DIGITAL CONVERSION OF NONLINEAR COMPENSATORS WITH ANTIRESET-WINDUP COMPENSATION: STUDIES, ANALYSIS AND DESIGN

A Thesis Presented

by

RICHARD J. SPANGENBERGER

Submitted to the Graduate School of the
University of Massachusetts in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL AND COMPUTER ENGINEERING

February 1992

Department of Electrical and Computer Engineering

DIGITAL CONVERSION OF NONLINEAR COMPENSATORS WITH ANTIRESET-WINDUP COMPENSATION: STUDIES, ANALYSIS AND DESIGN

A Thesis Presented

by

RICHARD J. SPANGENBERGER

Approved as to style and content by:

[illegible]

D. Looze, Chair

[illegible]

W. Gong, Member

[illegible]

T. Djaferis, Member

[illegible]

Lewis E. Franks, Department Head
Department of Electrical and Computer Engineering

ACKNOWLEDGEMENTS

The author wishes to thank Dave Markham, Tom Arnold, Virginia Crocker-Timmons and Dan Hogan for providing the opportunity to accomplish this thesis, and for their support and guidance.

The author also wishes to express his deep thanks to Doug Looze for his patience and guidance in helping me to focus and complete the thesis.

Deepest love and thanks to Anne Magnifico for her help, love, and especially her perseverance throughout the last three years. Thanks to my good friend Ed Taylor, and very special thanks to Dave Burkhardt.

ABSTRACT

DIGITAL CONVERSION OF
NONLINEAR COMPENSATORS WITH
ANTIRESET-WINDUP COMPENSATION:
STUDIES, ANALYSIS AND DESIGN

FEBRUARY 1992

RICHARD J. SPANGENBERGER, B.E.T., UNIVERSITY OF DAYTON
M.S.E.C.E., UNIVERSITY OF MASSACHUSETTS
Directed by: Professor Douglas P. Looze

All physically realizable systems are subject to saturations of one form or another. Control systems having saturations are susceptible to the nonlinear problem of reset-windup if the controllers within those systems contain one or more integrators. Reset-windup is a condition whereby the integrator continues to integrate the feedback error and add to the control signal, even in the presence of a decreasing error signal. This phenomenon can lead to excessive overshoot in the system in response to large setpoint changes.

This paper discusses the problem of reset-windup in detail and presents several methods for correcting this problem in continuous-time systems as discussed in existing controls literature. Two approaches to the elimination of reset-windup are discussed in detail: the conventional antireset-windup (CAW) scheme and the override signal (OS) scheme. The application of these methods to continuous-time systems is reviewed for simple example systems. The paper then proposes implementations of these methods for discrete-time systems, discusses problems associated with these implementations, including the phenomenon of "chatter", and presents design criteria to make these implementations useful. Finally, the practical application of antireset-windup compensation is discussed through the design of a digital controller for an existing system.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

ARW	Antireset-Windup
CAW	Conventional Antireset-Windup
PI	Proportional-Integral
PD	Proportional-Derivative
PID	Proportional-Integral-Derivative
OS	Override Signal

LIST OF SYMBOLS

Symbol	Nomenclature	Units
k	Discrete Sample Counter Variable	
s	Complex Laplace Transform Variable	
z	Complex z-transform Variable	
C	Controller/Compensator Output Signal	
E	Error Signal	
I	Plant Input Signal	
R	Command Signal, Reference, or Setpoint	
T,t	Time	Seconds
U	Control Output Signal	
Y	Plant Output Signal	
Z	z-Transform	
Z^{-1}	Inverse z-Transform	
α	Angular Acceleration	rad/sec^2
π	Pi	
θ	Angular Position	radians
ω	Angular Rate	rad/sec
$\Im$	Laplace Transform	
$\Im^{-1}$	Inverse Laplace Transform	

CHAPTER 1
INTRODUCTION AND HISTORICAL REVIEW

The control of physical systems by utilizing a computer is becoming more commonplace. The desirability to use digital electronics in control systems has been influenced by the continuing decrease in the cost of microprocessor, single-board computer, and other digital elements - coupled with the weight advantages and increased reliability that digital electronics offers. The advantages of digital logic for control are numerous: the flexibility of the control mechanisms is increased, and the control functions can be integrated with other digital or computing elements within the system. Additionally, the total system cost (including built-in-test capability, expandability, flexibility, etc.) of using a digital implementation is often cheaper, even though the analog control elements themselves may be less expensive than a microprocessor.

All physically realizable systems are subject to saturations of one form or another. Most controllers, whether they are digital or analog, contain one or more integrators, and control systems having saturations are susceptible to the nonlinear problem of reset-windup if the controllers within those systems contain integrators. Reset-windup is a condition where the integrator continues to integrate the feedback error and add to the control signal, even in the presence of a decreasing error signal. This phenomenon can lead to excessive overshoot in the system in response to large setpoint changes.

This paper discusses the problem of reset-windup in detail and presents several methods for correcting this problem in continuous-time systems as discussed in existing controls literature. Chapter 2 introduces the problem of reset-windup and explains the phenomenon in continuous-time systems along with the consequences of reset-windup including excessive overshoot and settling time. Chapter 3 presents two approaches to the elimination of reset-windup which are discussed in detail: the conventional antireset-windup (CAW) scheme and the override signal (OS) scheme. The application of these methods to continuous-time systems is reviewed for simple example systems. The paper then proposes implementations of these methods for discrete-time systems. The paper discusses problems associated with these implementations and introduces the phenomenon of "chatter". The paper then proposes design criteria for making these digital implementations useful. Finally, the paper shows how to apply the antireset-windup compensation to practical problems through the design of a digital controller for an existing system. Chapter 4 summarizes the conclusions of the paper.

CHAPTER 2

TECHNICAL BACKGROUND

2.1 Reset-Windup in Continuous Systems

All simple control systems, whether analog or digital, consist of a plant and a controller. Figure 1 depicts a simple analog control system. The plant is that which is to be controlled and may be a motor for positioning of a radar antenna, an aircraft control surface actuator, a chemical process flow control valve, or countless other mechanisms. The controller accepts the commands for controlling the plant and generates control signals that make the plant behave dynamically in some desired manner with a desired level of performance. In most analog control systems, this controller is a compensation network, or compensator, designed with op-amps and discrete components; in a digital control system, this compensator is a control algorithm which determines a command to be applied to the plant at discrete intervals (every sample time). This paper will use the term *compensator* to refer to this element in analog (continuous-time) systems, and the term *controller* in reference to digital (discrete-time) systems.

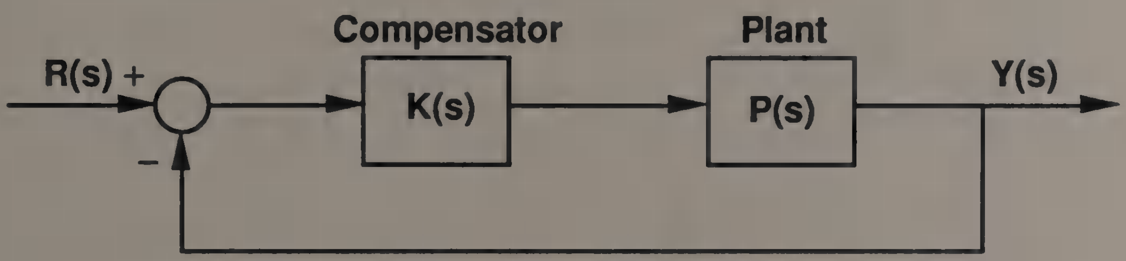


Figure 1. Simple Control System

Figure 1 is a block diagram model of the plant and compensator consisting of linear elements only. Most control system analysis uses linear models since there is a larger body of knowledge associated with linear analysis techniques whereas nonlinear analysis techniques are few and usually limited in their ability to predict overall system performance. However, no physically realizable plant or controller is purely linear. All physically realizable systems have limits to their performance: no physical systems can accelerate instantaneously; no systems have infinite linearity. Most mechanical systems have nonlinearities associated with them such as deadband, friction, and backlash.

System nonlinearities occur in two ways: some are inherent in the plant model, and some can be added by the designer [1]. Saturation is one of the most common nonlinearities present in almost all systems. Nearly all plants have some type of control input saturation and, often, it is the dominant nonlinearity [2]. In the case of a motor, the current that the motor can effectively handle is limited to some value and, as such, the maximum torque that the motor can deliver is limited. In the case of a position servo for controlling a track antenna, the angular excursion of the antenna is often limited to some predefined field of view. In the case of a hydraulic or pneumatic servo, there is always a limit to the pressure that can be applied within the system due to the design of the fluidic lines. All of these plants contain saturations. These saturations put limits on the ability of the controller to develop a certain performance from the system. Therefore, most systems are better modeled as shown in Figure 2.

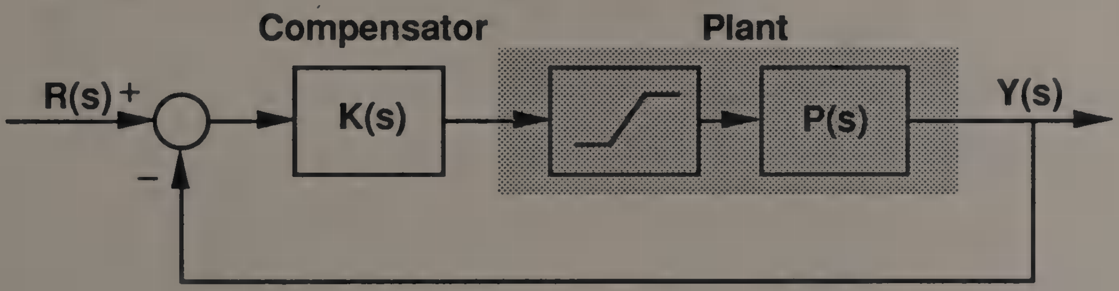


Figure 2. Simple Control System with Plant Saturation

In this model, an otherwise linear system contains an input magnitude saturation. The saturation block represents a saturation of the control signal that is applied to the the plant. The output of the saturation block is equal to the input (transfer function of 1) until the positive or negative saturation limit is reached, after which the output is held at the limit regardless of the input until the input returns to the linear region.

As previously stated, this ultimately limits the performance which can be derived from the system by restricting the response of the plant to a subset of controller commands; furthermore, the controller itself will contain saturations as well. In a continuous system, the output of the operational amplifiers will be limited by their rail voltage and current-limiting capability. In discrete systems, the controller will be limited by the computational limits of the computer (number of bits of computation), and A/D and D/A converter accuracies. For the sake of simplicity, all saturations affecting the control signal will be represented as the most restrictive saturation present to the plant input as shown in Figure 3.

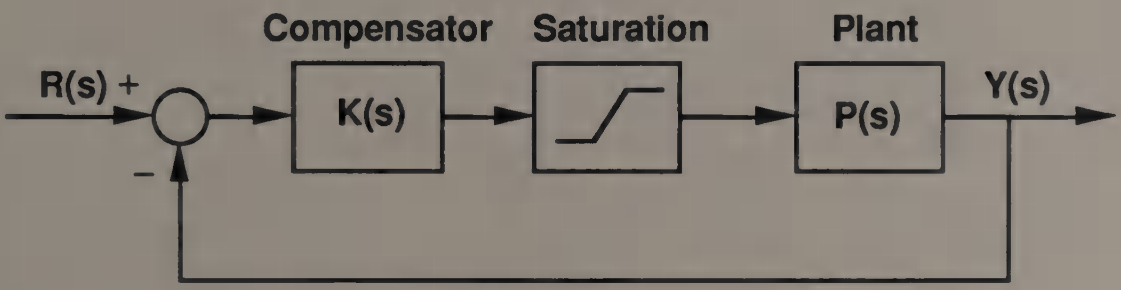


Figure 3. Control System with Saturation

Saturation within the system, in many cases, leads to the nonlinear phenomenon known as integrator reset-windup. Reset-windup occurs within compensators containing integrators. If a linear single-loop control system, such as Figure 3, is submitted to large deviations (e.g., during start up), the control variable may saturate. If there is still an error signal, it will be integrated and the integral term may become very large if the saturation lasts for an extended time period. This is called "reset-windup" because integral action is often called reset in instrumentation literature. Windup may lead to a large overshoot in the system response [3].

A more precise definition of reset-windup is given by Buckley [4]: *Reset-windup is the nonlinear behavior of a controller when saturated by a large error signal, such that the integrator within the controller continues to add to the saturated value even after the error is reduced and approaches zero. The integrator cannot begin to "discharge" until the sign of the error changes.* In other words, the control signal overshoots due to the continual charging of the integrator. This effect is highly nonlinear and does not appear in the usual linear equations for a controller or a plant.

In continuous systems, this windup is usually realized by the charging of a capacitor within the compensator. The effect is best illustrated by an example: assume a continuous system exists such as that in Figure 4.

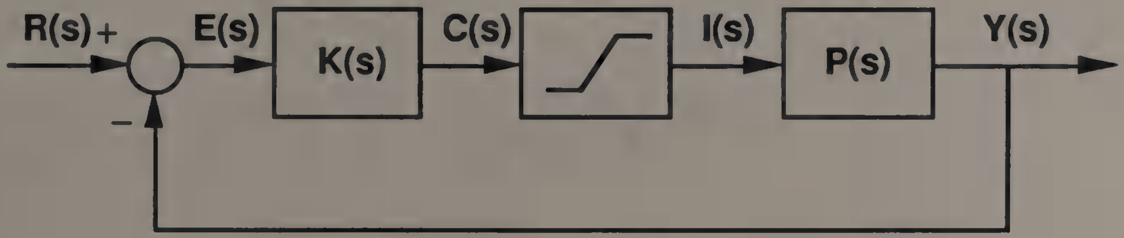


Figure 4. Example System for Windup

The system contains a plant, P(s), and a compensator, K(s). Unless otherwise stated, it will be assumed throughout this paper that the saturation block represents

$$\mathrm{Sat}(i) = \begin{cases} 1. & \text{if } c > 1 \\ c. & \text{if } -1 < c < 1 \\ -1. & \text{if } c < 1 \end{cases} \tag{1}$$

It is also assumed that the output of the saturation is available as a measurement and that there is no uncertainty in the saturation itself. This can be easily justified. Assuming that the saturation limits within the plant are known, the saturation can then be easily imposed on the compensator as part of the compensator design, thereby making such measurements available. For simplicity of example, assume that the plant dynamics for Figure 4 are described by

$$P(s) = \frac{1}{s} \tag{2}$$

and that the compensator is a simple proportional-integral (PI) controller described by

$$K(s) = K_1 + \frac{K_2}{s} \tag{3}$$

where $K_1 = 4$ and $K_2 = 16$. If the saturation is removed from the system temporarily, the linear open loop transfer function of the system is

$$L_{ol} = P{*}K = \frac{K_1 s + K_2}{s^2} . \tag{4}$$

The linear closed loop transfer function of this system, with no saturation, is

$$L_{cl} = \frac{P{*}K}{1 + P{*}K} = \frac{K_1 s + K_2}{s^2 + K_1 s + K_2} . \tag{5}$$

Assume that the system is in steady-state and that the output, y, of the plant is at a value of $y = 2$. Further assume that a step input to zero is applied at time $t = 0$. The response of the output y, the controller output signal c, and the plant input i is shown in Figure 5.

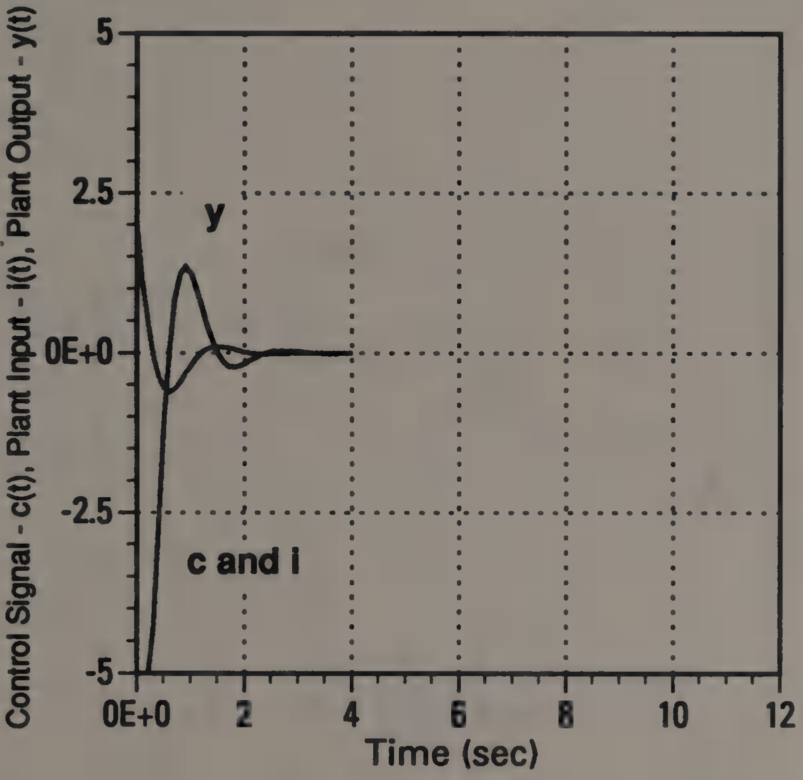


Figure 5. Linear Response of Simple System

Now observe the response of the system (Figure 6) under the same conditions but with the saturation included and described by equation (1). A fourth order Runge-Kutta simulation was developed to model the results discussed in Appendix B. Notice the very nonlinear effect on the output y due to this simple saturation. The output y exhibits classic windup. The large overshoot of the output is an example of the disadvantages of the windup phenomenon. Figure 7 shows the output along with the control signal c. Note that the large overshoot of the output is due to the large overshooting of the control signal.

Notice also that the control signal cannot begin to "discharge", as stated by the definition, until the error signal crosses zero and becomes negative. Finally, note that the windup phenomenon leads to a very long settling time due to the inability of the controller to "catch up" to the error signal.

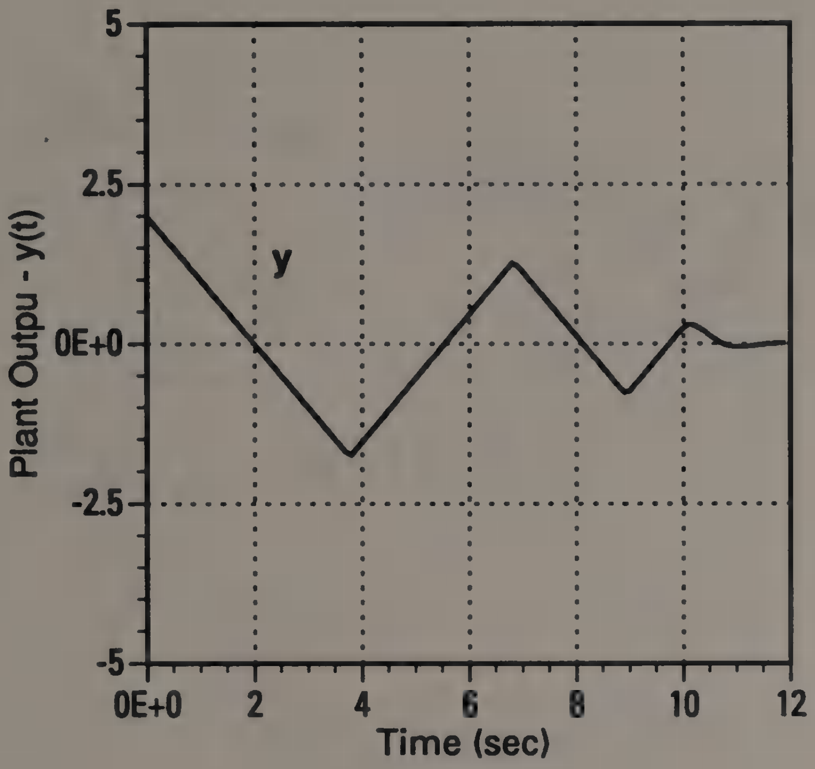


Figure 6. Response of Simple System with Saturation -y(t)

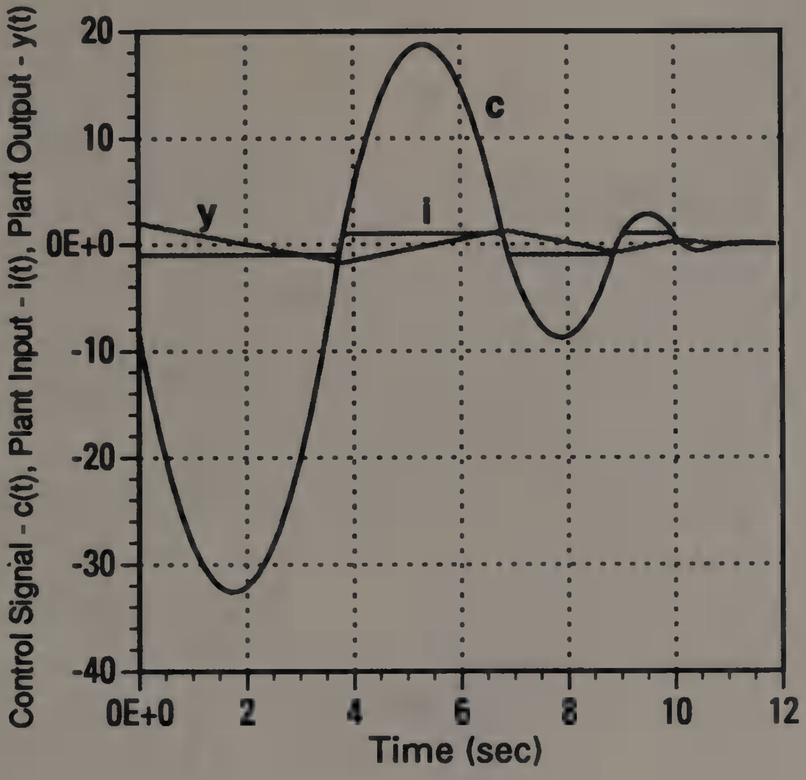


Figure 7. Response of Simple System with Saturation - c(t), i(t), y(t)

2.2 Antireset-Windup (ARW) Configurations in Continuous Systems

Several solutions to the problem of reset-windup have been proposed in the literature for continuous systems. All of these systems fall into the class of systems known as antireset-windup, or ARW, systems. There are two basic categories of ARW systems: conventional antireset-windup (CAW) configurations and override signal (OS) configurations. There are several ways to implement each configuration within each category. Some of the more common are discussed here.

2.2.1 Conventional Antireset-Windup Configurations

The first class of ARW configurations is the conventional antireset-windup or CAW configurations. Most are very similar and are designed around the premise of measuring the difference between the control signal and the output of the saturation, and modifying the input to the controller so as to keep the output of the controller at, or below, saturation. This has the effect of smoothing the response of the system near, or at, the saturation limits and keeping the system within the linear range of operation. This configuration is presented in Doyle and Smith [5], and Glattfelder and Schaufelberger [6]. The basic configuration of the CAW is shown below.

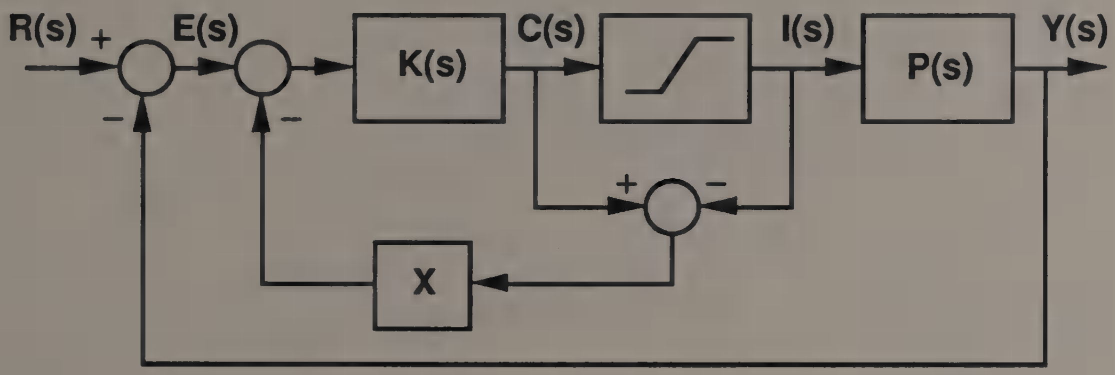


Figure 8. Conventional Antireset-Windup Configuration

In the CAW configuration, the input *c* to the saturation block and the output *i* of the saturation block are measured and the difference between the controller output and the plant input is fed back through a gain, X, to the controller. Windup will be prevented if the associated loop transfer function $L(s)_X = K(s)*X$ has a gain and a bandwidth much higher than that of L(s), which is equal to the forward path transfer function $L(s) = K(s)*P(s)$. It is suggested that the bandwidth of $L(s)_X$ be at least ten times that of L(s). As discussed in

Glattfelder [6], the actual implementation of the CAW can take many forms depending upon the design of the controller and the performance desired. For a generalized controller which has proportional, integral, and differential elements (a PID controller), the CAW can be implemented with the feedback around the entire controller, around the proportional and integral parts, or around just the integral portion of the controller as shown respectively in Figures 9a, 9b, and 9c.

