45 and tested the system with a step input of magnitude of 0.5. The uncontrolled system exploded at around $k = 20$ while the controlled system is marginally stable; it did not explode.

VI. Conclusions

In this paper, a self-learning Rule Base for command following in dynamical systems is presented. The learning is accomplished though reinforcement learning using an associative memory. A learning algorithm patterned after the dynamic programming is also proposed. Two unstable dynamical systems artificially created are used for testing and the Rule Base was used to generate a feedback control to improve command following ability of the otherwise uncontrolled systems. The numerical results are very encouraging. The controlled systems exhibit a more stable behavior and a better capability to follow reference commands.

There are several directions of further research following this preliminary work. One is to improve the reinforcement learning algorithm so that the feedback controlled system responses will more closely follow the reference inputs. We intend to borrow insights from dynamic programming as the reinforcement learning algorithm proposed is very similar to the standard dynamic programming algorithm. Another is to test the self-learning Rule Base with realistic dynamical systems, especially systems with model uncertainty and output noise. For realistic systems, it would be interesting to investigate how a human expert can modify and add to the learned Rule Base so as to incorporate his own knowledge into the final Rule Base. A third direction is to apply the self-learning Rule Base to other control problems.

References

[1] James P. Ignizio,*Introduction to Expert Systems: The development and Implementation of Rule-Based Expert Systems*, 1991, New York, N.Y.: McGraw-Hill, Inc.

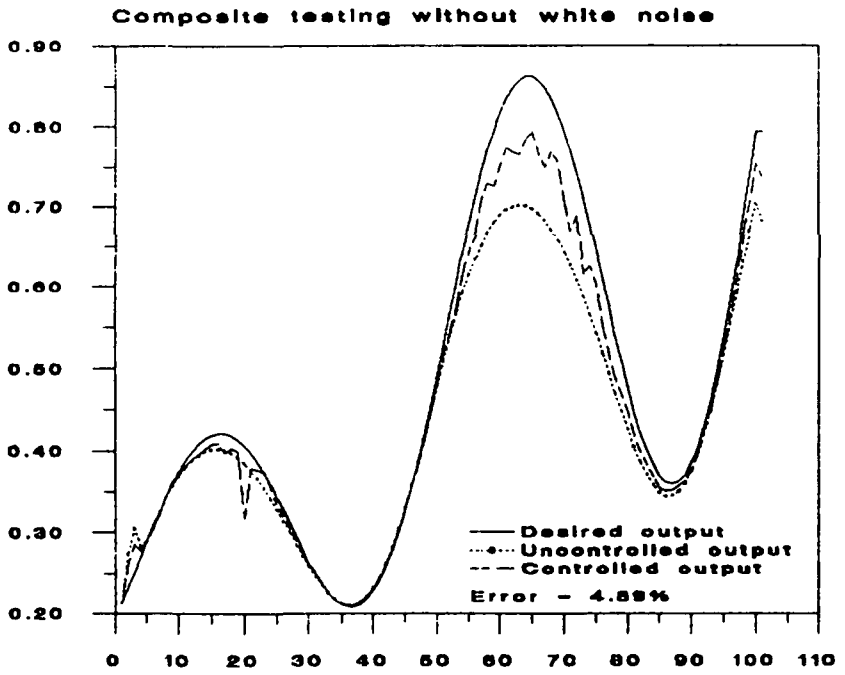
[2] Robert Hecht-Nielsen, *Neurocomputing*, 1991, Reading, MA: Addison-Wesley.

[3] L. A. Zadeh,"A Theory of Approximate Reasoning," *Machine Intelligence*, Vol. 9, J. Hayes, D. Michie, and L. Mikulich (eds.), New York:Elsevier, 1979.