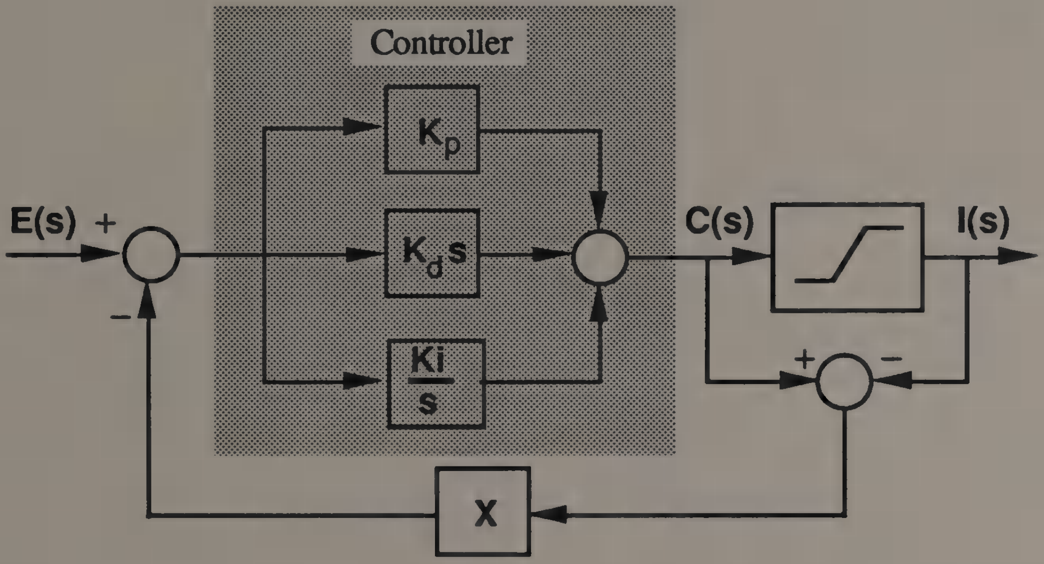


Figure 9a. CAW Configuration #1

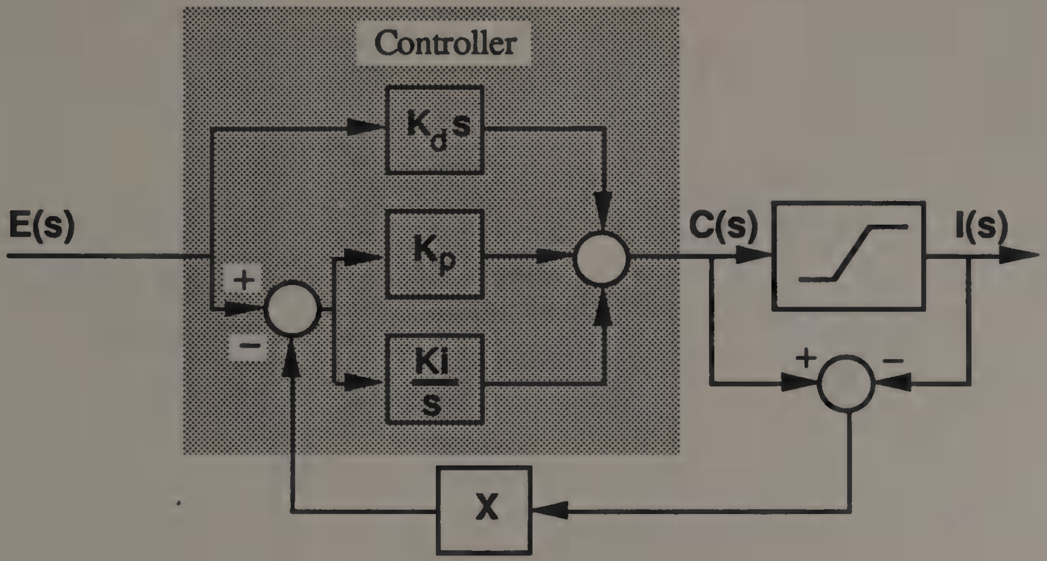


Figure 9b. CAW Configuration #2

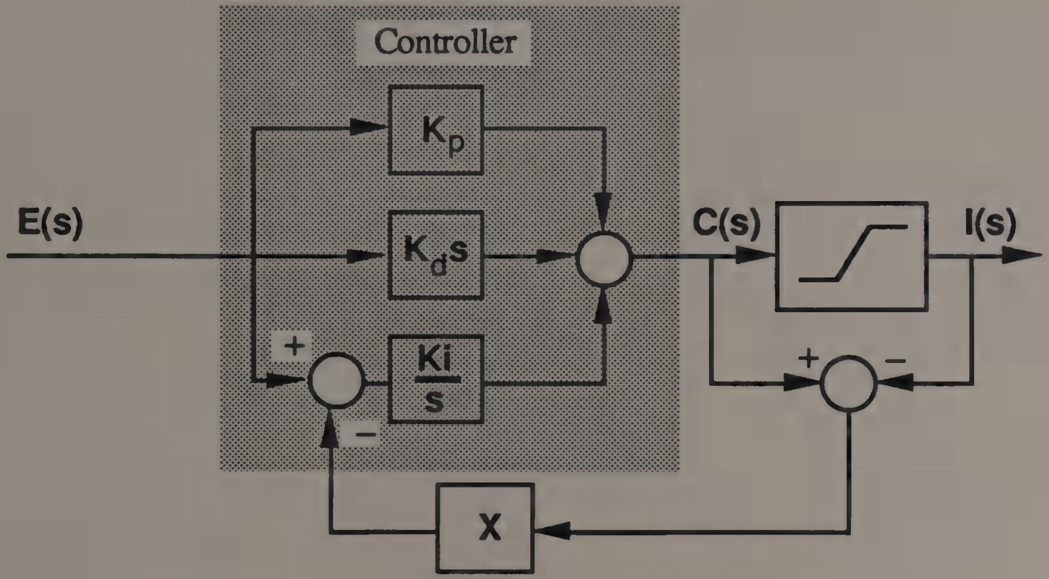


Figure 9c. CAW Configuration #3

2.2.2 Override Signal Configurations

The second class of ARW configurations is the override signal (OS) configuration. This type of configuration is presented by Glattfelder [7] and Buckley [8]. Rather than continuously adjusting the antireset compensation, as the CAW configurations do, the OS configurations switch-in the ARW compensation when needed and the normal compensation signal is switched-out. One such implementation given by Glattfelder [7] is shown in Figure 10. The controller output is compared to upper and lower limit setpoints, u_{hi} and u_{lo}, using a minimum and a maximum selector and high-gain amplifiers with gain K_2.

The upper and lower limit setpoints are set to the system saturation values. If c is driven towards either limit by the main error signal e, the corresponding high-gain feedback is then selected ("overriding" e) and adjusts c(t) continuously in such a way that c(t) never saturates. This is achieved by a proper selection of u_{lo}, u_{hi}, k_2. The actuator will, therefore, be assumed linear because it is used only in its linear range. u_{lo}, u_{hi} must be chosen to allow steady-state operation in the linear domain at the actuator [7].

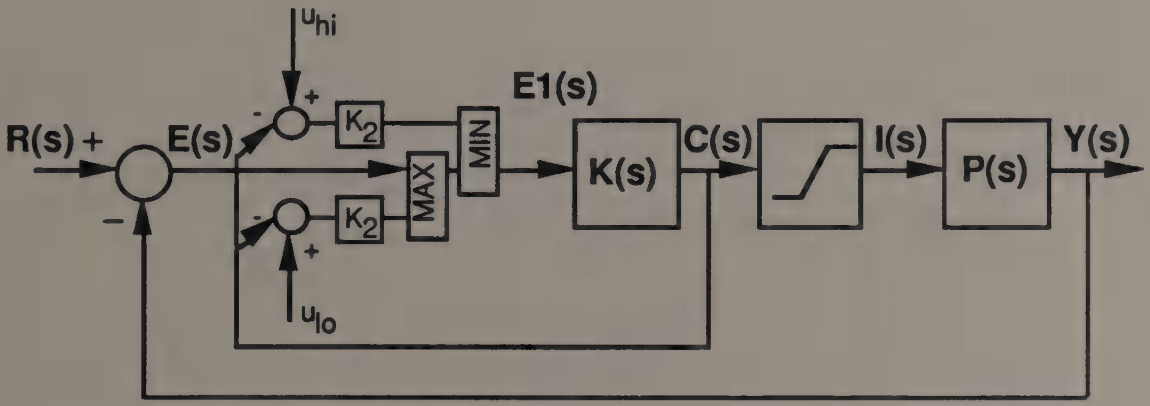


Figure 10. Override Signal Configuration by Glattfelder

Another configuration of the OS category of CAW systems is given by Buckley [8] and is shown in Figure 11. In this configuration, the override signals are switched-in discretely by operator control based on knowledge of the system performance under given conditions and on measurements of the system states as given to the operator. This configuration is generally used in systems that are slow and require an amount of operator intervention such as nuclear plants or chemical processing plants.

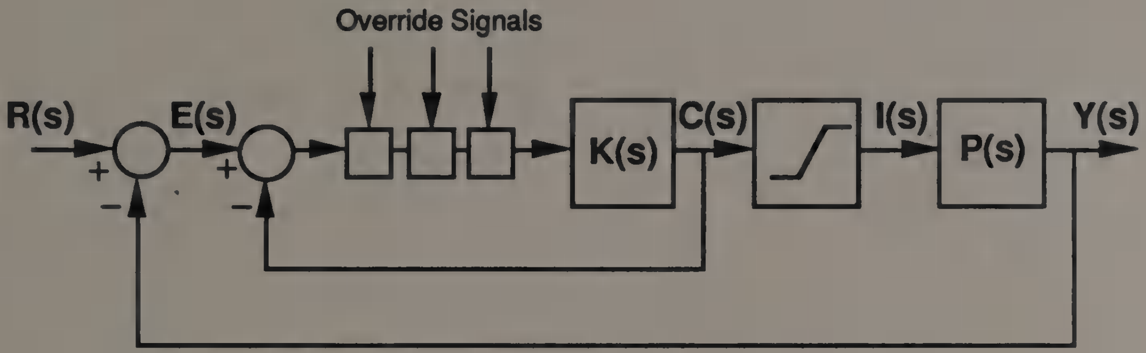


Figure 11. Override Signal Configuration by Buckley

2.3 Discrete Antireset-Windup (ARW) Configurations

The previous configurations, designed to solve the problem of ARW, are commonplace among the existing approaches used in continuous or analog controls design. However, very little has been written in the controls literature about the application of these, or similar techniques, to the problem of reset-windup in discrete systems. A discussion of a modified CAW system is described for discrete systems by Glattfelder [6]; however, the high gain feedback (X) is not included in that configuration. In Chapter 3, this paper will describe discrete implementations of the CAW configuration using a case study of a system presented by Doyle and Smith [9], and the OS configuration using a case study of a system presented by Glattfelder [7]. Discrete versions of each of these systems will be designed and the performance of the discrete designs will be analyzed in order to determine the important design criteria. The design methods derived for these simple systems will then be applied to a practical industrial application for a system.

CHAPTER 3

TECHNICAL DISCUSSION

3.1 Simple ARW Systems

3.1.1 CAW System

Doyle and Smith [9] describe an ARW system of the CAW type. Their paper describes ■ continuous system and its susceptibility to reset-windup caused by nonlinear saturation within the system. A CAW modification to the system is then made and the elimination of the reset-windup is shown. A brief review of Doyle and Smith's results will be described in this section; a discrete design of the system will be presented, aspects of the discrete design will be discussed, and an analysis of the design criteria will be provided.

3.1.1.1 Continuous CAW System

The continuous system presented by Doyle and Smith is shown in Figure 12. For this system, the transfer function of the controller, K(s), is given as

$$K(s) = \frac{2}{s + 0.1} \tag{6}$$

and of the plant, P(s), as

$$P(s) = \frac{s + 0.1}{2s} \quad . \tag{7}$$

The system is drawn somewhat differently in Doyle and Smith than in Figure 12 which presents an equivalent system consistent with the nomenclature of this paper.

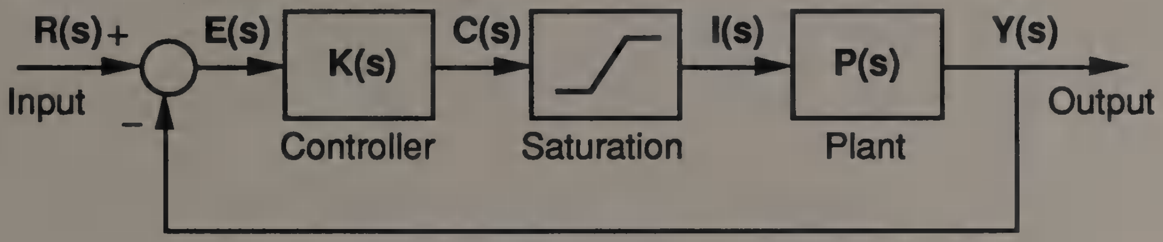


Figure 12. Continuous System by Doyle and Smith

The linear open-loop transfer function (without saturation) is

$$L_{ol} = P*K = \frac{1}{s}\ , \tag{8}$$

while the linear closed-loop transfer function of this system is

$$L_{cl} = \frac{P*K}{1 + P*K} = \frac{1}{s + 1}\ . \tag{9}$$

The forced response of *y(t)* and *c(t)* to a unit step input, *r(t)*, with all states set to initial values of zero is shown in Figure 13.

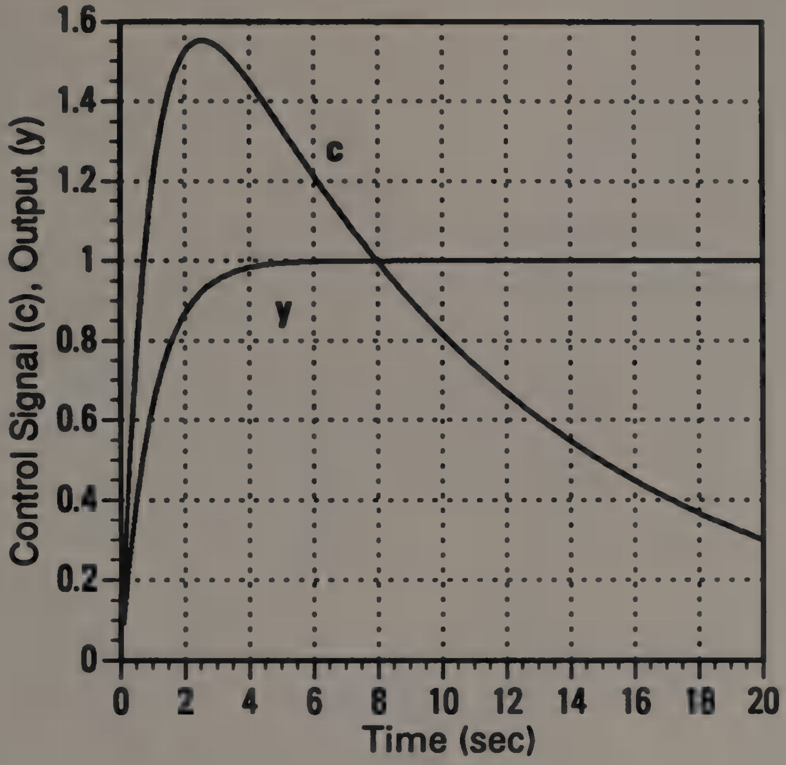


Figure 13. Linear Step Response of Doyle System

The response of the system with the saturation included, is shown in Figure 14. A fourth order Runge-Kutta simulation was developed to model the results and is discussed in Appendix B. Notice the nonlinear effect on the output *y(t)* due to this simple saturation. The output exhibits a classic symptom of reset-windup: large overshoot of the output caused by the large overshoot of the control signal. Note that K(s) need not contain pure integrators to produce windup, only relatively slow dynamics that are driven by the error when the system is in saturation [5].

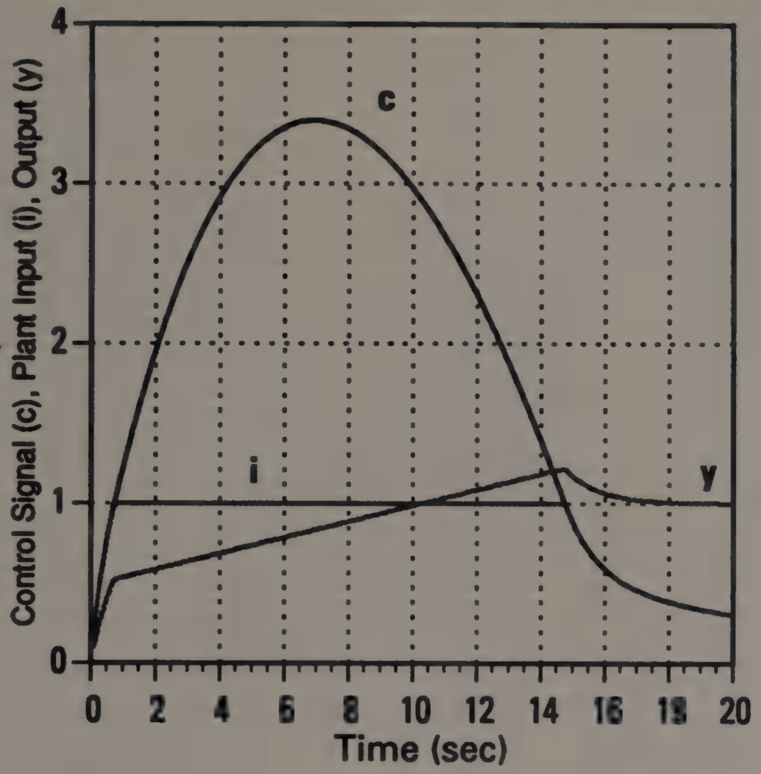


Figure 14. Step Response of Doyle System with Saturation

In order to eliminate windup, improve system performance and reduce overshoot, Doyle and Smith propose an ARW modification based on the CAW configuration discussed in Section 2.2.1. This configuration (Figure 15) uses high-gain feedback to modify the error signal.

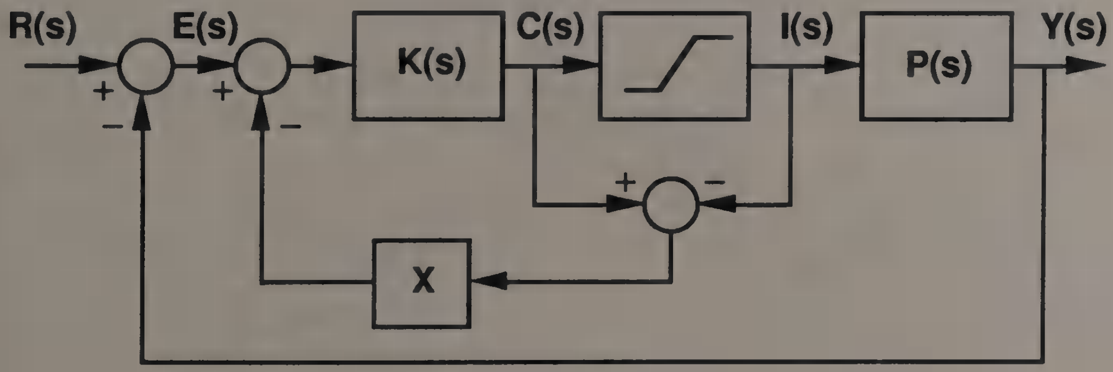


Figure 15. Doyle ARW Configuration for Simple System

The operation of the CAW compensation is fairly straightforward. The difference between the input to the plant, *i(t)*, and the output of the controller, *c(t)*, is measured and fed back through a gain, X, into the compensator, K(s). Windup is prevented if the associated loop transfer function $L_X(s) = K(s)*X(s)$ has a gain and bandwidth much higher (> 10) than that of L(s), the forward loop transfer function $L(s) = K(s)*P(s)$. The effect of the CAW compensation is to smooth the input to the controller when the system goes into saturation and the system is maintained within the linear region of operation (thus preventing windup).

The effectiveness of Doyle and Smith's CAW modification is demonstrated by a specific example. If a gain of 10 is chosen for X, the loop transfer function L_X is

$$L_X = K*X = \frac{20}{s + 0.1}. \tag{10}$$

The response of the system with the CAW modification is shown in Figure 16. The Runge-Kutta simulation which produced these results is discussed in Appendix B.

The performance improvement resulting from the ARW modification is evident. The ARW reduces the overshoot from 35% to 0%, thus eliminating overshoot from the output *y(t)*. This also reduces the settling time from three times that of the system without saturation to only two times that of the system without saturation. As can be seen, it is the CAW's effect on the control signal that improves the plant performance. The CAW compensation keeps the control signal at, or very close to, the saturation limit thus keeping it from "winding up". Another significant advantage to the CAW, which will be discussed in more detail in Section 3.2, is that the CAW produces system performance which is very well behaved with regard to overshoot. The system will exhibit a consistent overshoot despite the magnitude of the step change to the system.

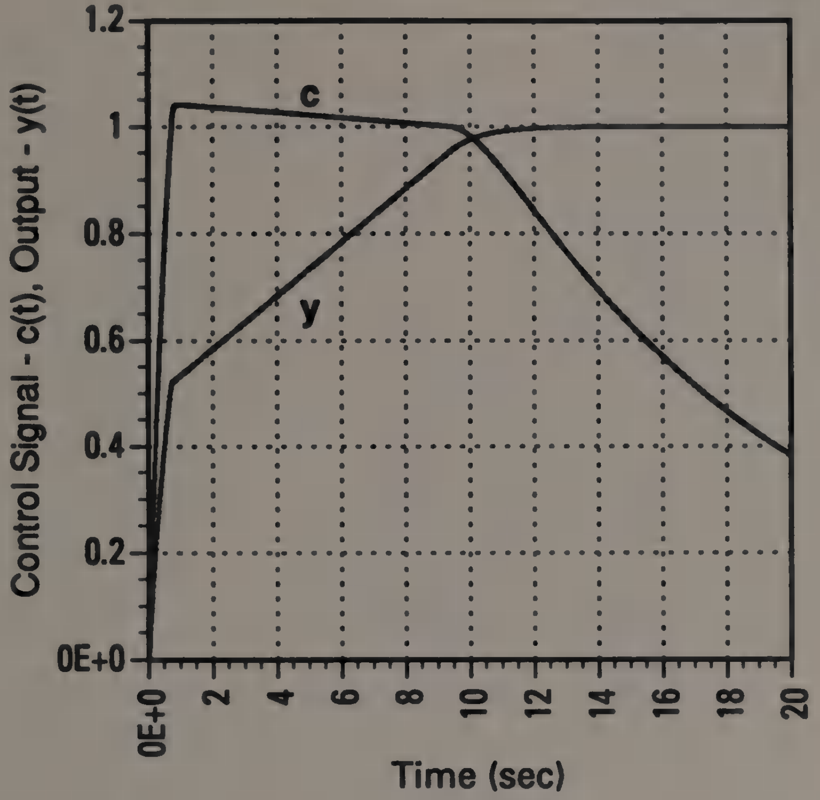


Figure 16. Step Response of Doyle System with Saturation and CAW Compensation

3.1.1.2 Design of a Discrete CAW System

There are many ways to approach the design of a discrete or digital controller, ranging from direct filter design in the z-domain to state-space methods. Design of a digital control system using transform techniques (design by discrete equivalent) is a viable and popular technique. Various methods exist for design by discrete equivalent and some are discussed in Appendix A. The technique is used here to design a discrete controller for the Doyle and Smith system and has been used to create a discrete model of the plant for simulation and analysis purposes.

The bilinear transform was used to design the discrete controller using the Doyle and Smith analog controller. The hold-equivalence transform method was used to create a discrete model of the plant using the original Doyle and Smith plant as a baseline. Other techniques such as the pole-zero mapping method were also used, yielding similar results, but are not discussed here. The discrete design of the system is shown in Figure 17.

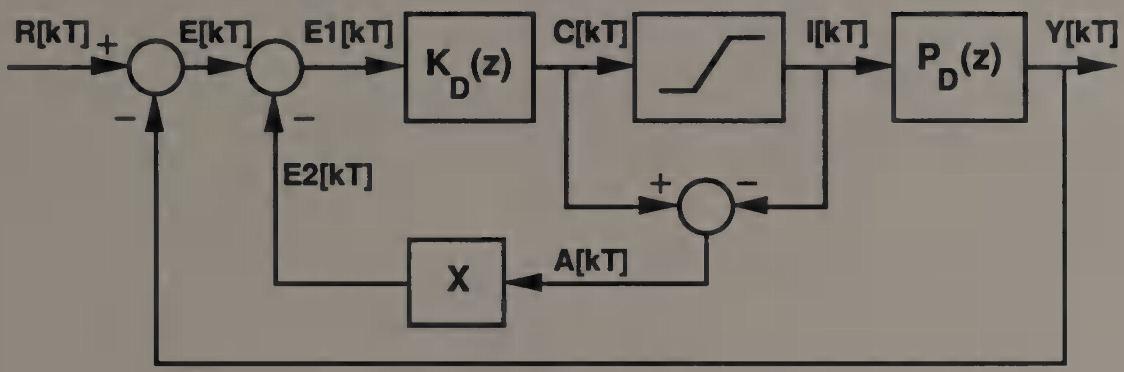


Figure 17. Discrete Design for CAW System

The compensator K(s) has been replaced with an equivalent discrete controller $K_D(z)$. The plant has been replaced with an equivalent discrete plant model $P_D(z)$. The high-gain feedback element, X, is retained. The controller $K_D(z)$ has been designed by applying the bilinear transform to the transfer function of the original compensator. In the compensator's continuous-time transfer function, written in terms of s, the substitution

$$s = \frac{2(z-1)}{T(z+1)} \tag{11}$$

is made for each s found in the equation. Given that the Doyle and Smith controller is stable and has the transfer function

$$K(s) = \frac{2}{s + 0.1} \tag{12}$$

the discrete controller is then found from the bilinear transform to be

$$K_D(z) = K1 * \frac{1 + z^{-1}}{1 + az^{-1}} \tag{13}$$

where

$$K1 = \frac{(2T)}{0.1T+2} \cdot \tag{14}$$

and

$$a = \frac{(0.1T-2)}{0.1T+2} . \tag{15}$$

A discrete model of the plant was developed for simulation purposes using the hold equivalence (step-invariance) transform method as discussed in Appendix A. Given that the original Doyle and Smith plant model is

$$P(s) = \frac{s + 0.1}{2s} , \tag{16}$$

the discrete plant model is then determined from the original transfer function of the plant as

$$P_D(z) = (1-z^{-1})\, Z\left\{\mathfrak{L}^{-1}\left\{\frac{P(s)}{s}\right\}\right\},$$

$$= K2 * \frac{1 + bz^{-1}}{1 - z^{-1}} \tag{17}$$

where

$$K2 = 0.5, \tag{18}$$

and

$$b = \frac{(0.05T-0.5)}{0.5} . \qquad (19)$$

With $K_D(z)$ and $P_D(z)$ now determined, the complete design for the discrete CAW system is shown in Figure 18.

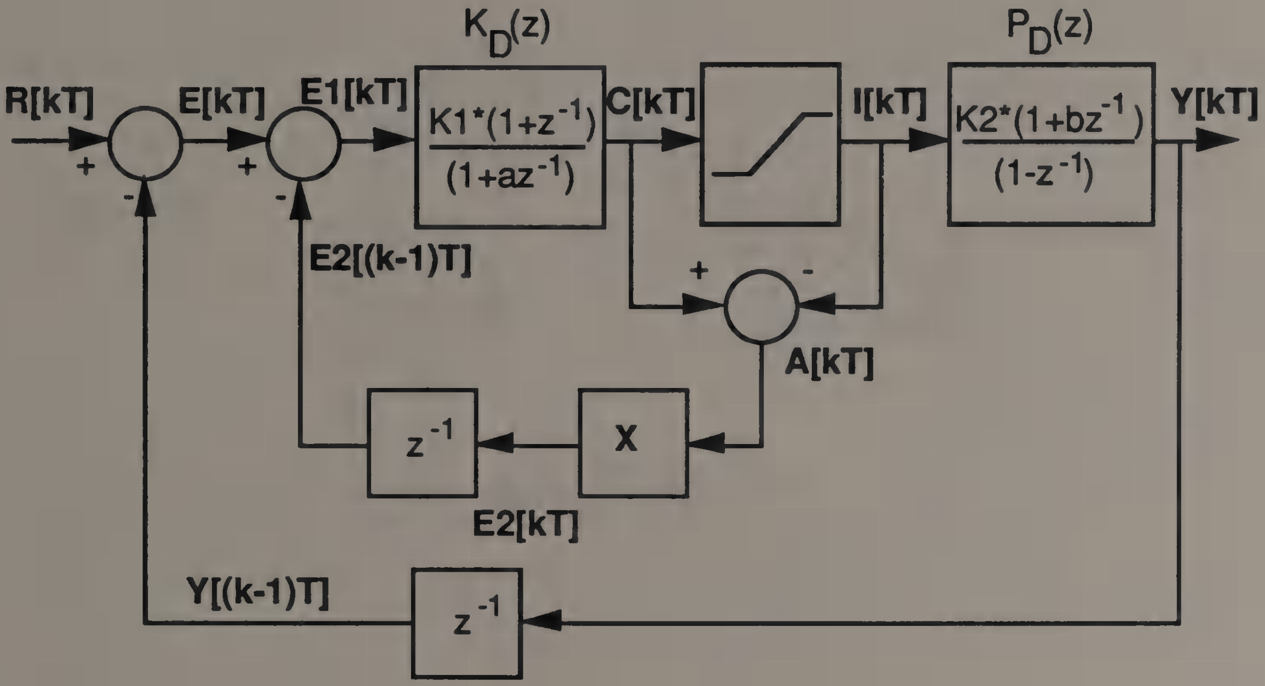


Figure 18. Discrete Design for Doyle CAW System

A discrete simulation of the above system was developed to analyze and compare the performance of the discrete and continuous systems. The details of this simulation are discussed in Appendix B. Step responses were used as a measure of the accuracy of the design and to determine system behavior, as system characteristics were varied.

3.1.1.3. Comparison of Continuous and Discrete CAW Designs

System performance was analyzed using the simulation developed without changing the design of the controller or plant. The only parameter within the continuous-time system that can be varied is the feedback gain X. The only parameters within the discrete-time system that can be varied are the feedback gain X and the sample time T. A study of system step response versus X and T was conducted on the discrete simulation in order to establish design criteria for these parameters. An interesting phenomenon was discovered during this study which was evident in the discrete system but did not occur in the continuous system. This phenomenon is referred to as "chatter", a high-frequency/low-amplitude (small signal) oscillation of the output.

With constant sample time, examinations of step responses for varied values of X revealed a threshold value at which chatter occurred in the output. A specific example with X = 10 and T=0.01 is provided to illustrate this phenomenon. The step response of the discrete system is shown in Figure 19. Notice that this closely matches that of the continuous system with the same gain in Figure 16. Comparison of this step response to that of the discrete system with X = 500 and T = 0.01 in Figure 20 shows the chatter phenomenon. Notice that the output signal "chatters" between the original trajectory of the output and some other amplitude. It will be shown that the frequency and amplitude of this chatter is a function of the values of T and X for a given system.

In order to show that the original continuous system is not susceptible to this chatter, observe the step response of the continuous system with a gain, X = 500, in Figure 21. Simulation runs with values of X as high as 100,000 were run and there was no evidence of chatter in the continuous system for any of these values. This is shown in Figures 22 and 23 which are plots of chatter amplitude versus gain X for the discrete and continuous designs respectively.

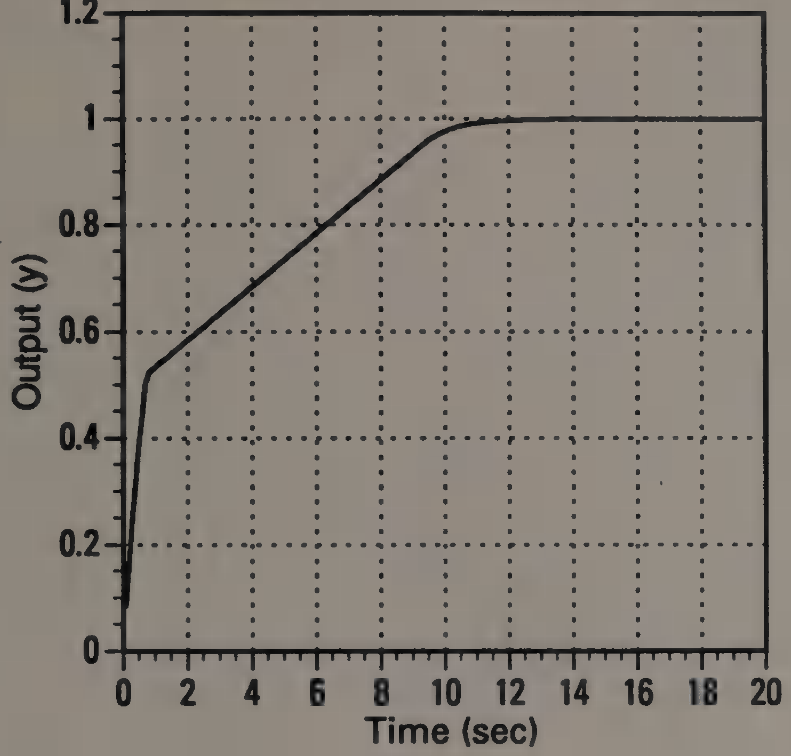


Figure 19. Step Response of Discrete CAW System (X=10, T=0.01)

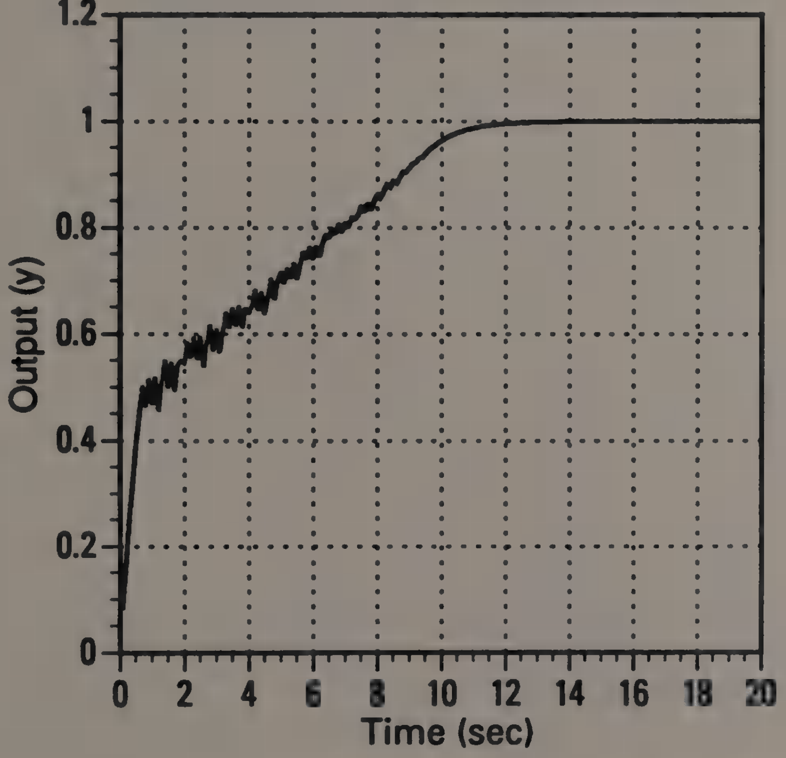


Figure 20. Step Response of Discrete CAW System (X=500, T=0.01)

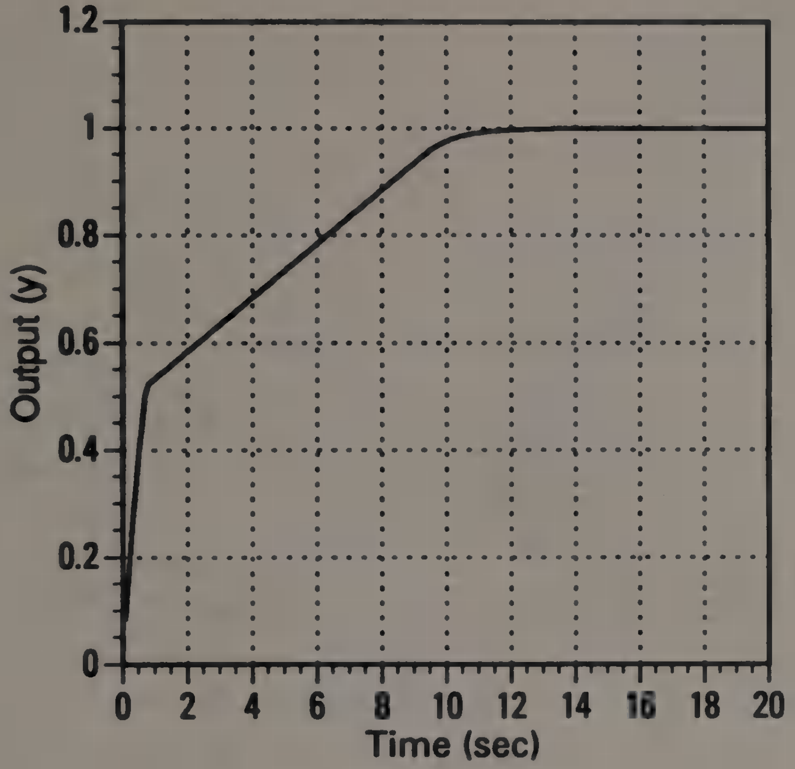


Figure 21. Step Response of Continuous CAW System (X=500)

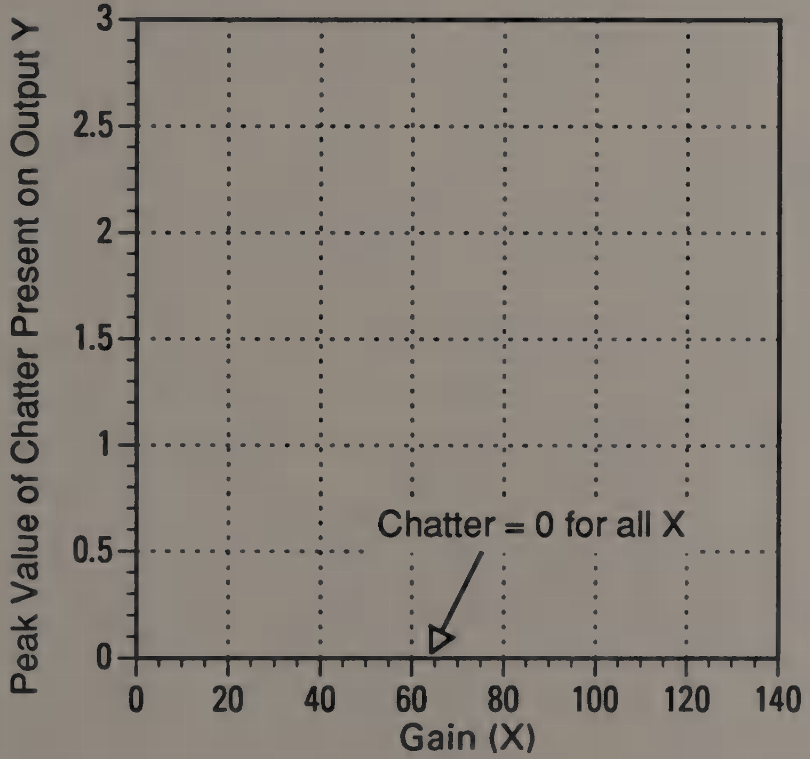


Figure 22. Chatter Amplitude vs. Gain X - Continuous CAW System

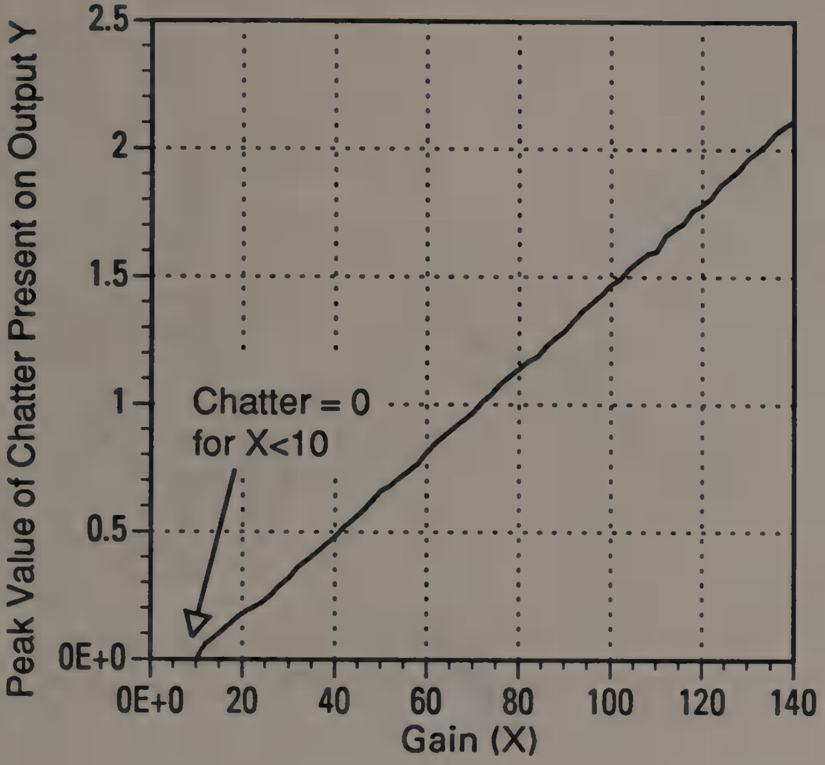


Figure 23. Chatter Amplitude vs. Gain X - Discrete CAW System (T=0.01)

The chatter appeared for values of X above a threshold value (with fixed T). The chatter point and the amplitude of the chatter increased linearly with X above that threshold. The value of the threshold was found to be related to X and T: the chatter point decreased as either X or T was decreased. For instance, the chatter point for T=0.05 seconds is about X=21 while for T=0.01 seconds, it is about X=100 . The next section describes the analysis performed in order to determine the cause of the chatter, and to quantify the chatter threshold in terms of X and T.

3.1.1.4 Analysis of Chatter versus Feedback Gain and Sample Period

It can be seen from the system block diagram (Figure 17) that the system consists of two closed loops: the outer closed loop that controls the plant itself, and the inner loop consisting of the CAW loop around the controller. Careful examination of the output waveform shown in Figure 20 divides the response into three sections, as shown in Figure 24. Chatter is present only in Section II which corresponds to the time when antireset-windup compensation is present. The chatter is thus connected with either the inner loop or a combination of the inner and outer loops, and is not a phenomenon of the outer loop itself.

Several analysis techniques were used to determine the source of the chatter including the Circle Criteria by Zames [10], root locus, and bode diagrams. Through the use of these techniques, it was determined that the chatter was connected with the inner loop only and not a combination of the inner and outer loops. This conclusion is explained through an analysis of the design of the inner loop when the system is in saturation as shown in Figure 25.

The block diagram of the inner loop can be rearranged using a technique first described by Zames [10] for use with the Circle Criteria. A nonlinear system is first separated into its linear and nonlinear parts. The linear portion of the inner loop is shown in Figure 26, within the shaded box, while the nonlinear element is shown as a feedback element around the linear portion. During saturation, the contribution to *i[kT]* from the nonlinearity block is constant. The response of the inner loop during this time can thus be analyzed by looking only at the linear portion of the inner loop. The transfer function from *I* to *C*, T_{IC}, is derived to be

$$T_{IC}(z) = \frac{K1*X*z^{-1}(1+z^{-1})}{1 + z^{-1}(a+K1*X) + z^{-2}(K1*X)} \; . \qquad (20)$$

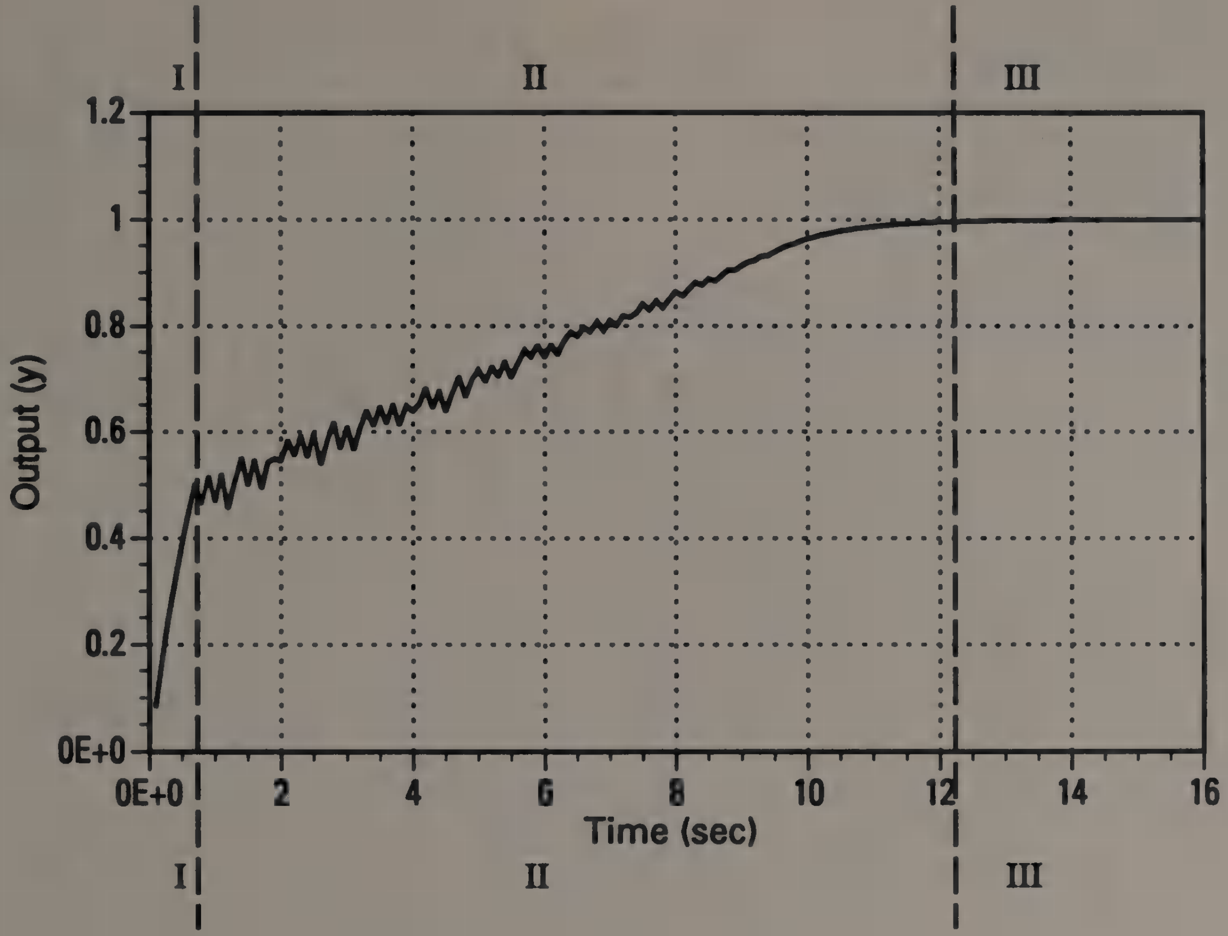


Figure 24. Step Response of Discrete CAW System (X=500, T=0.01)

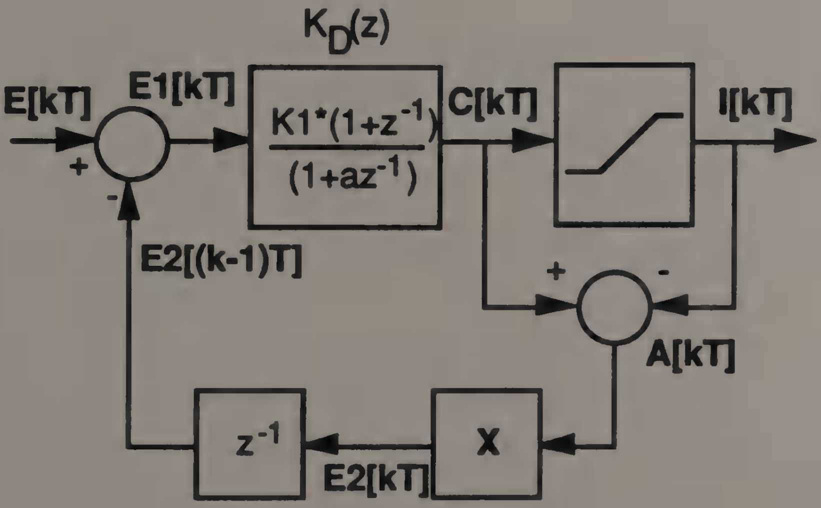


Figure 25. Inner Loop of Discrete CAW System

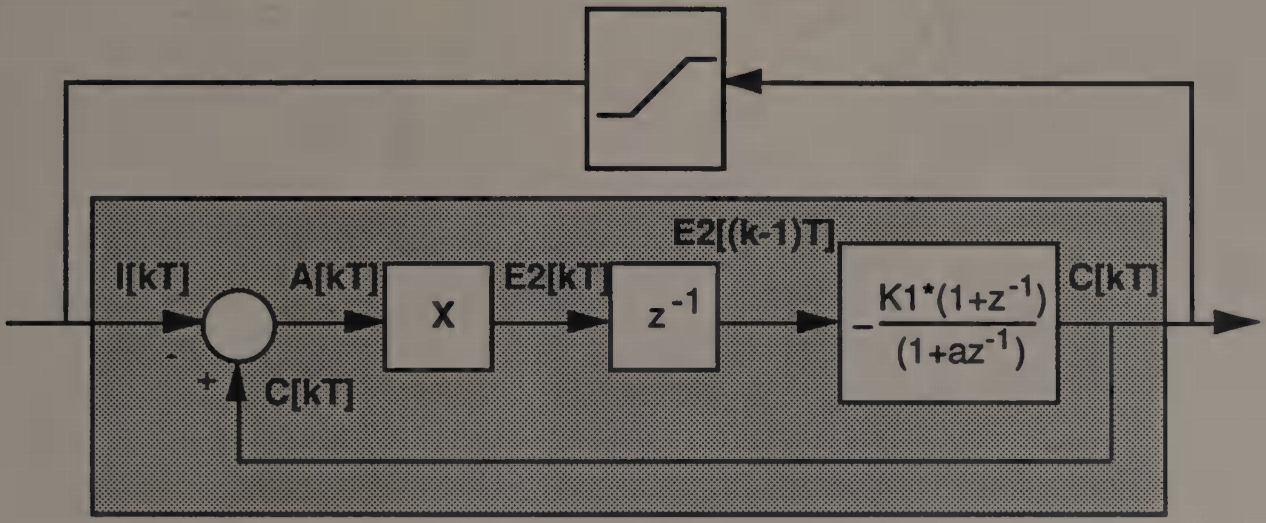


Figure 26. Inner Loop of Discrete CAW System - Zames Form

The inner loop, as described in Zames' form, lends itself to analysis by the root locus technique for discrete systems. The response of the closed inner loop can be described in terms of the closed loop poles and zeros by plotting the open loop forward transfer function and varying the forward path gain, X. The root locus of equation (20) is shown in Figure 27. Stability is maintained in a discrete system as long as the poles of the closed loop remain within the unit circle. The response is shown normalized to T because different values of X and T do not change the shape of the root locus but only the actual roots at a given position along the locus in this particular system. The curve could just have easily been normalized to X. The root locus shows that the inner loop will become unstable for values of X and T, placing the poles of the closed loop transfer function T_{ic} outside the unit circle. This is the cause of the chatter in the overall system. When the gain X of the inner loop becomes large enough for a given T, the phase difference between *I* and *C* becomes greater than 180 degrees. The inner loop no longer operates with negative feedback but, instead, has positive feedback. Normal operation of the loop is such that the feedback signal *C* minus *I* fed back through the gain X modifies *E1* in such a way as to hold the output of the compensator just at the level of saturation.

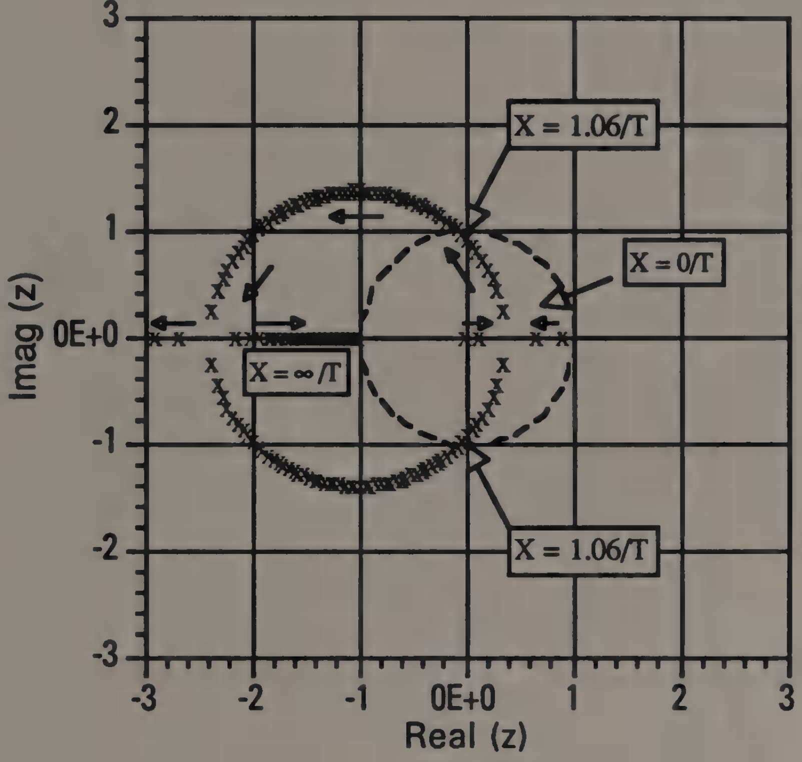


Figure 27. Root Locus of Discrete CAW Inner Loop

When X gets too large, however, the controlling signal *E3* becomes unstable, overshooting the desired control value, and the inner loop causes the compensator output to return to the linear, unsaturated region. At the next sample time, the inner loop is not in the system and the compensator realizes that the trajectory of the control signal is in the wrong direction for the error signal present. The compensator then issues a control signal which drives the system back into saturation. The inner loop once again becomes part of the system, overcompensates, and the process continues. This is the chatter phenomenon being observed. The overcompensating aspect of the CAW signal is what leads to chatter in the system. The chatter is characterized by an oscillating turn-on and turn-off of the

CAW compensation signal. This chatter is seen by examining unit step responses of the inner loop for various values of X as shown in Figures 28 through 32.