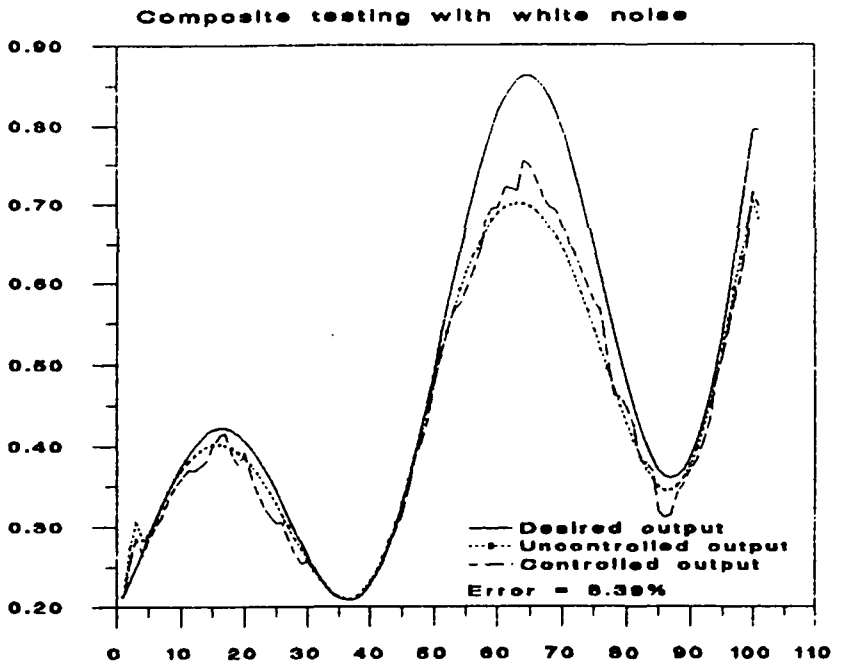
[4] Chen, S., and S. A. Billings, "Representations of non-linear systems: the NARMAX model," *International Journal of Control,* 49, 3, 1013-1032, 1989.

[5] Cleveland, W.S., "Robust Locally Weighted Regression and Smoothing Scatter Plots," *J. Amer. Statist. Assoc.*, Vol.-74, pp. 828-836, 1979.

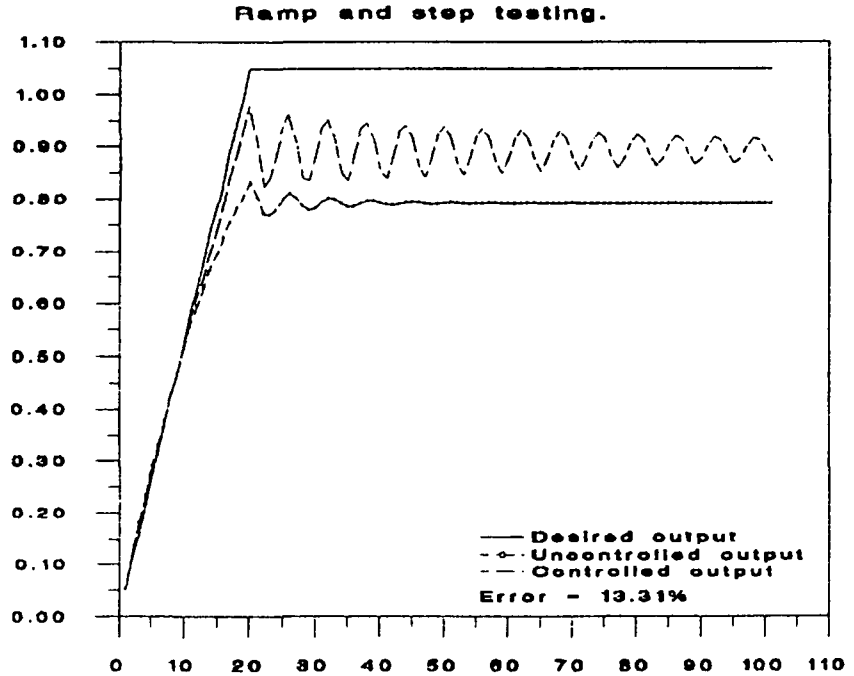
[6] Friedman, J.H., "Multivariate Adaptive Regression Splines," *The Annals of Statistics*, Vol. 19, No. 1, pp. 1-141, March, 1991.