The step response of the inner loop corresponds to the response of the CAW compensation signal *e2[kT]* in Figure 18. Figures 31 through 35 show the simulation run outputs for the signals *e2[kT]* and *y[kT]* for the same values of X and T as in Figures 28-32. The time scales are shown enlarged so that the response of *e2[kT]* for each sample period can be seen.

The root locus predicts that the chatter point should be at approximately 106 for T = 0.01. This matches the simulation values previously obtained (chatter point at about 100-110) and confirms that the chatter is caused by instability of the inner loop which results in a overshooting of the compensation signal.

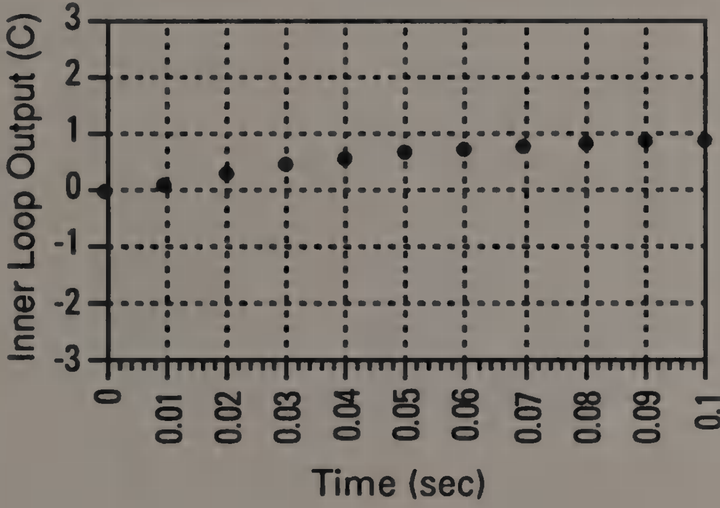


Figure 28. Step Response of Discrete CAW Inner Loop (X=10, T=0.01)

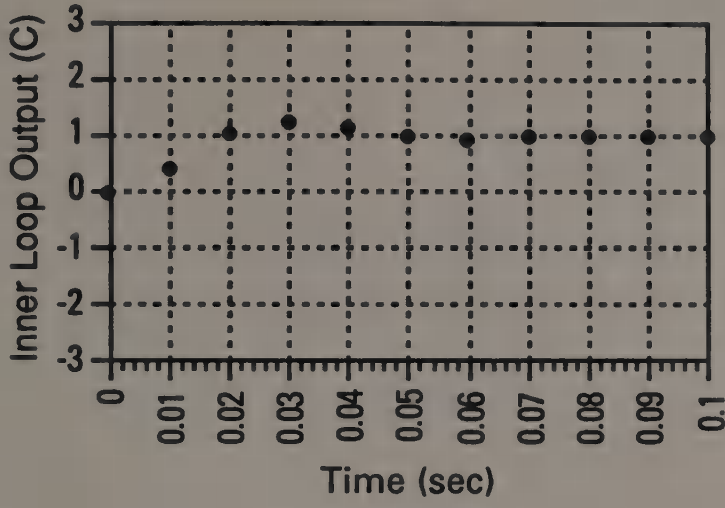


Figure 29. Step Response of Discrete CAW Inner Loop (X=40, T=0.01)

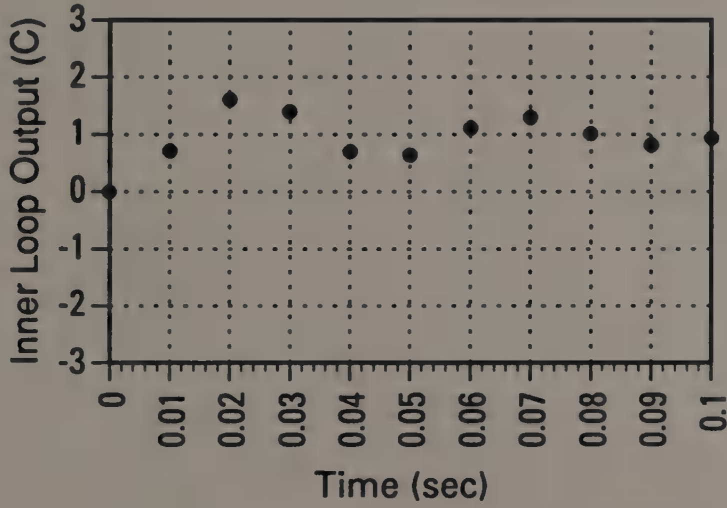


Figure 30. Step Response of Discrete CAW Inner Loop (X=70, T=0.01)

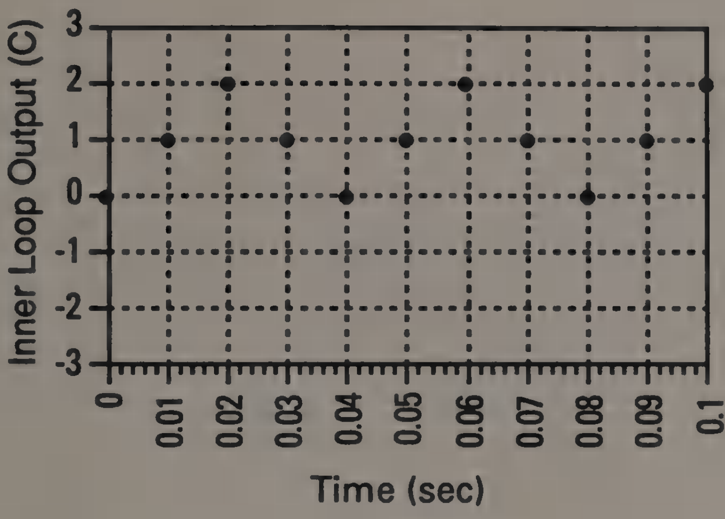


Figure 31. Step Response of Discrete CAW Inner Loop (X=100, T=0.01)

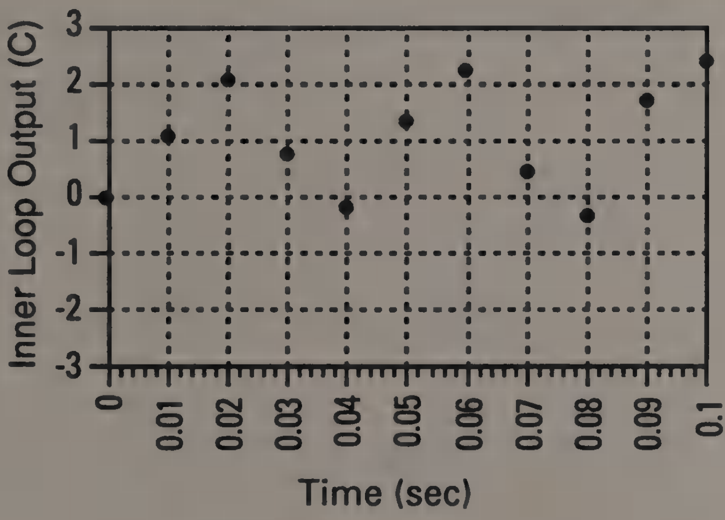
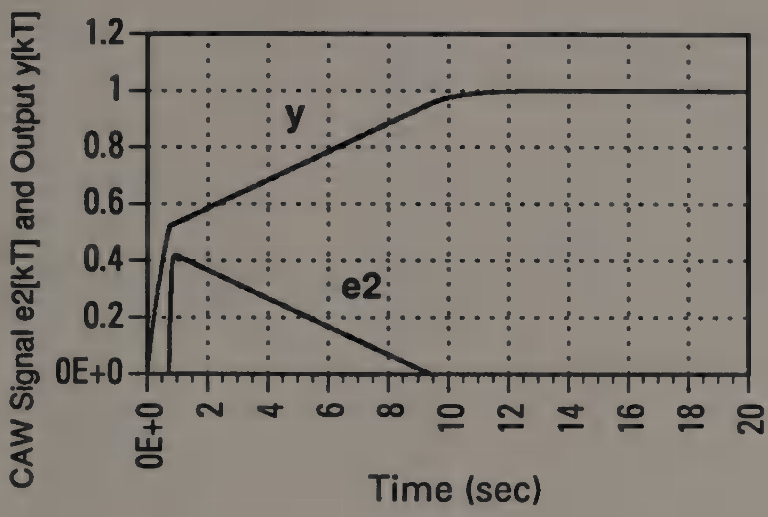


Figure 32. Step Response of Discrete CAW Inner Loop (X=110, T=0.01)



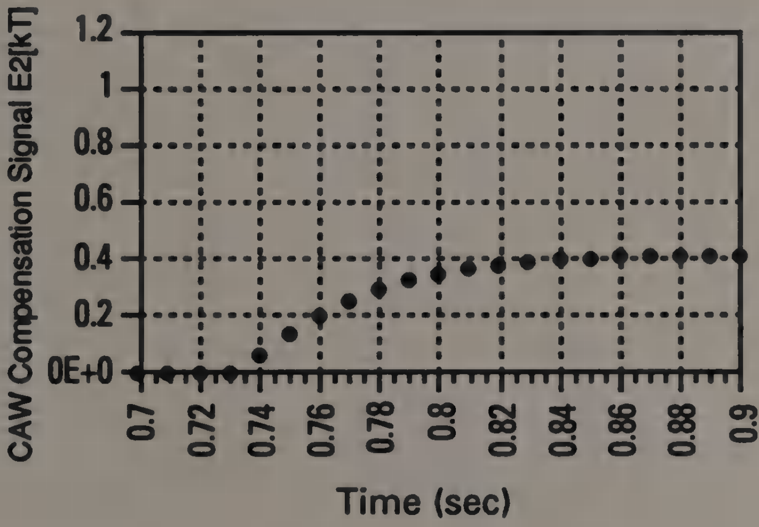
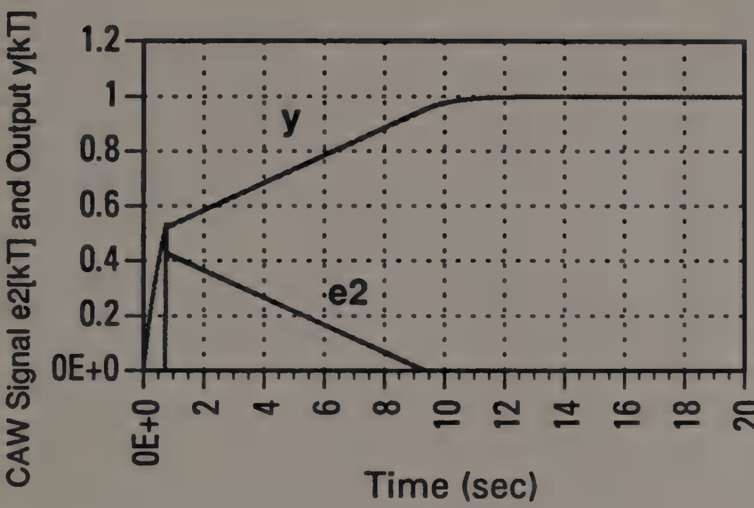


Figure 33. Step Response of Discrete CAW System (X=10, T=0.01)



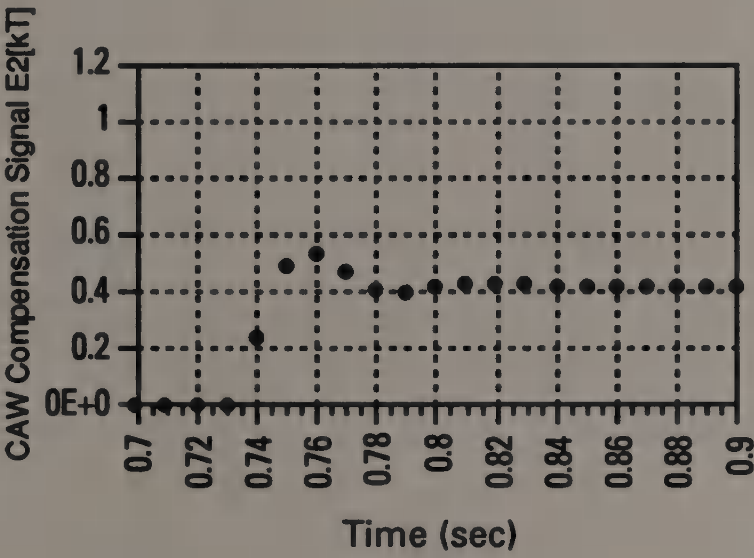
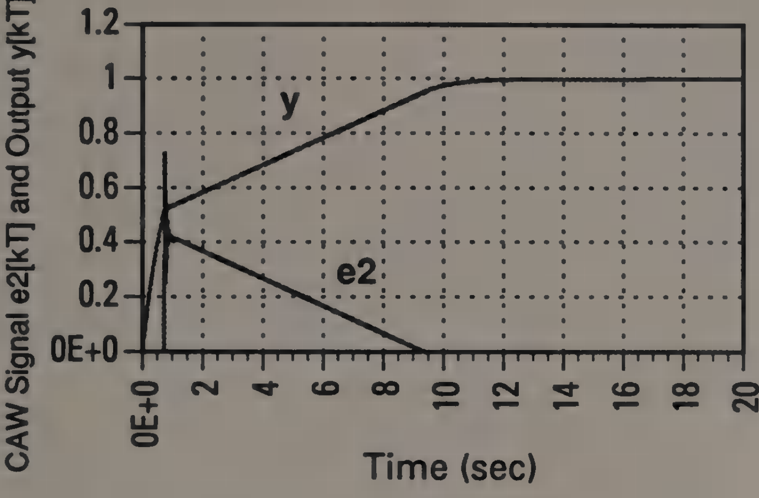


Figure 34. Step Response of Discrete CAW System (X=40, T=0.01)



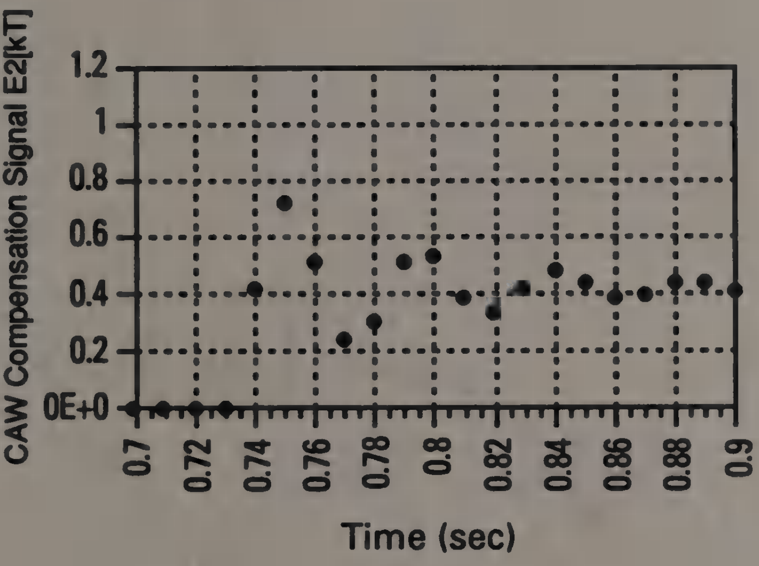
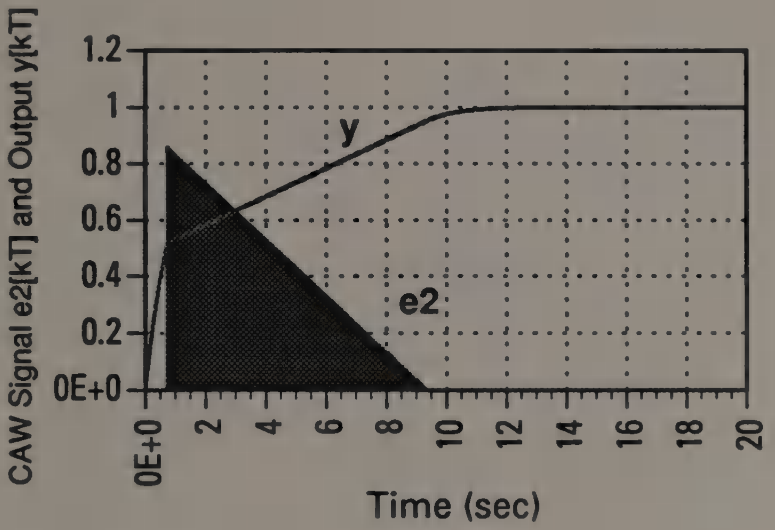


Figure 35. Step Response of Discrete CAW System (X=70, T=0.01)



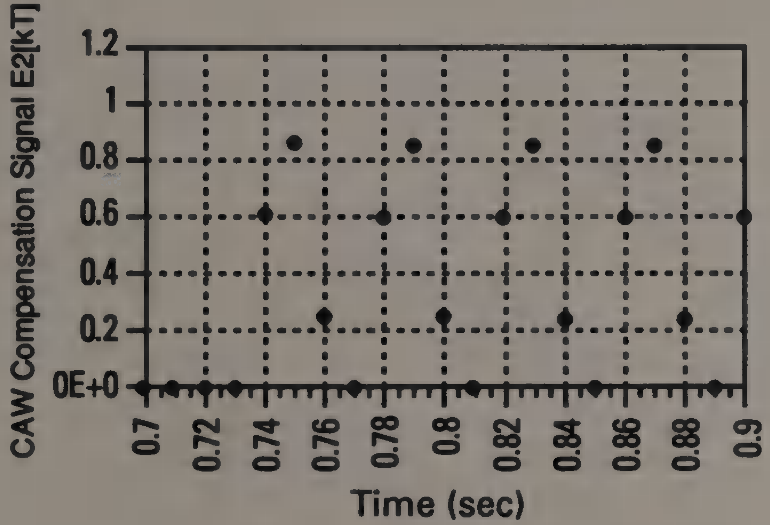
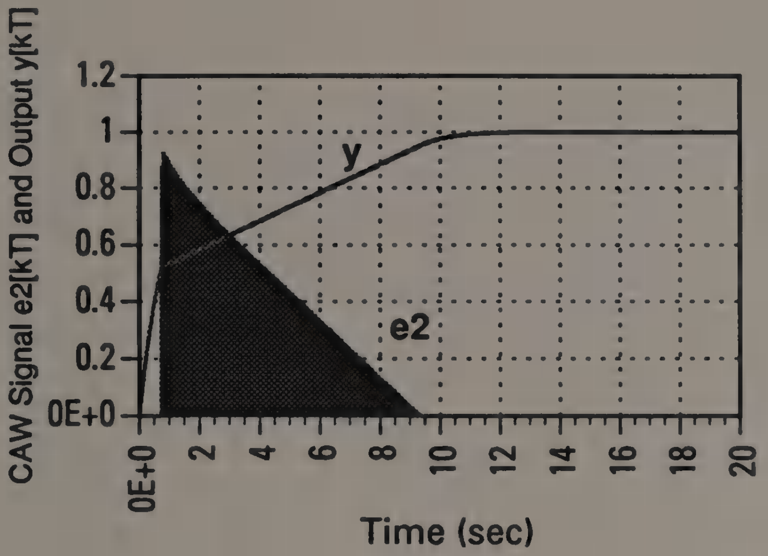


Figure 36. Step Response of Discrete CAW System (X=100, T=0.01)



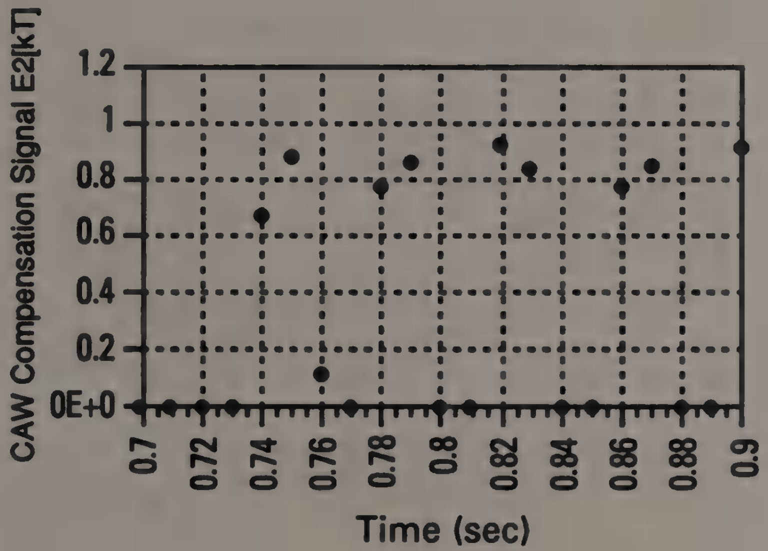


Figure 37. Step Response of Discrete CAW System (X=110, T=0.01)

continuous system remains stable, i.e. the poles remain in the left-half side of the s-plane and the control signal can never overshoot. Thus, chatter can never be realized with the original continuous system.

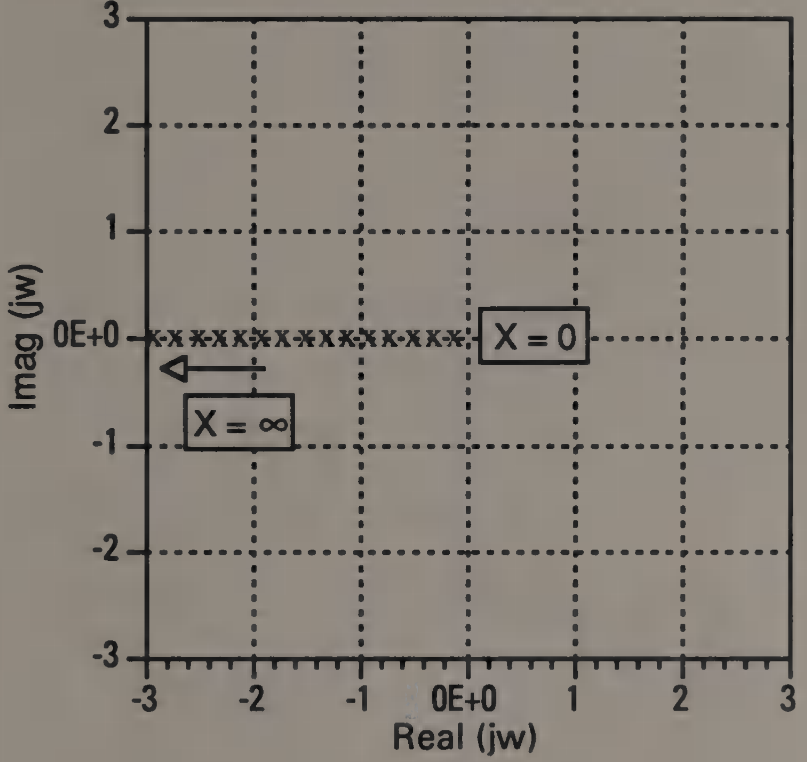


Figure 38. Root Locus of Doyle Continuous CAW Inner Loop

The reason for chatter appearing in the discrete system is examined in more depth by comparing the inner loops of the discrete and continuous designs and the associated closed loop transfer functions. The block diagram for the inner loop of the continuous design is shown in Figure 39. The block diagram for the inner loop of the discrete design is redrawn in Figure 40.

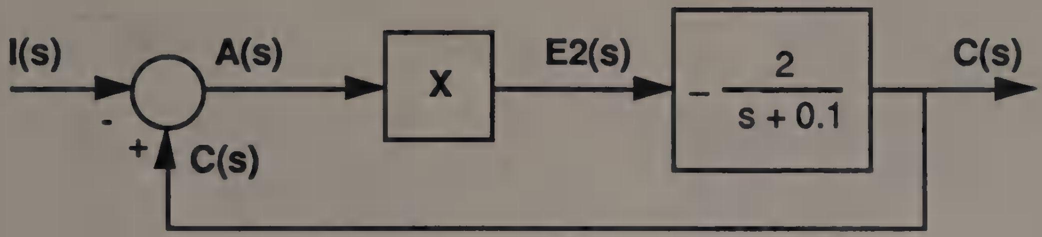


Figure 39. Inner Loop of Continuous CAW System

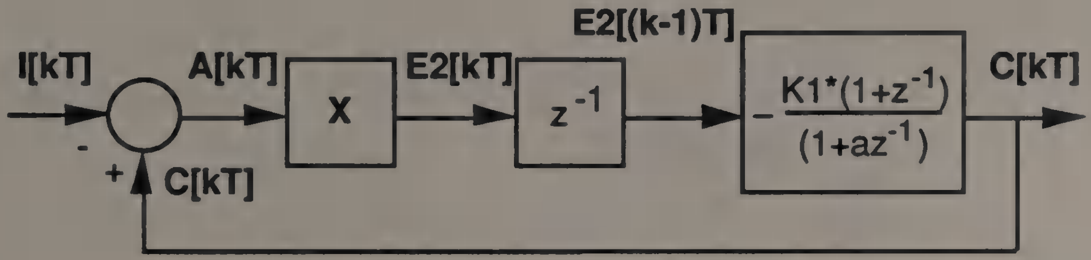


Figure 40. Inner Loop of Discrete CAW System

The transfer function for the continuous system is

$$T_{ic}(s) = \frac{2X}{s + (2X+0.1)} , \tag{21}$$

and for the discrete system it is

$$T_{ic}(z) = \frac{\frac{2TX}{0.1T+2}(z+1)}{z^2 + \frac{T(0.1+2x)-2}{0.1T+2} z + \frac{2TX}{0.1T+2}} . \tag{22}$$

The characteristic equation for T_{ic} in the continuous system is of order one, whereas it is of second order for the discrete system. A second order system (discrete inner loop) has the potential of becoming unstable, while a first order system (continuous inner loop) does not. Both inner loops are identical in order for the controllers because the bilinear transform of a continuous filter results in a discrete filter of the same order in the denominator. The extra pole in the discrete system results from the additional delay in the feedback path of the inner loop shown in Figure 18.

This delay is necessary in the discrete design because the controller cannot have *a priori* knowledge of the output state of the controller. A unit delay is therefore placed between the output of the controller and the input of the inner loop summing junction. The extra delay makes the chatter possible in the discrete design. This does not mean, however, that only discrete designs which implement the CAW modifi-cation are susceptible to chatter. A higher order compensator in a continuous system would show chatter as well under the proper conditions.

It is possible to demonstrate a continuous CAW system with chatter by redesigning the compensator in the original Doyle system to be of second order. Assume that K(s) in Figure 15 is given by

$$V^{-1}(s) = \frac{2}{s + 0.1} * \left\{\frac{20}{s + 20}\right\}^2 . \qquad (23)$$

Assume that the plant is described by equation (7). The root locus for the inner loop of this continuous system is shown in Figure 41.

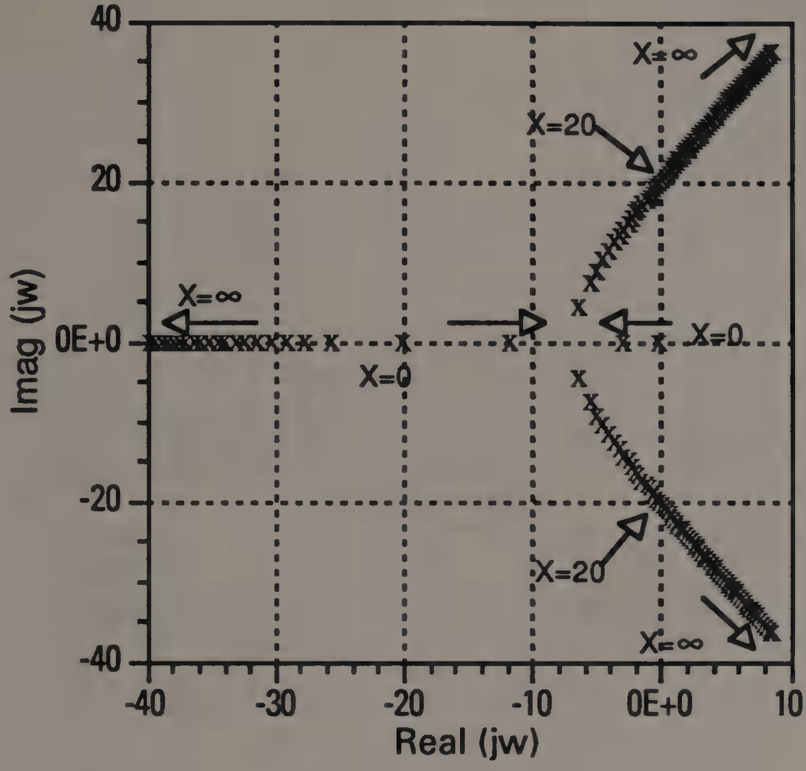


Figure 41. Root Locus of Complex Doyle Continuous CAW Inner Loop

The inner loop is stable up to a gain of X = 20. The system should, therefore, show no chatter below a value of 20 and should show chatter at values above 20. This result was confirmed using a Runge-Kutta simulation similar to the one used above (discussed in Appendix B). The step response for various values of X using this simulation is shown in Figure 42.

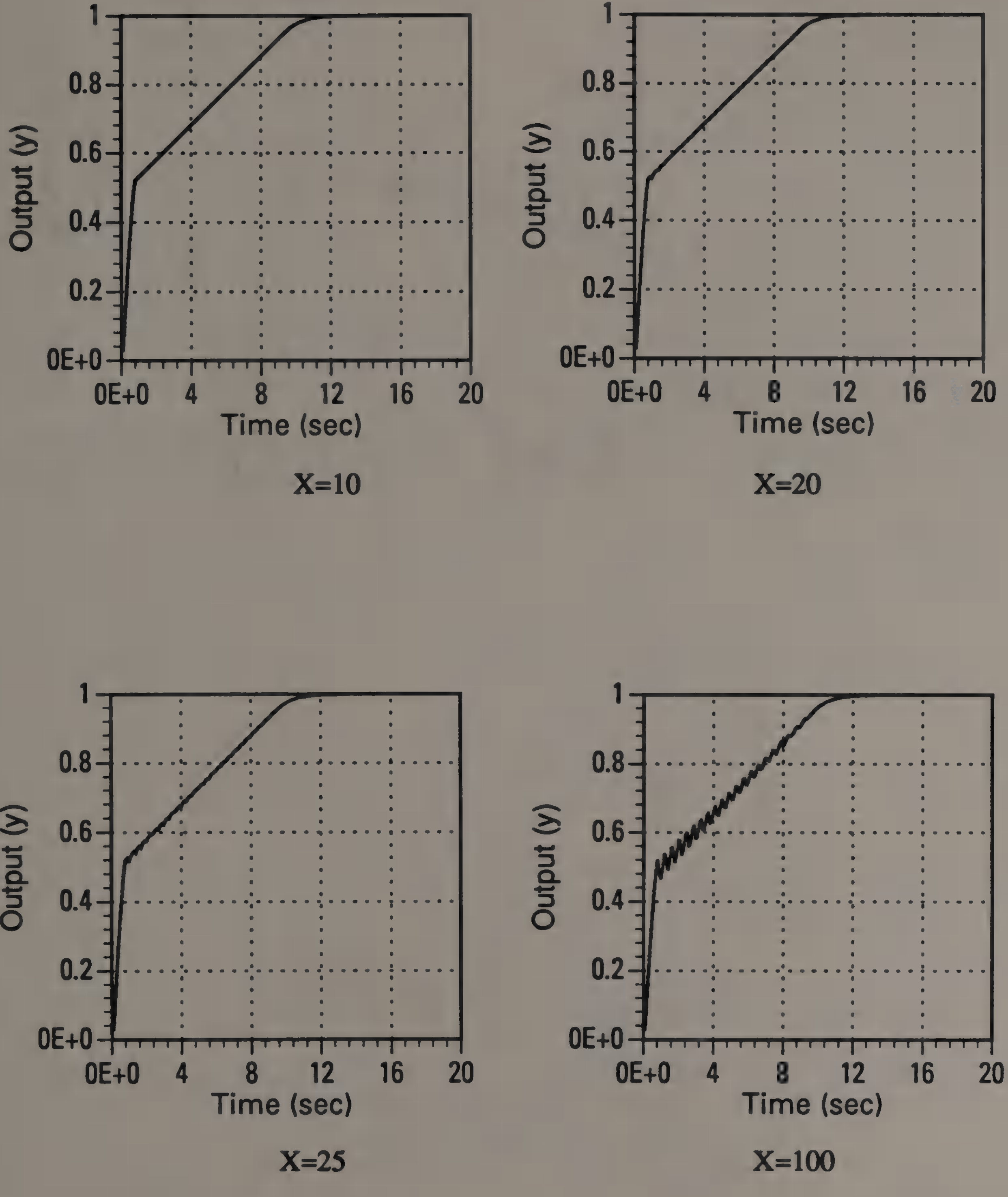


Figure 42. Step Responses of Continuous CAW System (Complex Compensator)

It is important to recognize that the chatter results when the control signal overshoots past zero when the inner loop is first activated into the system. This causes the inner loop to then turn off and the process of chatter is created. For the Doyle and Smith system, this occurs at instability of the inner loop. However, it need not necessarily be so: chatter can be seen in a system which is below instability for different controllers. To demonstrate this, assume that (in the discrete design of the CAW shown in Figure 18) the controller and plant are discrete versions of those used in the example in Chapter 2,

$$K(s) = K_1 + \frac{K_2}{s} \tag{24}$$

and

$$P(s) = \frac{1}{s}, \tag{25}$$

where $K_1 = 4$ and $K_2 = 16$. Applying the bilinear transform to K(s) and the step-invariant transform to P(s) yields

$$K_D(z) = K_1 + \frac{K_2 * T}{2} * \frac{1 + z^{-1}}{1 - z^{-1}} \tag{26}$$

and

$$P_D(z) = \frac{T * z^{-1}}{1 - z^{-1}}. \tag{27}$$

Figure 43 illustrates the design if the CAW is implemented around the integral portion of the controller only.

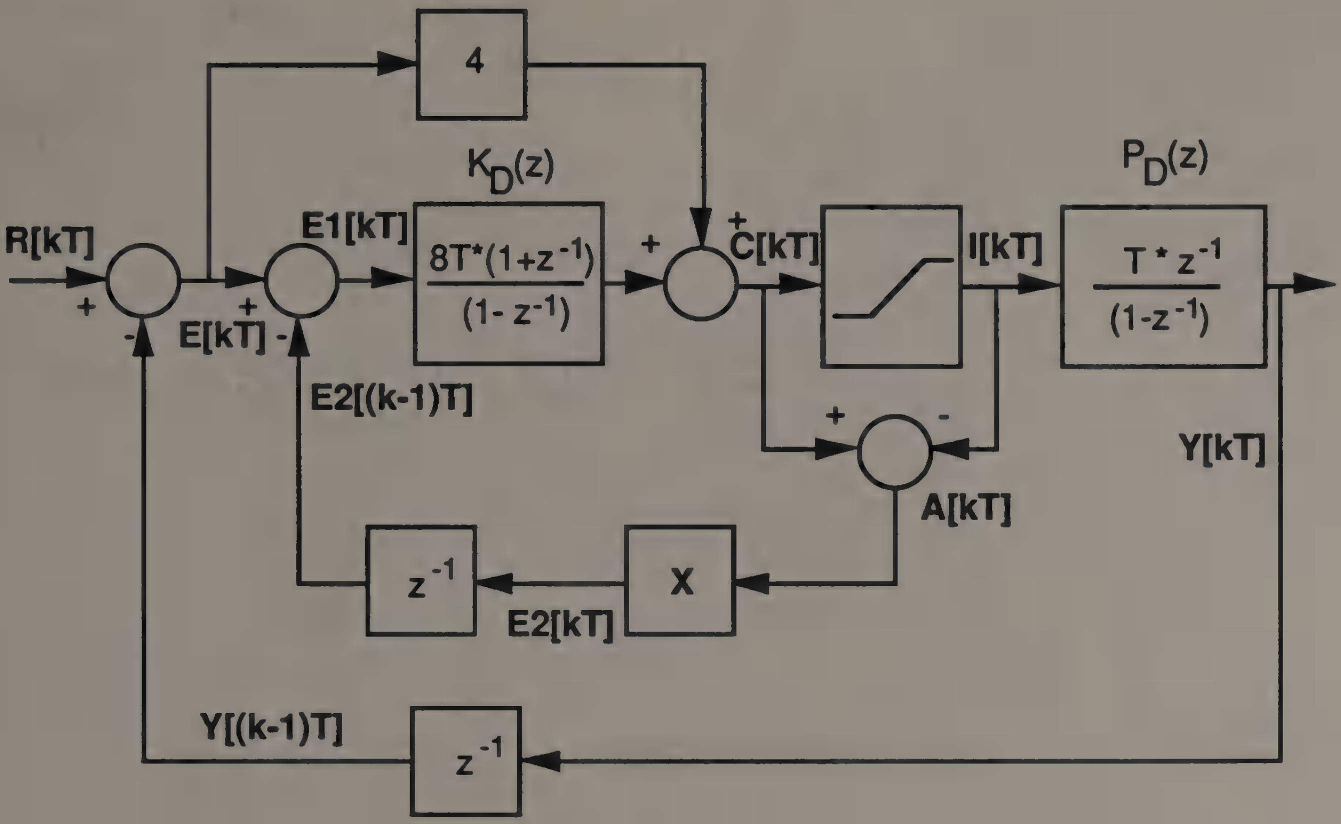


Figure 43. Discrete CAW Design for Example Controller and Plant

The inner loop for this system is shown in Figure 44 and its roots locus plot for sample period $T = 0.001$ is shown in Figure 45.

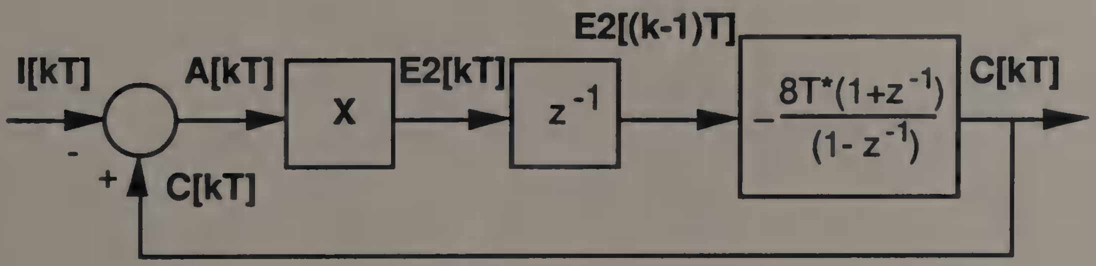


Figure 44. Inner Loop of Example System

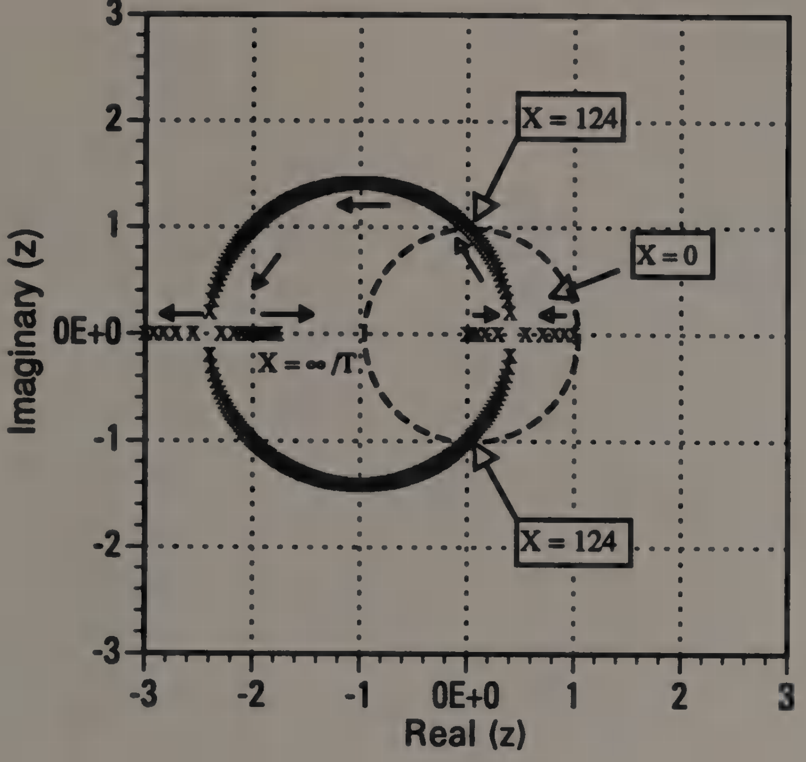


Figure 45. Root Locus Plot for Inner Loop of Example System

Although the discrete root locus shows that the inner loop of this system is stable, for T=0.001 until X = 124, the system begins to show chatter as low as X=30 as shown in Figure 46. The reason for this being that the inner loop of this design is second order as well. In this case, however, the control signal operates very close to zero as a result of the controller design and, as the gain X is increased, even small amounts of overshoot in the control signal, *e2[kT]*, cause it to go below zero thereby turning off the inner loop and causing chatter. This can be seen by looking at the control signal for X=20, 25, and 30 in Figures 47, 48, and 49. There is no overshoot below zero of the control signal for X=25 (and below). However, when X=30, the control signal overshoots the desired value and dips below zero. This shuts off the inner loop for a period of time, thus causing chatter. Although it is not as noticeable as it is with the previous controller and plant, and does not last as long, it is still chatter and should be avoided.

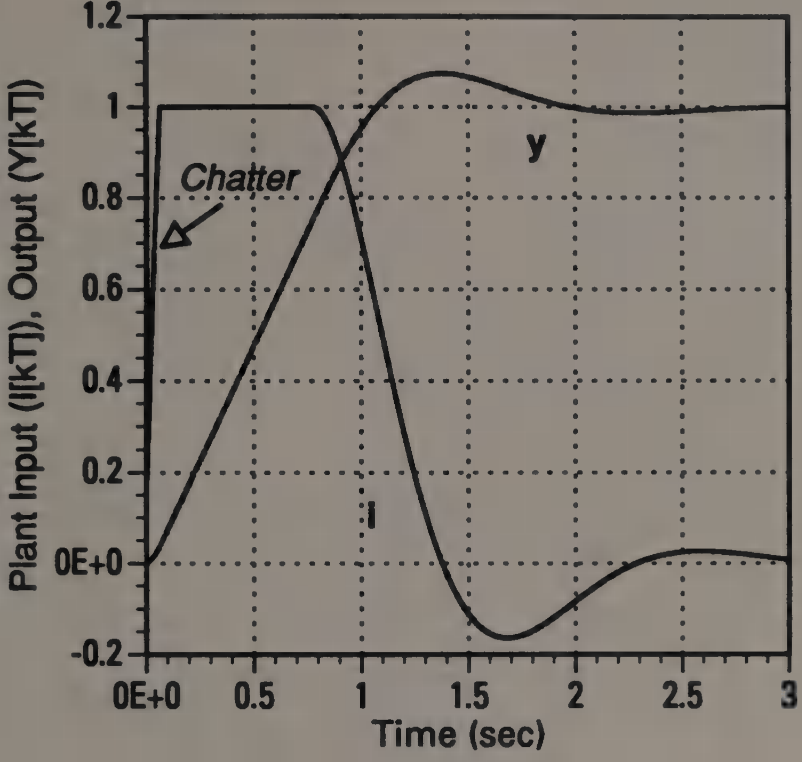


Figure 46. Step Response of Example System - Plant Input and Output Gain X = 30

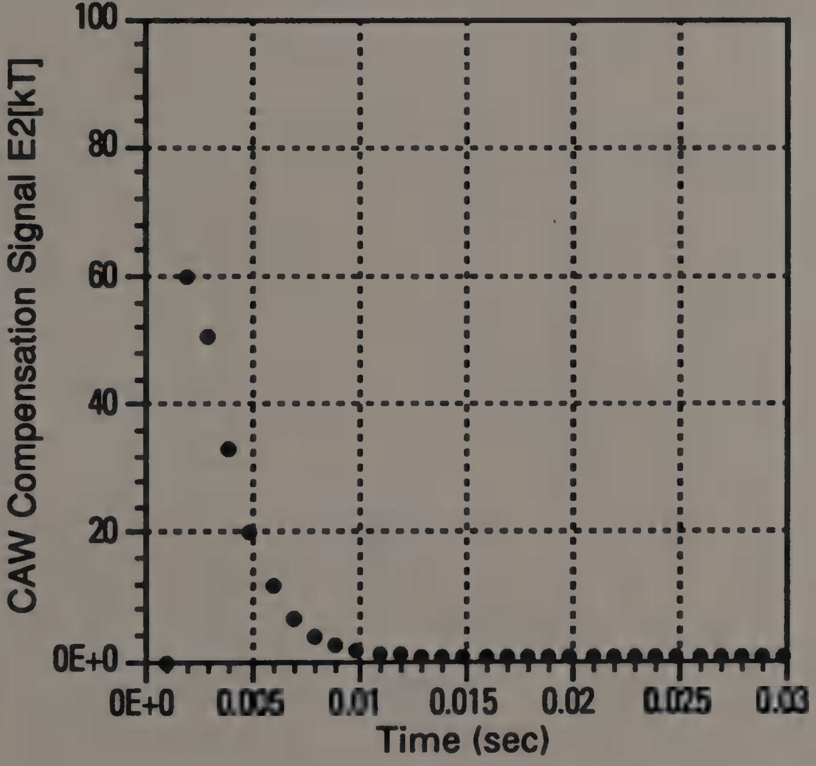


Figure 47. Step Response of Example System - CAW Signal e2[kT] Gain X = 20

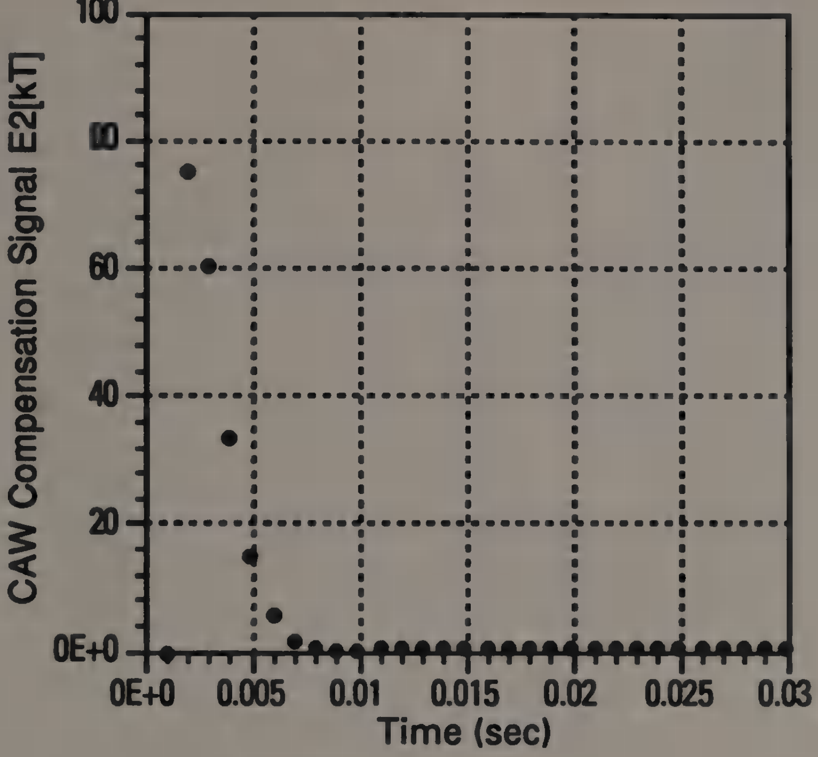


Figure 48. Step Response of Example System - CAW Signal e2[kT]
Gain X = 25

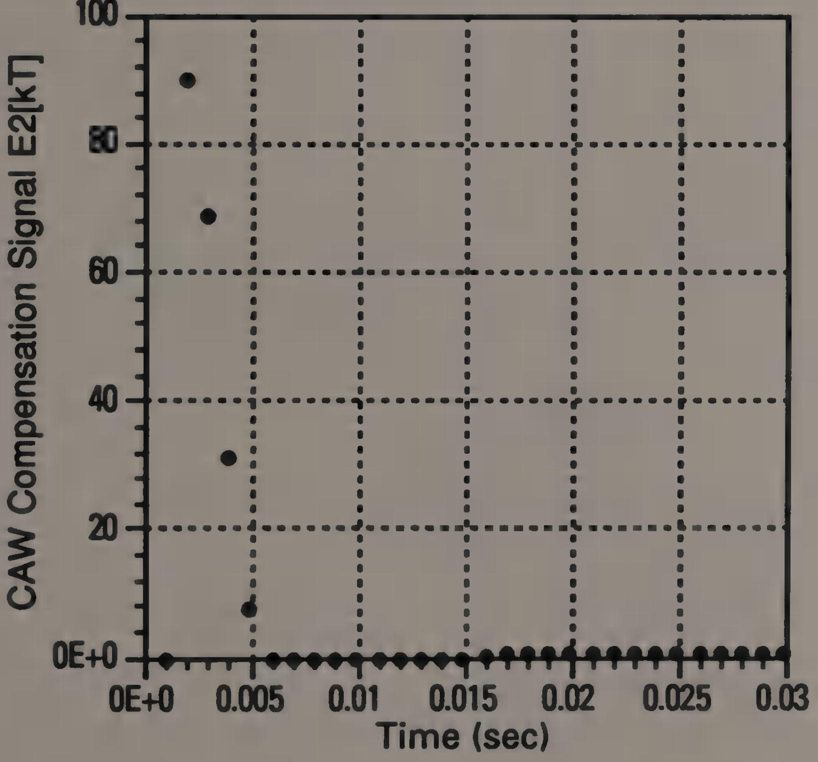
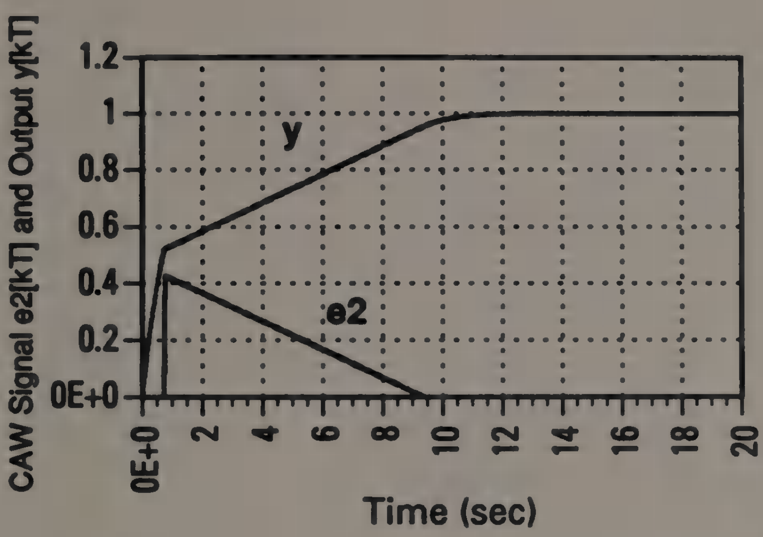


Figure 49. Step Response of Example System - CAW Signal e2[kT]
Gain X = 30

3.1.1.5 Avoiding Chatter by Bounding Sample Period and Feedback Gain

Chatter, and its associated problems, are avoided by designing the inner loop of the CAW system so that the overshoot of the control signal is zero or minimal. For discrete systems where the inner loop is of second order, this is achieved through a judicious selection of the feedback gain X and the sample time T. For a continuous system, this is achieved through selection of only the feedback gain X. Depending upon the design of the controller itself, the degree of overshoot causing chatter will vary. As a design criteria, however, it is suggested for systems where the inner loop is of second order (such as those discussed here) that X and T be chosen such that the damping ratio of the closed loop equation for the inner loop is between 0.9 and 1.0. This will keep the response of the inner loop fast; the overshoot small or zero; avoid chatter for all such controllers; and behave with performance similar to the continuous design. Adjusting the inner loop so that the denominator in the closed loop transfer function T_{ic} has a damping ratio of 0.9 or 1.0 is suggested as a design criteria. Using this criteria for the discrete CAW design of Figure 18 and a damping ratio of 0.9 results in an $X = 21$ for a $T = 0.01$. The step response of the design for these values of X and T is shown below in Figure 50.