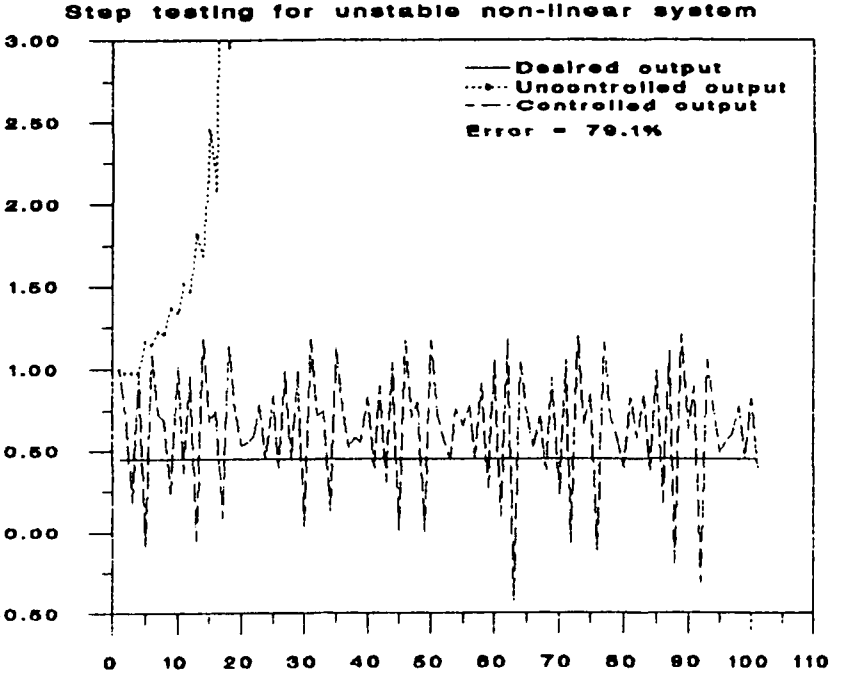
Example 1 : test 5



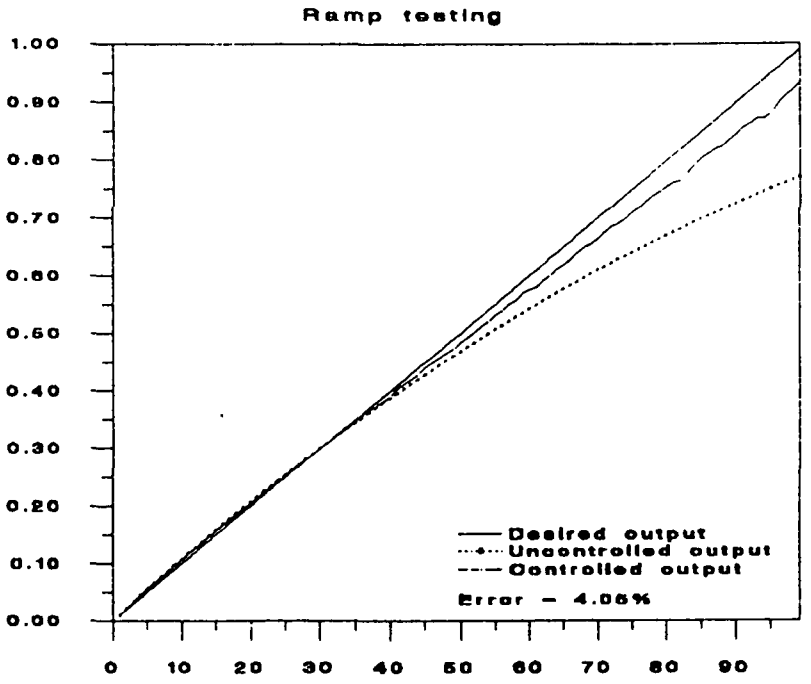
Example 1 : test 6



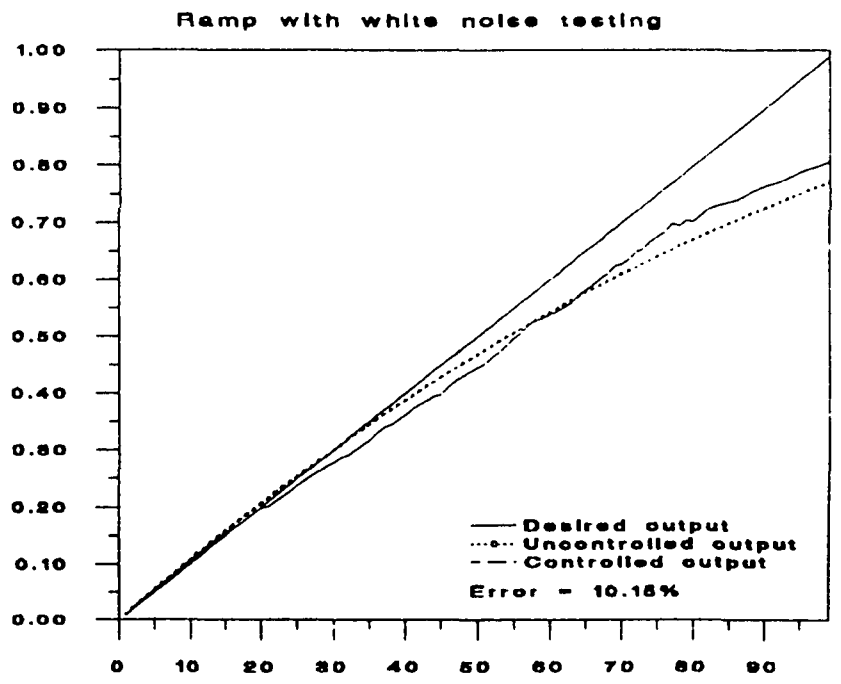
Example 1 : test 7