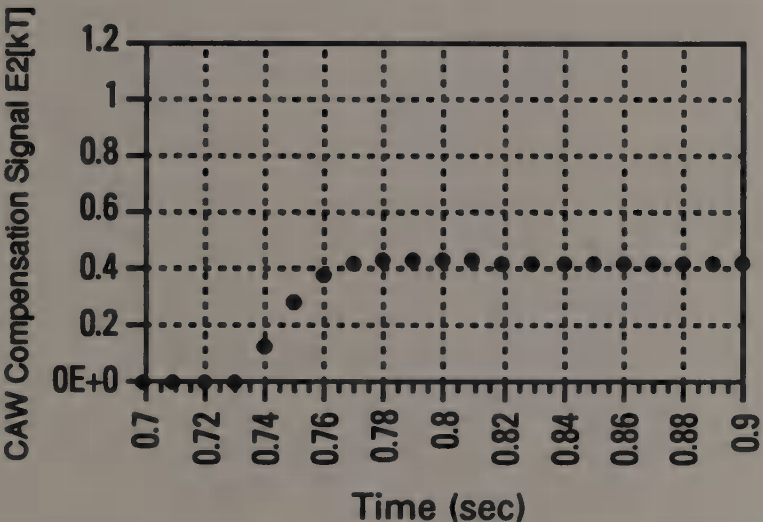


Figure 50. Step Response of Optimized Discrete CAW System

3.1.2 OS System

Glattfelder and Schaufelberger [11] describe an ARW system of the OS type. Their paper describes a continuous system without ARW and shows that it is susceptible to reset-windup caused by saturation within the plant. The OS modification to the system is described along with the improvement in system response. A brief review of these results will be described, followed by a presentation of a discrete design of the system and an analysis of the design criteria.

3.1.2.1 Continuous OS System

The continuous system presented by Glattfelder and Schaufelberger [11] is shown in Figure 51. For this system, the compensator, K(s), a proportional-integral controller and is given as

$$K(s) = K_p + \frac{K_i}{s} \tag{28}$$

where K_p is the proportional gain and is equal to 2 and K_i is the integral gain and is equal to 4. The plant P is given as

$$P(s) = \frac{1}{s} . \tag{29}$$

Note that the system depicted in Figure 51 appears somewhat different in the original paper since it been changed to be consistent with the nomenclature of this paper.

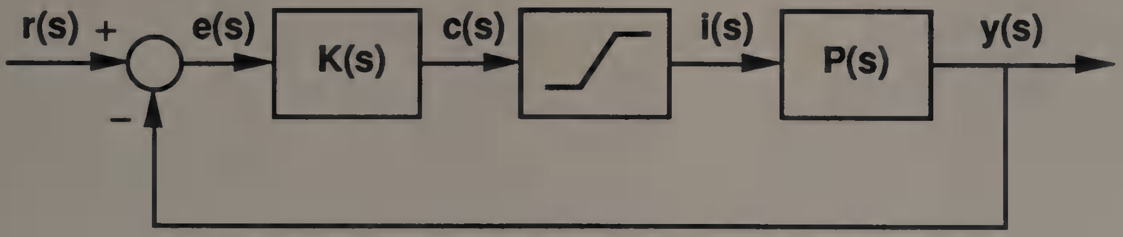


Figure 51. Continuous System by Glattfelder and Schaufelberger

If the saturation is temporarily removed from the system, the linear loop transfer function is

$$L_{ol} = P*K = \frac{K_p s + K_i}{s^2} . \tag{30}$$

The linear closed loop transfer function of this system with no saturation is

$$L_{cl} = \frac{P*K}{1 + P*K} = K_p * \frac{s + K_i/K_p}{s^2 + K_p s + K_i} . \tag{31}$$

The response of y and c to a unit step input at r with all states set to initial values of zero is shown in Figure 52.

Observe the response of the system with the saturation included and described by equation (1). This response is shown in Figure 53. A fourth order Runge-Kutta simulation was developed to model the results and is discussed in Appendix B. The large overshoot due to reset-windup is apparent. The overshoot in the system with saturation is over twice that of the system without saturation.

In order to eliminate the windup, improve system performance and reduce overshoot, Glattfelder and Schaufelberger propose an ARW configuration based on the OS configuration shown in Section 2.2.2. The configuration (Figure 54) switches in another signal to override the present error signal (which is input to the controller) when the plant input becomes saturated.

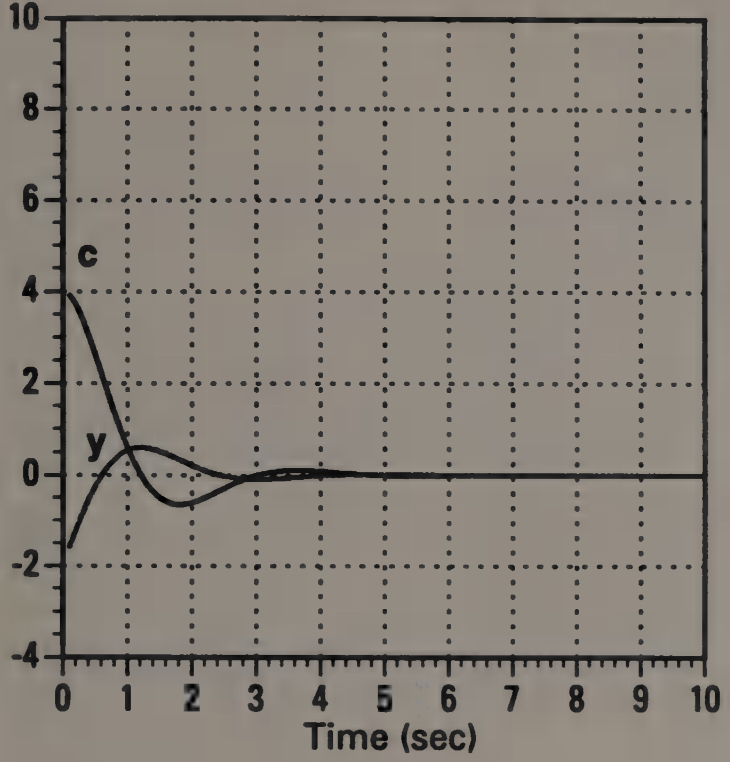


Figure 52. Linear Step Response of Glattfelder and Schaufelberger System

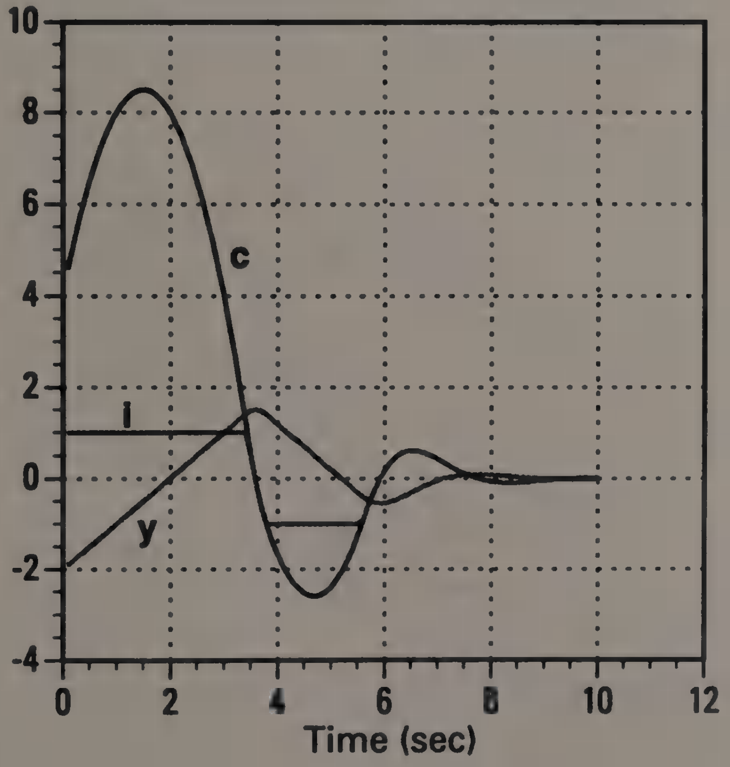


Figure 53. Step Response of Glattfelder System with Saturation

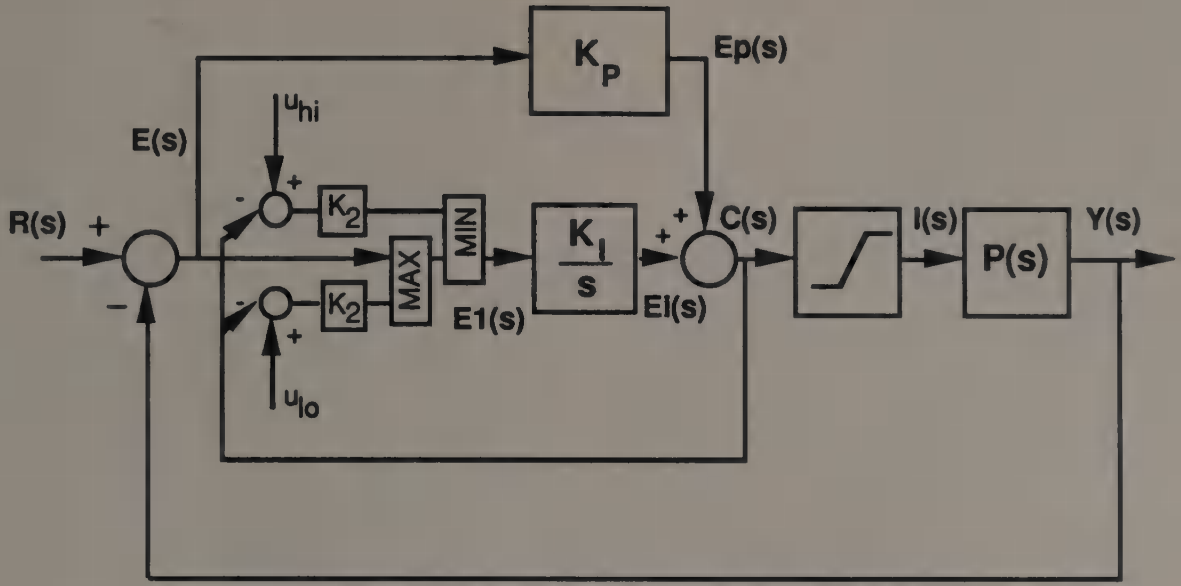


Figure 54. Glattfelder OS Configuration for Continuous System

In Figure 54, the compensator output c is compared to the upper (u_{hi}) and lower (u_{lo}) limit setpoints using a minimum and a maximum selector and high-gain amplifiers K_2 [7]. If c is driven towards either limit by the main error signal e, the corresponding high-gain feedback is selected ("overriding e ") and adjusts $e_i(t)$ continuously in such a way that $c(t)$ never saturates. This is achieved by a proper selection of u_{lo}, u_{hi}, and K_2. The compensator will, therefore, be assumed linear because it is used only in its linear range [7]. Note that, in this implementation, Glattfelder and Schaufelberger choose to operate the ARW on the integrator portion of the controller only. This is similar to implementation 9c discussed in Section 2.2.1.

The response of the continuous system with the OS modification (Figure 55) shows the effectiveness of this configuration (the Runge-Kutta simulation which produced these results is discussed in Appendix B).

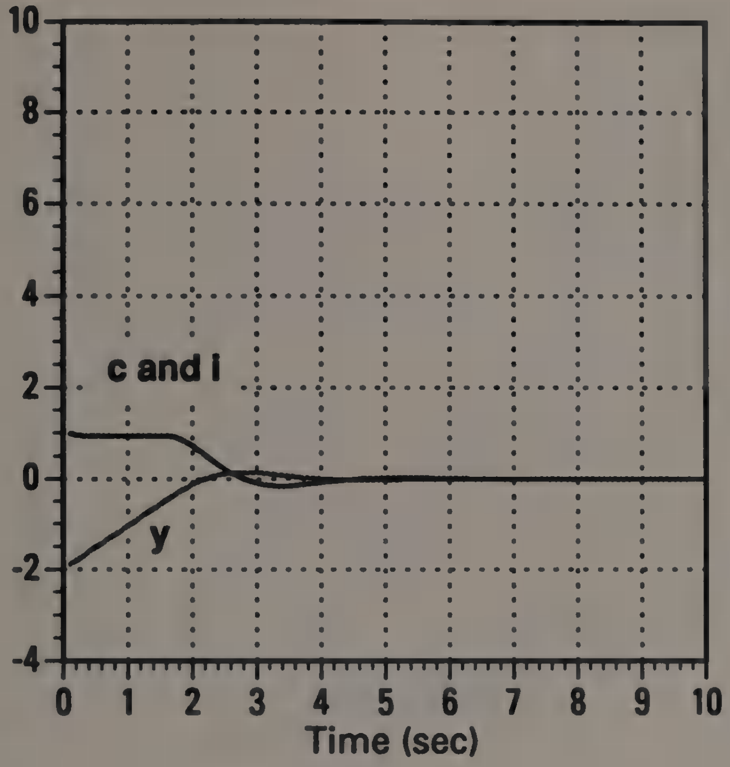


Figure 55. Step Response of Glattfelder OS System with Saturation

The performance improvement resulting from the ARW modification is obvious: the ARW reduces the overshoot by 91% and decreases the overall settling time by about 50%.

3.1.2.2 OS System with Doyle Plant and Compensator

In order to compare the performance of the Doyle and Smith CAW to the Glattfelder and Schaufelberger OS implementations, assume that the compensator and plant used to analyze the CAW system is introduced into the OS System. Assume that for Figure 54, the controller K(s) is

$$K(s) = \frac{2}{s + 0.1} \tag{32}$$

and the plant P(s) is

$$P(s) = \frac{s + 0.1}{2s} \; . \tag{33}$$

The nonlinear response of the system with saturation would be identical to that in Figure 14 showing the classical overshoot problem associated with the windup. The OS implementation is added to the system (Figure 56) to solve the reset-windup problem.

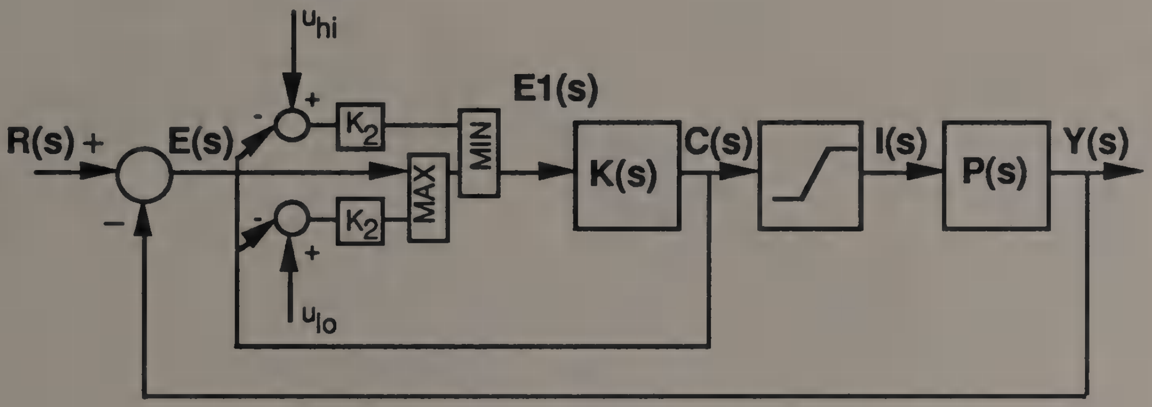


Figure 56. Continuous OS System #2

Inclusion of the OS compensation to the system eliminates the effect of the reset-windup as demonstrated by the simulation results to a step response (Figure 57).

Note that the response of the system using the OS compensation is very similar to that of the system using the CAW compensation. This leads to the conclusion that the OS approach and the CAW approach, while appearing to be quite different, operate in a similar fashion. This conclusion is true and will be investigated later in more detail.

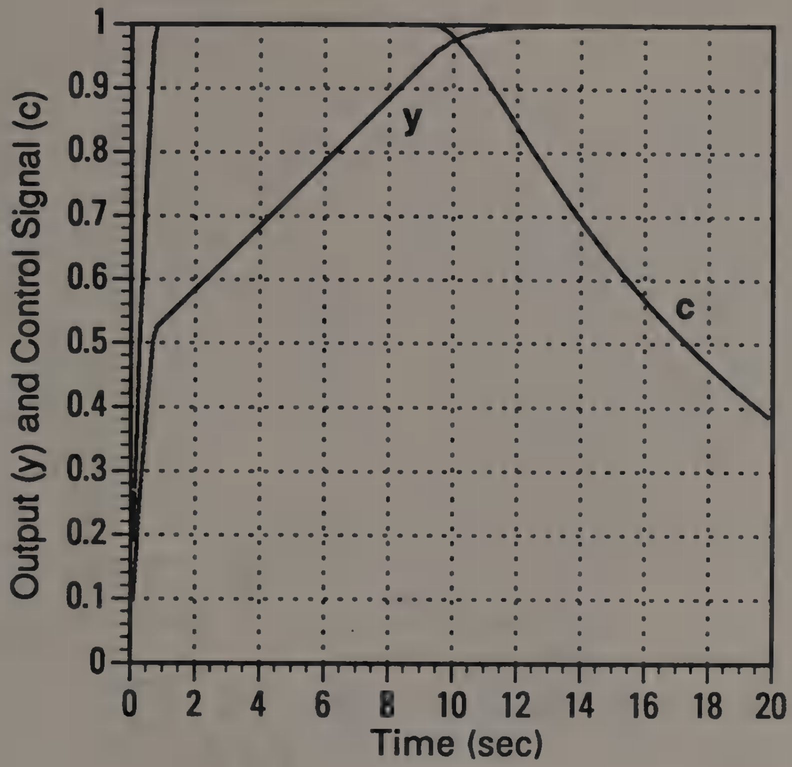


Figure 57. Step Response of Continuous OS System #2

3.1.2.3 Design of a Discrete OS System

Design by discrete equivalent has been used here, as with the CAW System, to develop a discrete design for the Glattfelder and Schaufelberger OS system. The bilinear transform was used to design the discrete controller, and the hold-equivalence transform was used to design a discrete model of the plant. This results in a $K_D(z)$ and $P_D(z)$ identical to equations (13) and (17):

$$K_D(z) = K_A * \frac{1 + z^{-1}}{1 + az^{-1}} \tag{34}$$

where K_A and a are described by equations (14) and (15), (with K_A replacing K_1) and

$$P_D(z) = K_B * \frac{1 + bz^{-1}}{1 - z^{-1}} \tag{35}$$

where K_B and b are described by equations (18) and (19) respectively (with K_B replacing K_2). The discrete design of the system is shown in Figure 58.

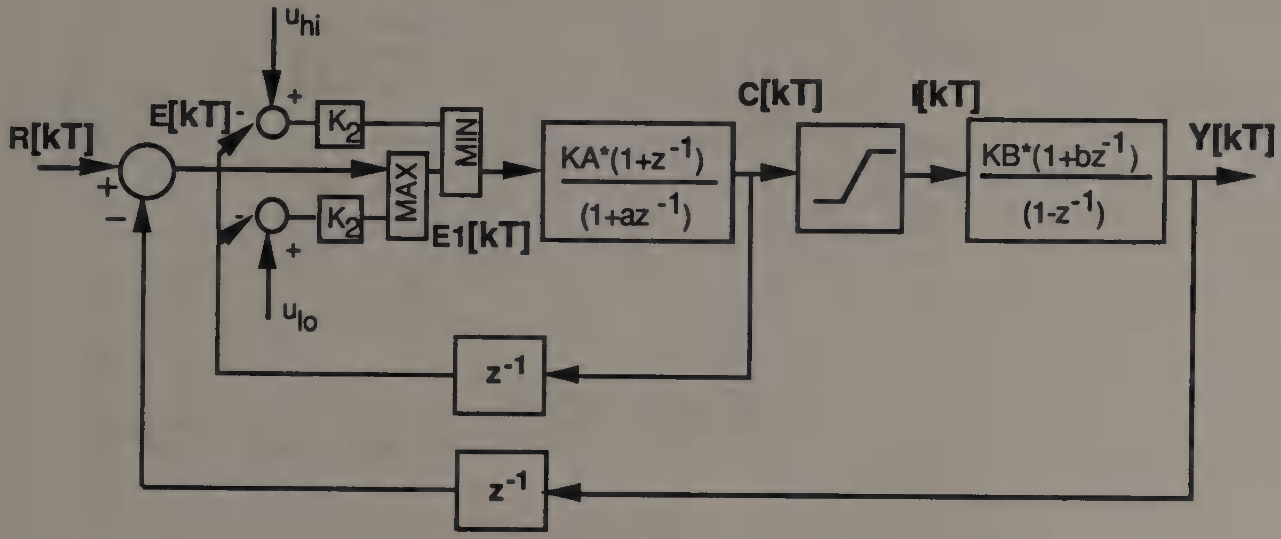


Figure 58. Discrete Design of OS System

Again note that a discrete delay of one (z^{-1}) is necessary in each feedback because the controller cannot have prior knowledge of the output states before they are computed.

To analyze the performance of the discrete model and compare it to the original continuous system, a discrete simulation of this design was developed (Appendix B). Responses to step inputs were used as a measure of the accuracy of the design and were used to determine system behavior as various characteristics of the system were varied.

3.1.2.4. Comparison of Continuous and Discrete OS Designs

System performance was analyzed using the simulation developed. If the design of the plant and controller is unchanged, the only parameter within the continuous system that can be varied is the override signal gain K_2. Both the gain K_2 and the sample time T can be varied within the discrete system. A study of system step response versus K_2 and T was conducted on the discrete simulation to discover whether chatter could be achieved in the OS system as well. It was found that chatter could also be present in the discrete OS system as shown in Figures 59-61 but, as before, not in the continuous design. It was also discovered that the chatter occurred at values of K_2 which were close to those for X in the CAW system for the same value of T. Analyzing the OS design proved why this should be so.

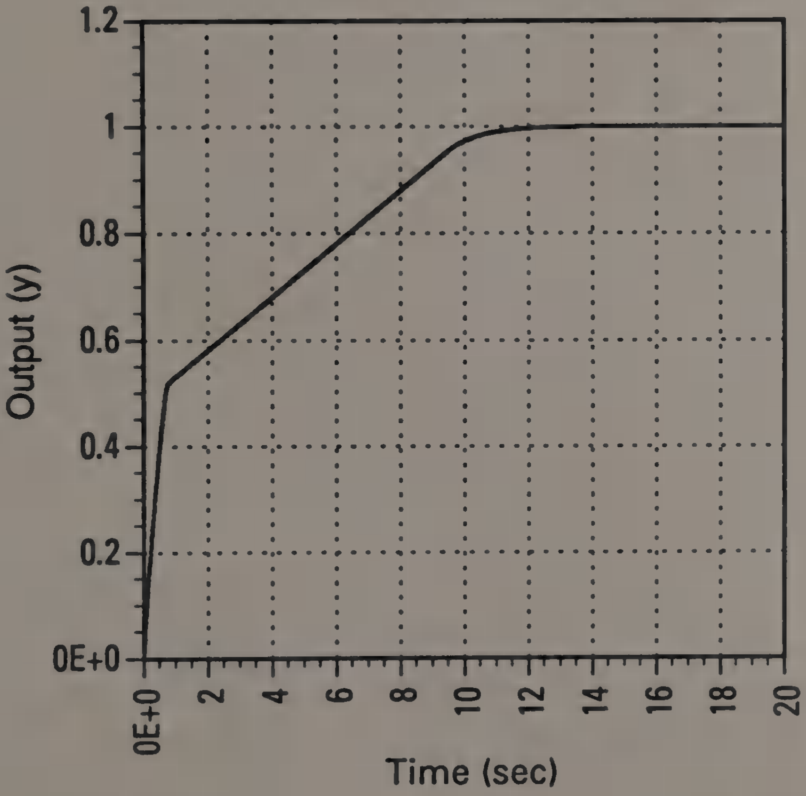


Figure 59. Step Response of Discrete OS System (X=10, T=0.01)

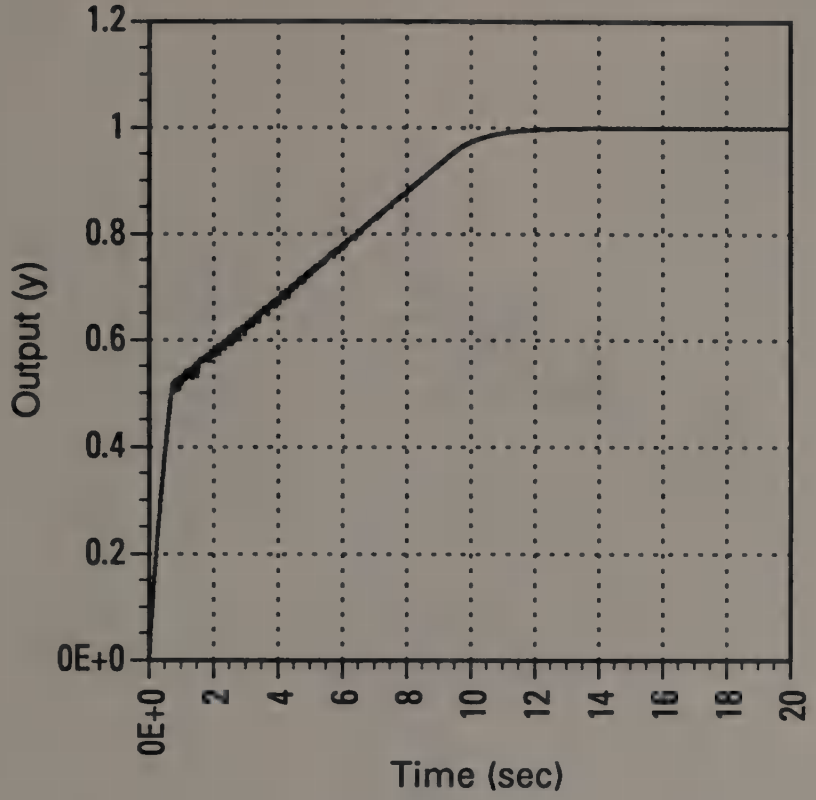


Figure 60. Step Response of Discrete OS System (X=500, T=0.01)

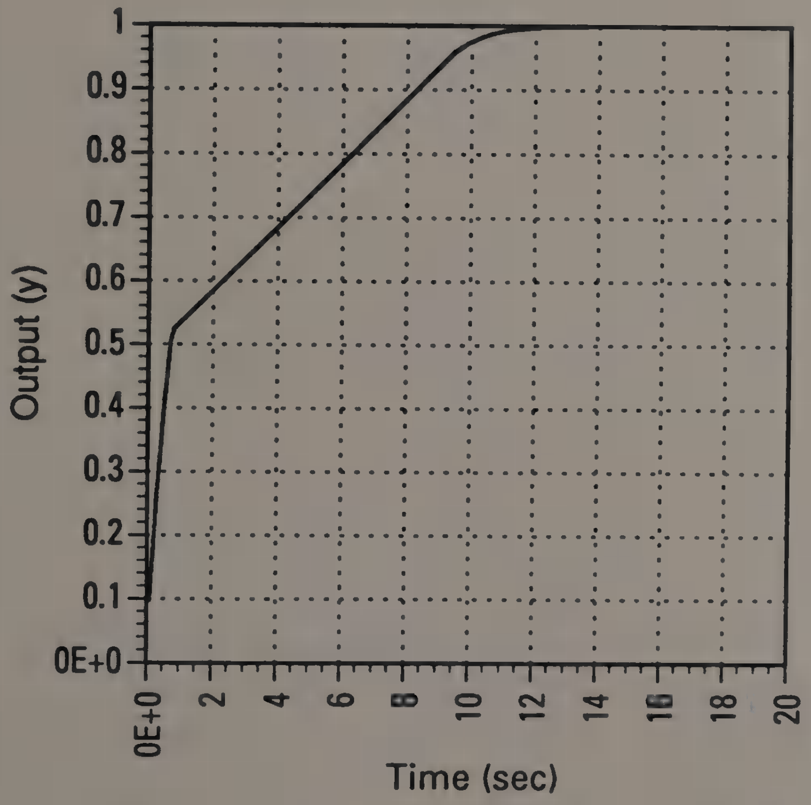


Figure 61. Step Response of Continuous CAW System (X=500, T=0.01)

In the OS design it was found, as with the CAW system, that the chatter appeared when the value of K_2 exceeded a certain threshold (with a fixed T) and the amplitude of the chatter increased linearly with K_2 above that threshold. Given the same plant and compensator as the CAW example, the chatter was found to appear at a value of K_2 which was equal to that for X (given the same T). Given a sample time T = 0.01 seconds, for example, the chatter began to appear in the system at K_2 of approximately 100-110. The next section describes the analysis done in order to determine the cause of the chatter in the OS system and how to quantify the chatter threshold in terms of K_2 and T.

3.1.2.5 Avoiding Chatter by Bounding Sample Period and Feedback Gain

Close examination of the OS system led to the conclusion that the operation of the OS compensation is, in a certain sense, similar to that of the CAW even though the implementation appears quite different. The OS system operates on the principle of an inner loop which modifies the error signal input to the compensator when the output of the compensator is above the saturation values of the saturation block. The CAW differs from the OS in that its inner compensation signal is added (negatively) to the original error signal in order to reduce the input error signal to the controller. In the OS system, the inner compensation signal is switched in and the original error signal is switched out such that the inner compensation signal becomes the new error signal to the controller.

It was suspected that the operation of the inner loop was the cause of the chatter in the OS system just as in the inner loop of the CAW. Because the Min and Max selectors of the OS system are nonlinear and difficult to analyze, the inner loop of the OS system is difficult to analyze as well. Glattfelder and Schaufelberger [11], however, provide a means of replacing the Min and Max selectors with an equivalent nonlinear representation which makes the system easier to visualize and analyze. The Min-Max selector pair can be replaced by a nonlinear equivalent shown in Figure 62.

Placing the equivalent representation into the OS system of Figure 58 gives an equivalent representation of the system (Figure 63) which will allow easier analysis of the operation of the inner loop.

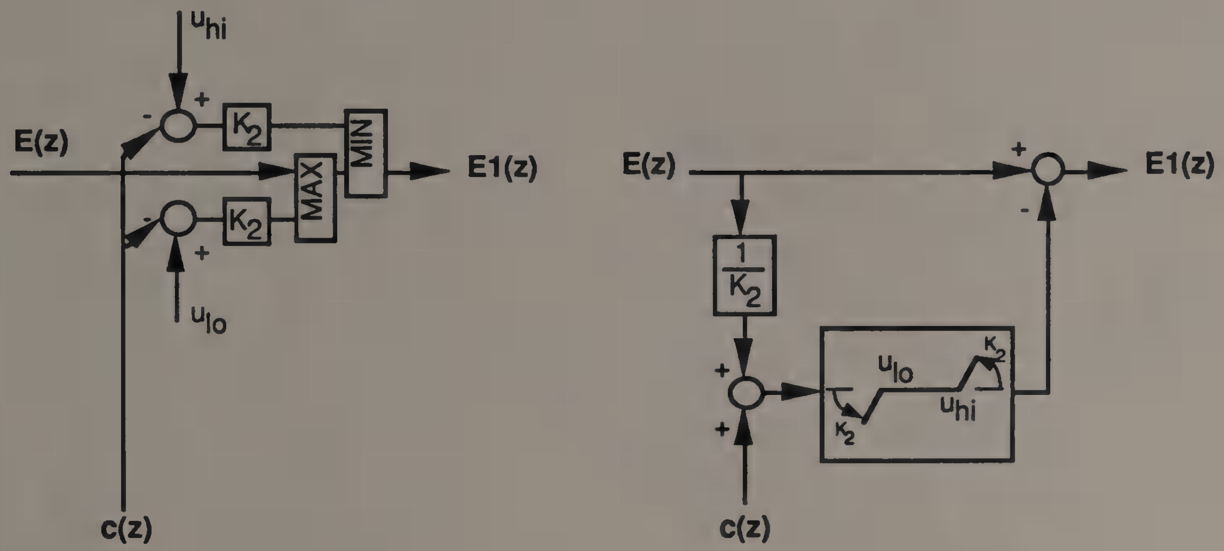


Figure 62. Equivalent Representations of the Nonlinear Feedback

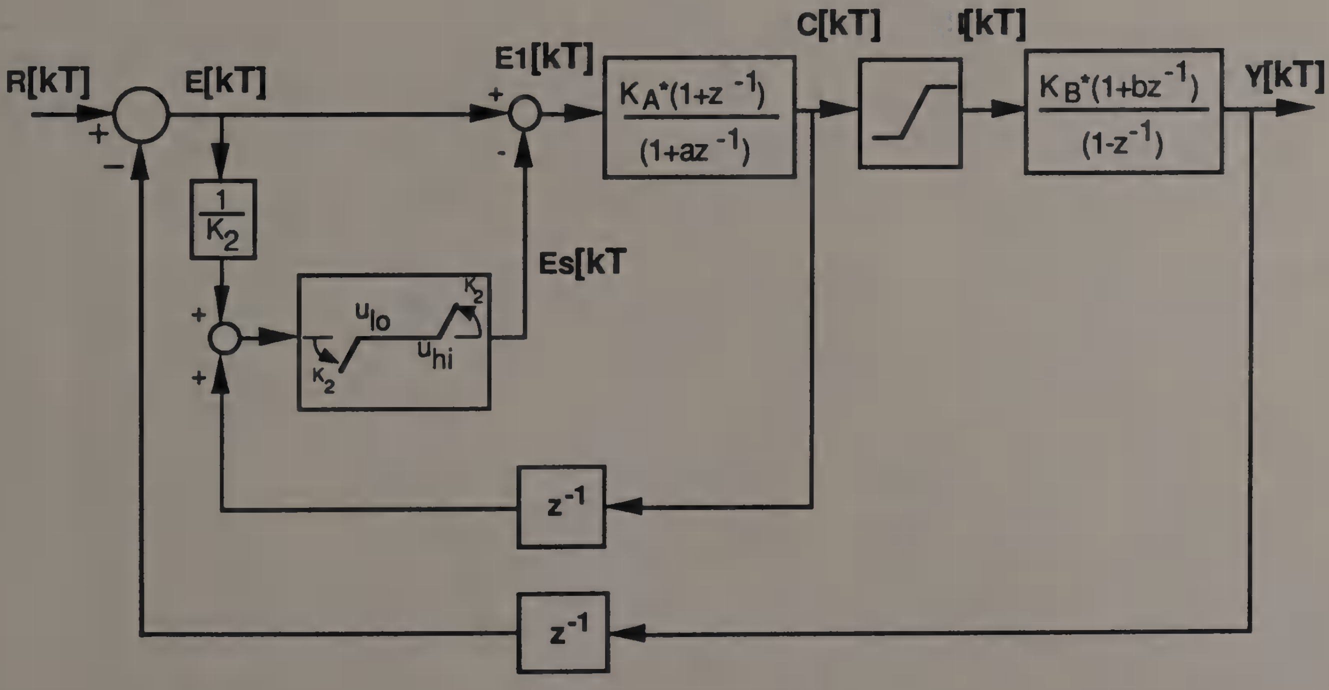


Figure 63. Equivalent Discrete OS Design

When the control signal c is not above the saturation values and the saturation block is not in saturation, the override signal e_s from the OS compensation is zero. When the control signal drives the saturation block into negative or positive saturation, however, the nonlinear block from the equivalence will be in the linear region and will be equal to a constant value, K_2. Also, the contribution from the path e through $1/K_2$ through the nonlinear equivalence block will be a constant. The inner loop of the OS system, therefore, consists only of the path from c through discrete delay, through the gain K_2 and through the controller $K_D(z)$. The inner loop can therefore be reduced to the representation shown in Figure 64.

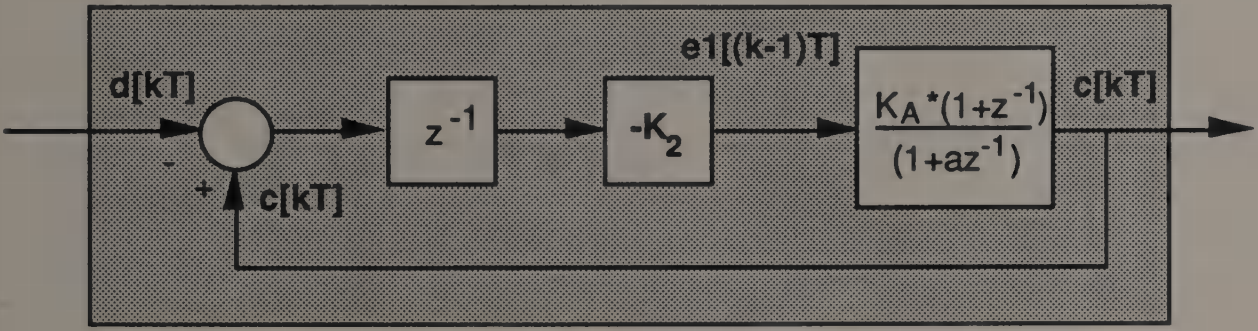


Figure 64. Inner Loop of Discrete OS System

The signal $d(k)$, and the summing junction at $c(k)$ and $d(k)$, has been added to the loop so that the effect of step responses into the inner loop can be analyzed. As in the CAW system, when the OS compensation is switched into the system, the effect is that of a step function into the inner loop at the point c. To link the chatter within the system to the behavior of the inner loop, the step response of the inner loop shown in Figure 65 was analyzed for values of K_2 and T. The transfer function from d to c (T_{dc}) is

$$T_{dc}(z) = -\frac{K_A*K_2*z^{-1}(1+z^{-1})}{1 + z^{-1}(a+K_A*K_2) + z^{-2}(K_A*K_2)} \quad . \qquad (36)$$

This is exactly the same as the transfer function of the inner loop for the discrete CAW system with K_2 replacing X and with the exception of the negative sign. The step response of the inner loop to values of K_2 for T=0.01 are shown in Figure 65-69. This exactly matches Figures 28-32 with the exception of the polarity. Therefore it was determined that the operation of the OS compensation and the CAW is the same.

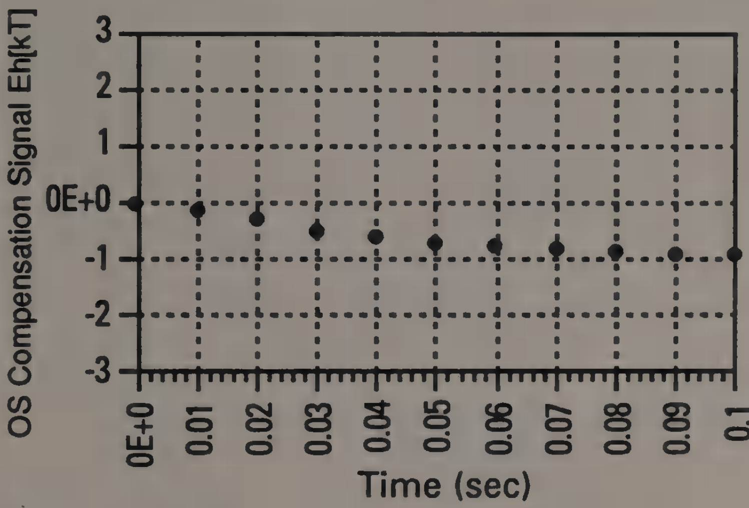


Figure 65. Step Response of Discrete OS Inner Loop (X=10, T=0.01)

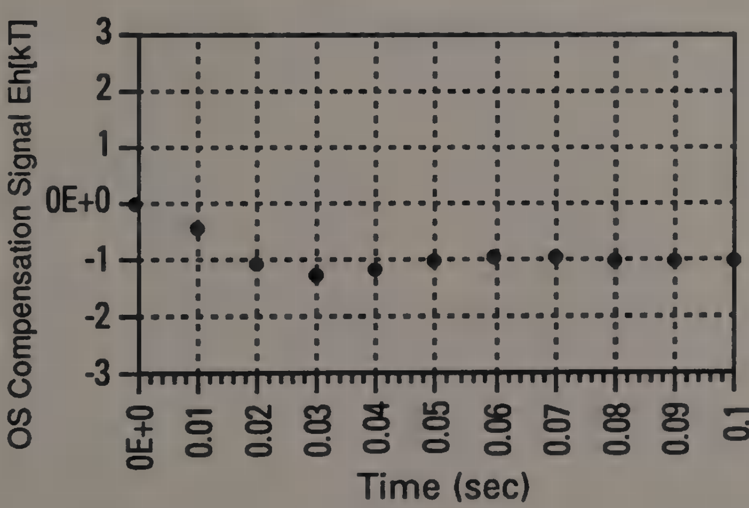


Figure 66. Step Response of Discrete OS Inner Loop (X=40, T=0.01)

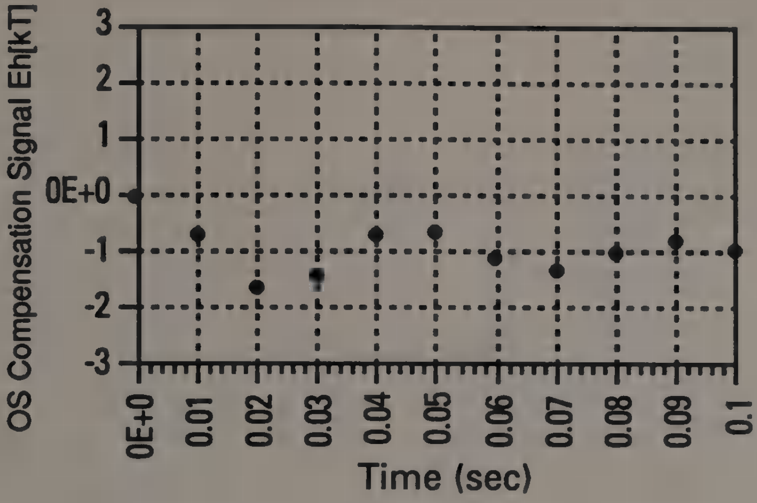


Figure 67. Step Response of Discrete OS Inner Loop (X=70, T=0.01)

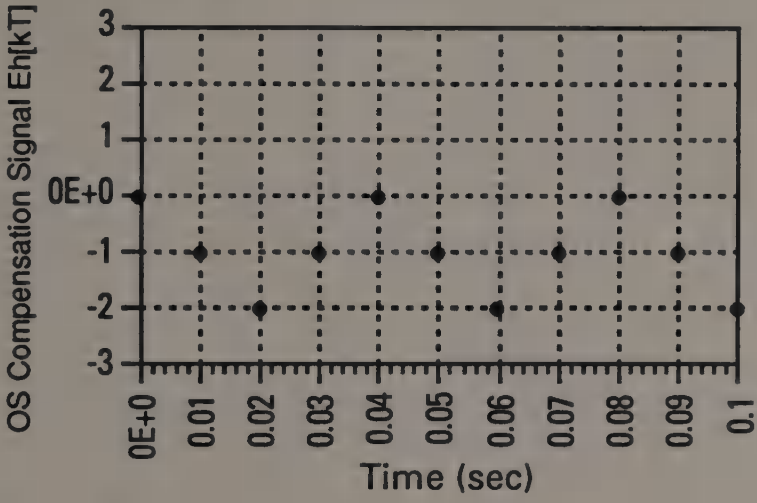


Figure 68. Step Response of Discrete OS Inner Loop (X=100, T=0.01)

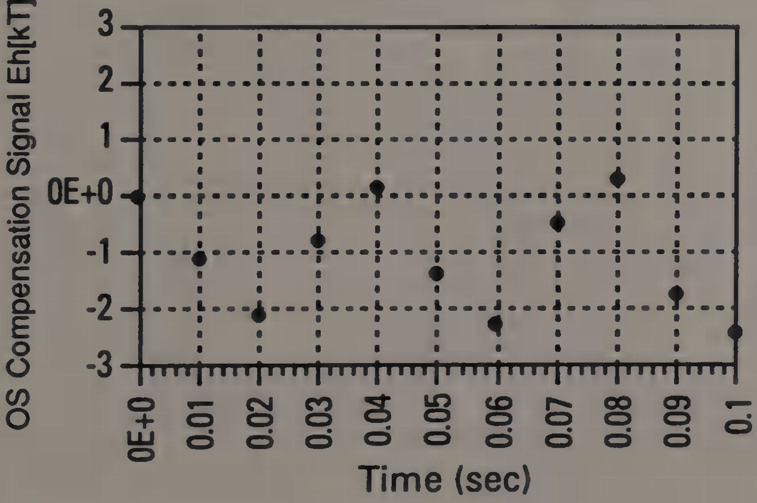


Figure 69. Step Response of Discrete OS Inner Loop (X=110, T=0.01)

The cause of the chatter in the OS system is similar to that in the CAW system. If *eh* becomes less than *e* , the OS compensation will then be shut off and the error signal *e1*, which was equal to *eh*, will be replaced with *e*. This should happen normally when the plant's response causes *e* to reduce below the value of *eh* . The signal *eh* responds in the same way as that predicted by the output of the inner loop in Figure 64.

If the overshoot is too much however, as it is if the inner loop becomes unstable, then *eh* will overshoot below *e* and the inner loop will shut off momentarily. The system dynamics will then realize that the trajectory of the control signal is in incorrect for the desired response, and on the next sample time, will begin driving back toward saturation. When saturation is reached, the process begins again.

It is therefore apparent that the behavior of the OS design is identical to that of the CAW design. Its performance with other controllers such as the PI controller used for the second CAW example will also be the same as for the CAW design. It is recommended, therefore, that values of T and K_2 are chosen such that the overshoot is kept at a minimum while the response of the inner loop is kept at least ten times faster than the outer loop.

The recommended design criteria for a second order inner loop is to adjust the inner loop for a damping ratio of 0.9 to 1.0. Using this criteria for the discrete OS design of Figure 58 gives a K_2 of 21 for a $T = 0.01$. The step response of the design for these values of T and K_2 is shown in Figure 70 .

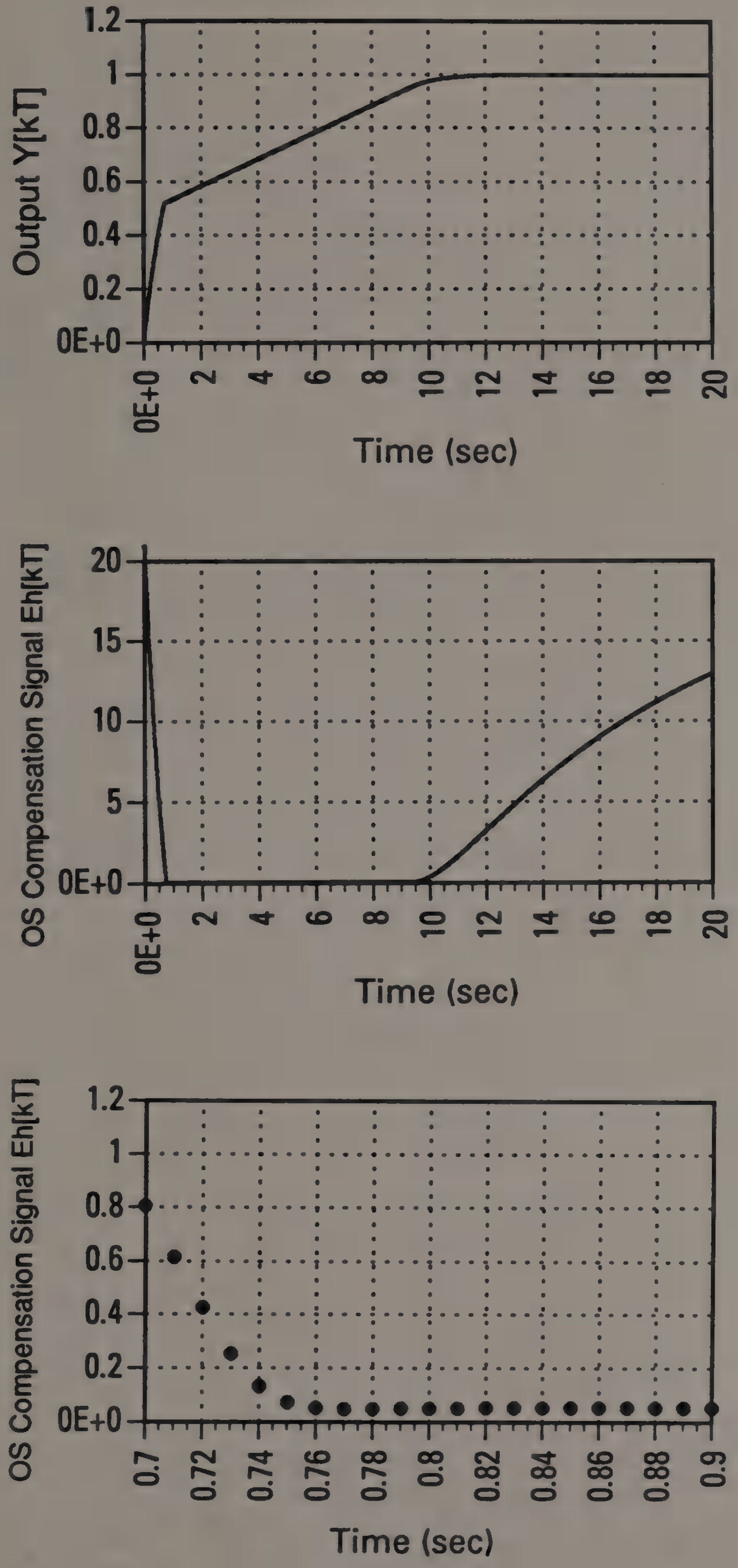


Figure 70. Step Response of Optimized Discrete OS System

3.2 Application of ARW to Track Radar Control System

3.2.1 Existing Continuous Track Radar Control System

This section discusses the application of a CAW to a practical application. Figure 71 depicts an existing track radar antenna system used for tracking aircraft.