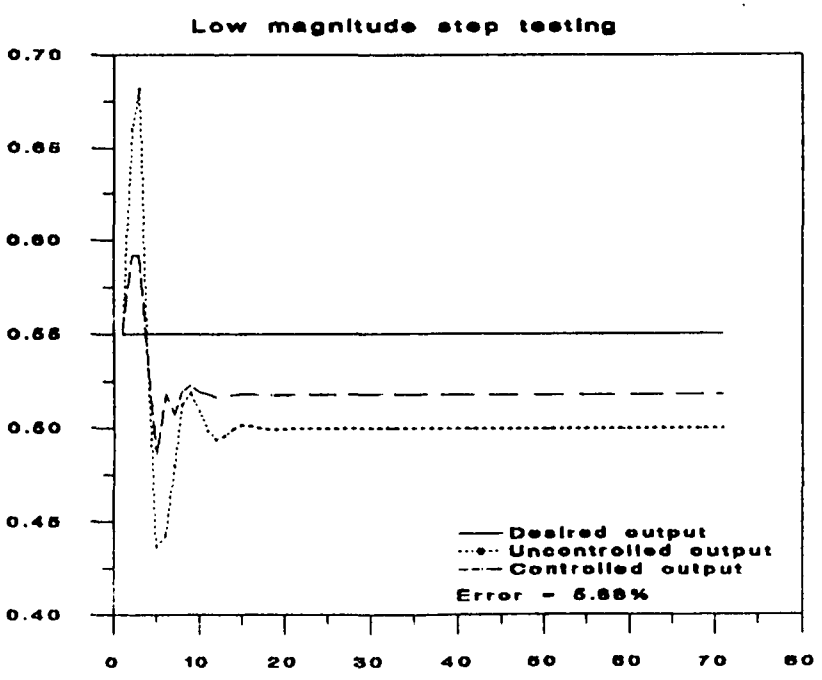
Example 2



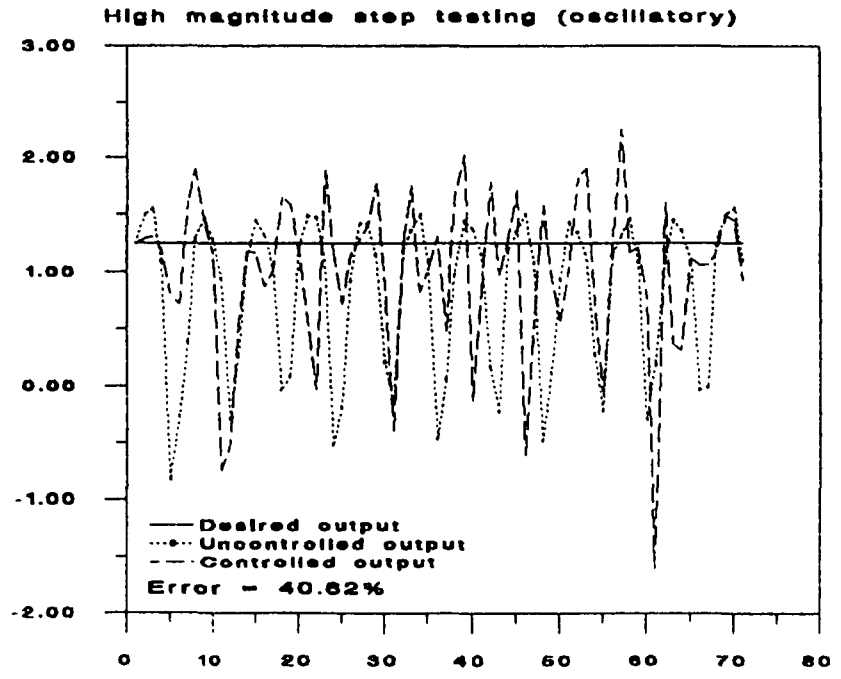
Example 1 : test 1



Example 1: test 2



Example 1 : test 3



Example 1 : test 4