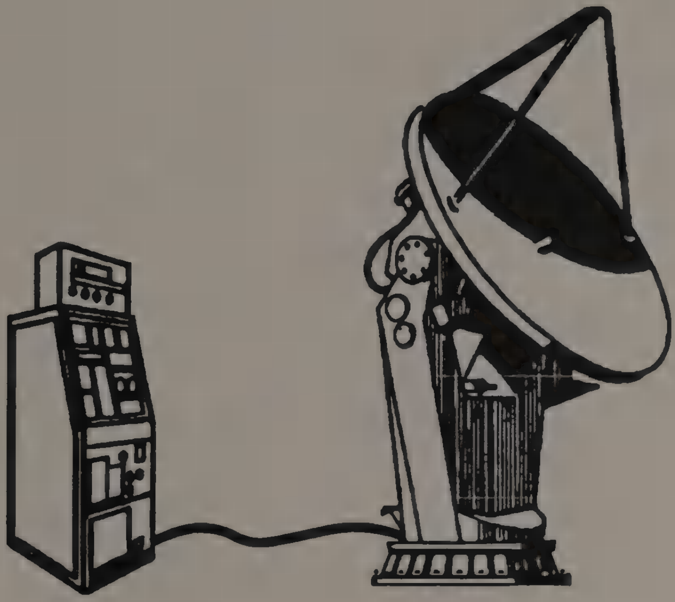


Figure 71. Aircraft Track Radar Antenna System

This system's tracking rate and position are controlled by two rate servos. The desired rate, $\omega_{p,}$ is commanded by a control computer and the closed-loop servo commands the motor to slew the platform at the desired rate. The antenna platform is gimballed in two-axes: elevation and traverse, with one control loop for each axis. A block diagram of the servo control system is shown in Figure 72. Both axes have a similar design with the difference being the actual values within the blocks. The discussion in this section will be limited to only the traverse axis since the design process and the results will be similar for either axis; thus, the transfer functions shown are for the traverse axis. The transfer functions for each block are:

$$K_{CC} = 5.11 \qquad \text{volts/(rad/sec)}, \tag{37}$$

$$G_I(s) = \frac{10.50 * \left\{1 + \frac{s}{10.74}\right\}}{s} \qquad \text{volts/volt}, \tag{38}$$

$$G_C(s) = \frac{34.01}{\left\{1 + \frac{s}{1024.59}\right\}} \qquad \text{volts/volt}, \tag{39}$$

$$G_M(s) = \frac{0.28}{\left\{1 + \frac{s}{1.18}\right\} * \left\{1 + \frac{s}{98.91}\right\}} \qquad \text{(rad/sec)/amp}, \tag{40}$$

$G_G(s) =$

$$\frac{6.406611384e12}{\left\{s^2+766.41s+681377.11\right\} * \left\{s^2+2668.20s+3046393.90\right\}}$$

$$\text{volts/(rad/sec)}, \tag{41}$$

$$G_F(s) = \frac{1.66}{\left\{1 + \frac{s}{148.27}\right\}} \qquad \text{volts/volt}, \tag{42}$$

$$G_{PA}(s) = 1 \qquad \text{amps/volt}. \tag{43}$$

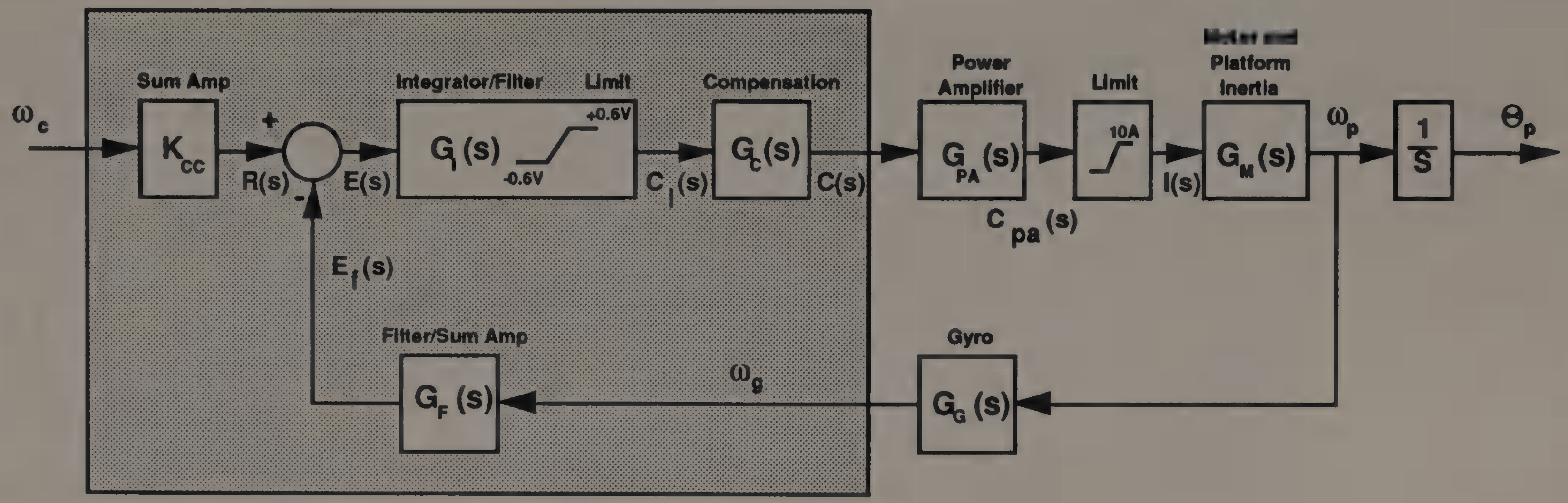


Figure 72. Track Radar Rate Servo Control System Block Diagram Existing Analog Design

The commanded rate, ω_p, is compared with the actual platform rate measured by a rate gyro in the feedback loop. The feedback gyro measures the actual platform rate, including platform disturbances such as base motion, and generates a voltage proportional to the instantaneous platform rate. This rate is filtered and summed in with the amplified command to generate an error signal at the summing junction which is input to the integrator/filter and then to the compensator which generates the current command to the power amplifier. The power amplifier converts the commanded control signal, a voltage, to a current command and that command drives the motor. The current command to the motor generates a torque on the motor shaft which then moves the platform at the commanded rate. The motor is connected directly to the platform, without gearing, and the transfer function for the motor contains the inertia of the antenna platform.

All of the blocks located in the shaded area contribute to the control signal that controls the plant. The existing servo system implements analog compensation elements. These elements are mechanized with analog op-amps and discrete components. It is desired to replace the existing analog control system with a digital one in order to reduce the production cost of the system and to increase the system reliability and maintainability. The elements shown within the shaded area, therefore, will be replaced with software algorithms running in a computer. In order to accomplish this, it is necessary to develop discrete versions of the elements which can be implemented in the computer.

The analog design in Figure 72 contains two nonlinear elements of importance. The output of the power amplifier for the motor is limited to 10 amperes in order to protect the motor from being damaged by a current command that would exceed the current carrying ability of the motor's primary windings (20 amperes). Figure 73 shows the linear step response of the system (no saturation in the power amplifier); the output of the integrator/filter is shown in Figure 74; and the output of the power amplifier is shown in Figure 75.

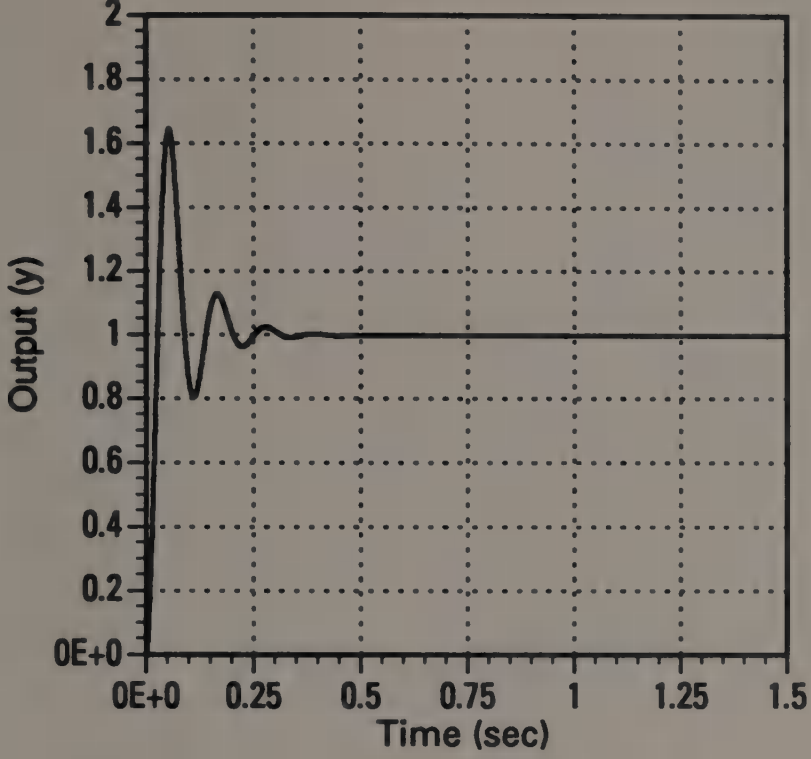


Figure 73. Linear Step Response Output (y) - Track Rate System

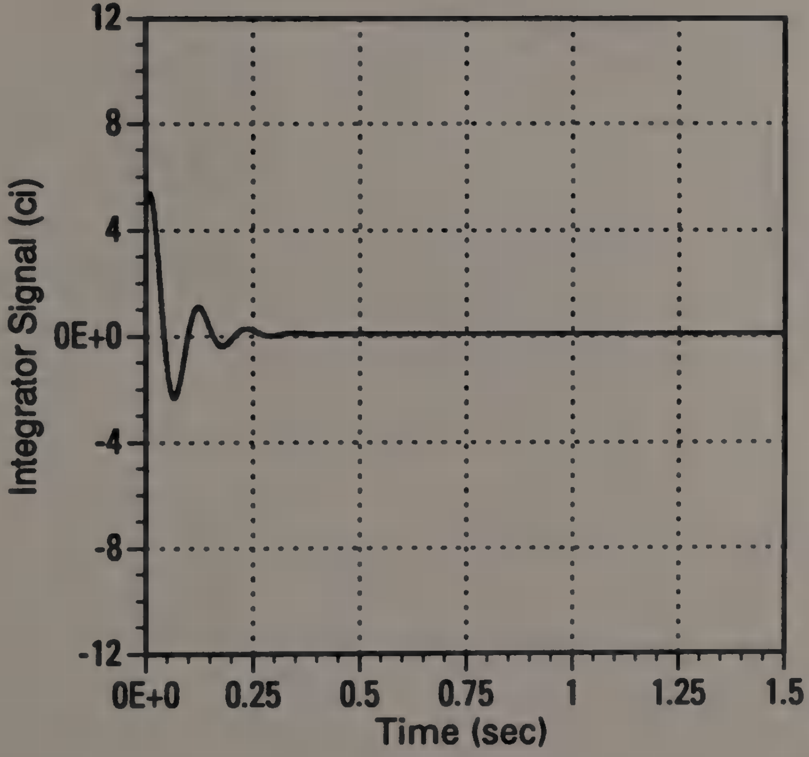


Figure 74. Linear Step Response Control (c_i) - Track Rate System

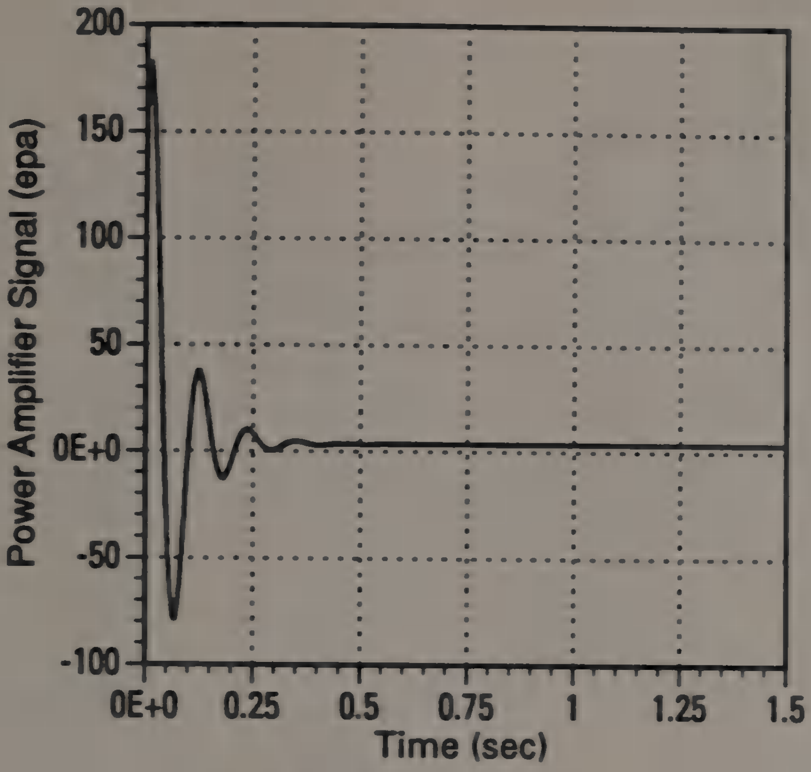


Figure 75. Step Response Power Amplifier (c_{PA}) - Linear System

Figure 75 illustrates that, without the limiting in the power amplifier, the current commands would far exceed the motor's limits. This is prevented by the saturation in the power amplifier. However, a problem introduced by this protection can be seen in Figures 76 and 77. The step response of the system with saturation in the power amplifier causes the system to exhibit classic reset-windup. The control signal clearly shows the classic windup phenomenon with the resulting large overshoot (66.1%) and long settling time (1.0 seconds) in the output.

To eliminate the problem of reset-windup in the analog system, the original design includes a clamp on the integrator/filter. This clamp limits the output of the filter to 0.6 Volts. The clamp is mechanized with diodes across the filter elements within this block and will be discussed in more detail in Section 3.2.3. To show the affect of this ARW compensation in the original system, observe the step response of the system with this clamp included. Figures 78 and 79 illustrate the step response of the system with the existing windup compensation.

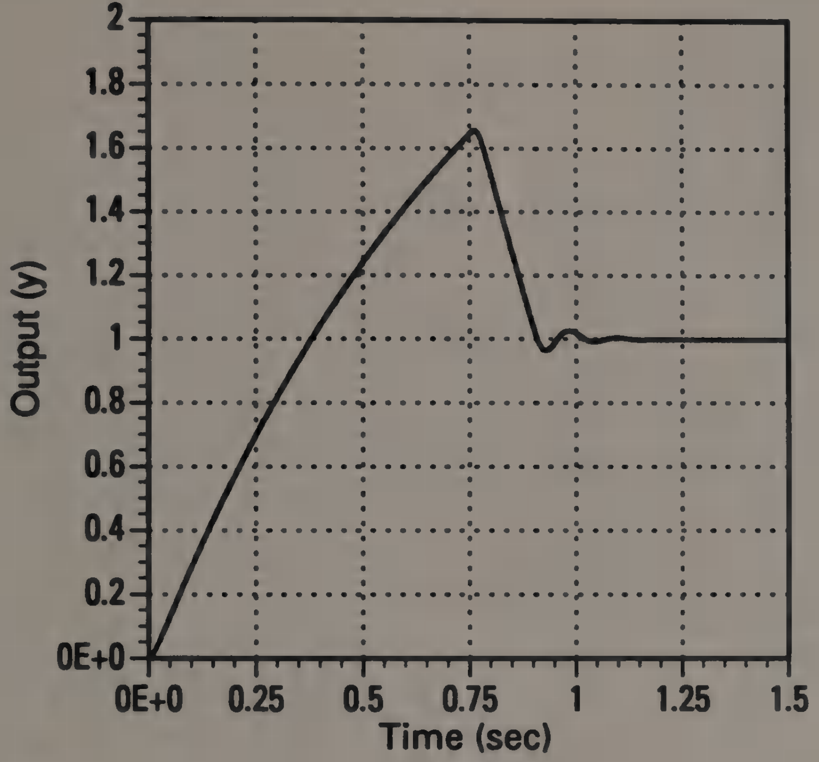


Figure 76. Step Response Output (y) - System with Saturation in Power Amp

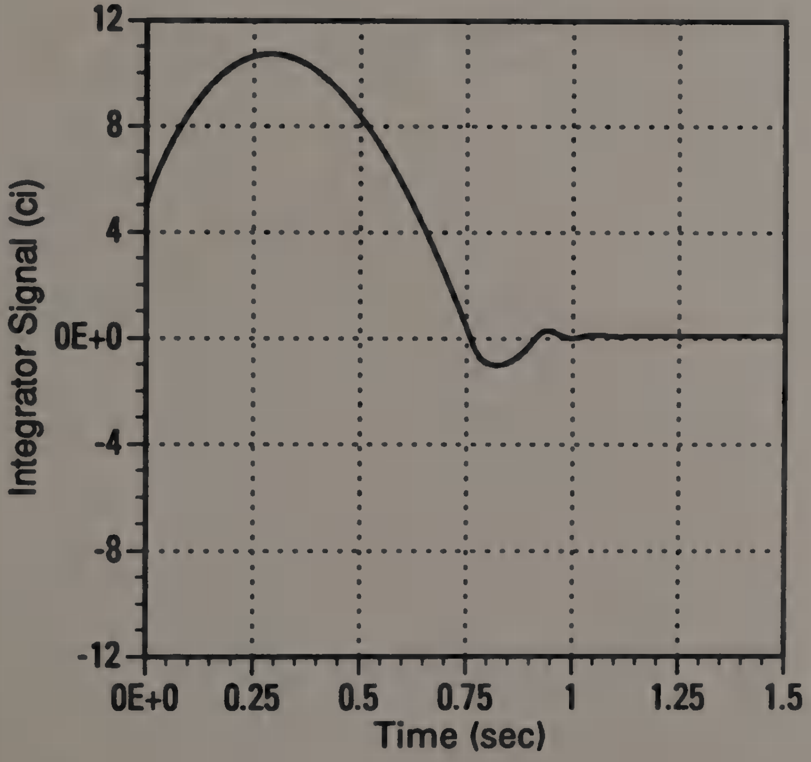


Figure 77. Step Response Control (c_i) - System with Saturation in Power Amp

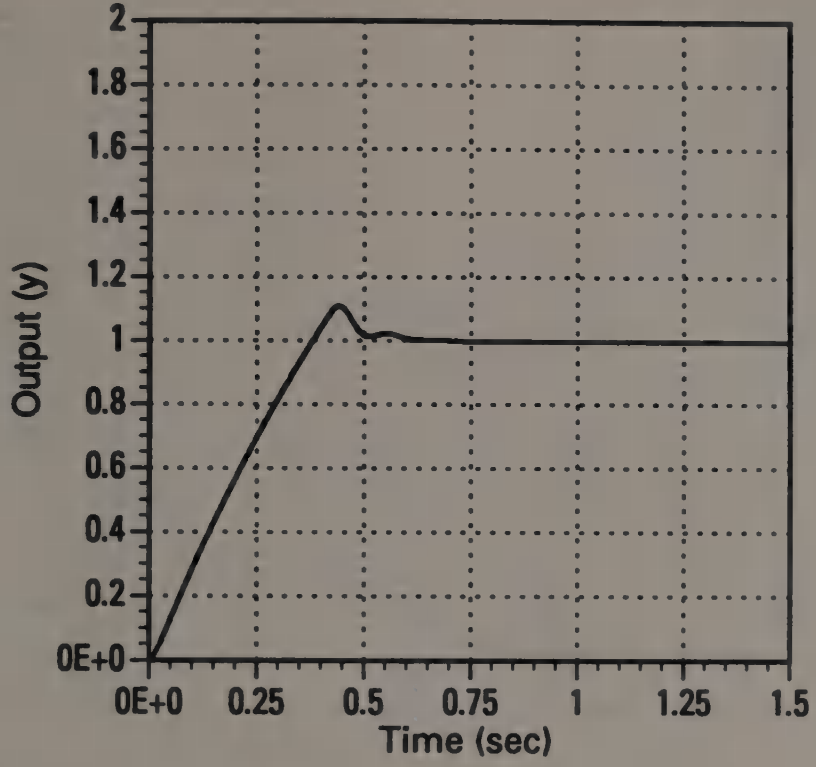


Figure 78. Step Response Output (y) - System with ARW in Integrator/Filter

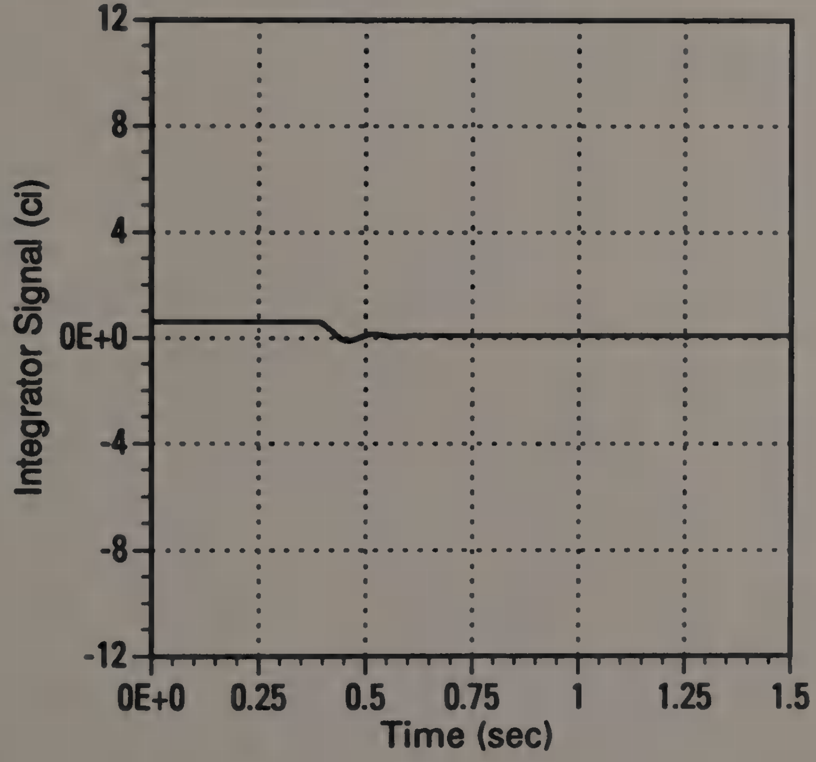


Figure 79. Step Response Control (c_i) - System with ARW in Integrator/Filter

3.2.2 Discrete Design of Track Radar Control System

As previously stated, it is desired to design a digital control system to replace the existing analog system shown in Figure 72. The approach chosen for the design was to replace the integrator/filter G_I, the compensator G_C, and the filter G_F with discrete designs developed using the method of design by discrete equivalent. The blocks within the shaded region are considered the controller for the digital system and each block was replaced with an equivalent discrete design developed using the bilinear transform as discussed in Appendix B.

Discrete models of the plant (motor G_M), and the feedback sensor (gyro G_G), were designed in order to model the discrete design, determine the system performance, and compare it to the performance of the existing system. These designs utilized the step invariant transform (also discussed in Appendix B). The discrete design for the track rate servo control system is shown in Figure 80.

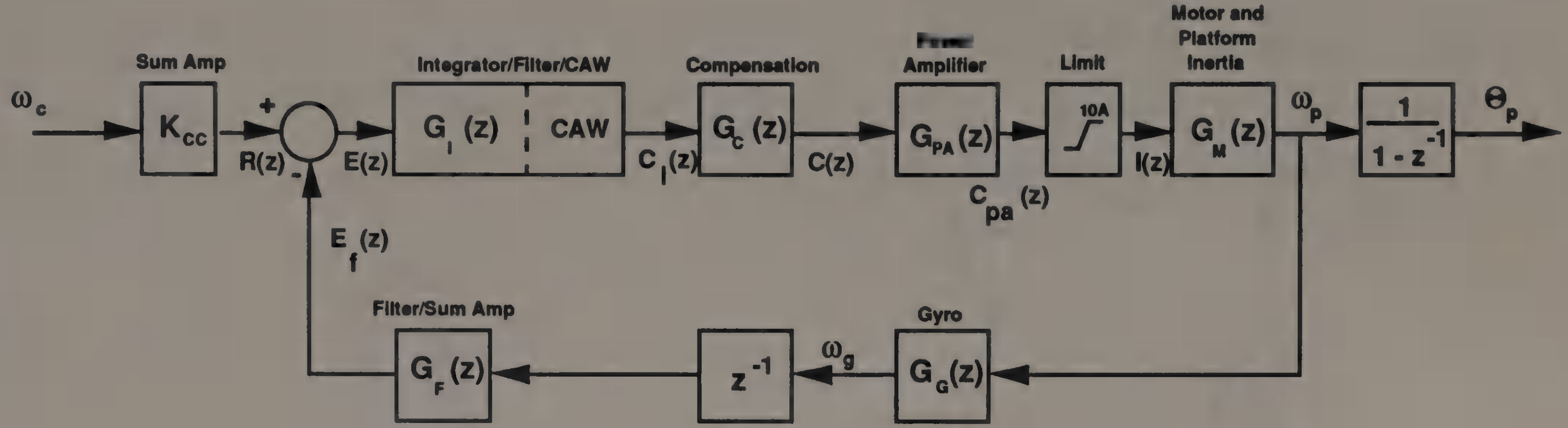


Figure 80. Track Radar Rate Servo Control System Block Diagram Discrete Design

The transfer functions for each of the blocks are:

$$K_{CC} = 5.11 \qquad \text{volts/(rad/sec)}, \qquad (44)$$

$$G_I(z) = \frac{K_I * \left\{1 + A_I z^{-1}\right\}}{1 - z^{-1}} \qquad \text{volts/volt}, \qquad (45)$$

$$K_I = \frac{10.5 * (10.74T + 2)}{21.48}$$

$$A_I = \frac{10.74T - 2}{10.74T + 2}$$

$$G_C(z) = \frac{K_C * (1 + z^{-1})}{1 + A_C z^{-1}} \qquad \text{volts/volt}, \qquad (46)$$

$$K_C = \frac{34846.30T}{1024.59T + 2}$$

$$A_C = \frac{1024.59T - 2}{1024.59T + 2}$$

$$G_M(z) = \frac{K_{MA} + K_{MB}*z^{-1} + K_{MC}*z^{-2}}{1 + K_{MD}*z^{-1} + K_{ME}*z^{-2}} \qquad \text{(rad/sec)/amp}, \qquad (47)$$

$$K_{MA} = K_a - K_1 + K_2$$

$$K_{MB} = (-K_a*\exp(-K_c*T)) - (-K_a*\exp(-K_b*T)) + (K_1) + (K_1*\exp(-K_c*T)) - (K_2) - (-K_2*\exp(-K_c*T))$$

$$K_{MC} = (K_a*\exp(-K_b*T-K_c*T)) - (K_1*\exp(-K_c*T)) + (K_2*\exp(-K_c*T))$$

K_{MD} = $(-\exp(-K_c{*}T))$

K_{ME} = $(\exp(-K_b{*}T-K_c{*}T))$

$K_a = 0.28222$

$K_b = 1.181$

$K_c = 98.91$

$K_1 = 0.2856304$

$K_2 = 3.41047e\text{-}3$

$G_G(z) =$

$$\frac{K_{Ga}{*}T^4{*}(1 + 4z^{-1} + 6z^{-2} + 4z^{-3} + z^{-4})}{K_{GA} + K_{GB}z^{-1} + K_{GC}z^{-2} + K_{GD}z^{-3} + K_{GE}z^{-4}}$$

volts/(rad/sec) . (48)

$K_{GA} = K_{G1}{*}K_{G4}$

$K_{GB} = (K_{G1}{*}K_{G5})+(K_{G2}{*}K_{G4})$

$K_{GC} = (K_{G1}{*}K_{G6})+(K_{G2}{*}K_{G5})+(K_{G3}{*}K_{G4})$

$K_{GD} = (K_{G2}{*}K_{G6})+(K_{G3}{*}K_{G5})$

$K_{GE} = K_{G1}$

$K_{G1} = 4 + (2{*}K_{Gb}{*}T) + (T^2{*}K_{Gc})$

$K_{G2} = -8+ (2{*}T^2{*}K_{Gc})$

$K_{G3} = 4 - (2{*}K_{Gb}{*}T) + (T^2{*}K_{Gc})$

$K_{G4} = 4 + (2*K_{Gd}*T) + (T^2*K_{Ge})$

$K_{G5} = -8 + (2*T^2*K_{Ge})$

$K_{G6} = 4 - (2*K_{Gd}*T) + (T^2*K_{Ge})$

$K_{Ga} = 6.406611384e12$

$K_{Gb} = 766.41$

$K_{Gc} = 681377.11$

$K_{Gd} = 2668.20$

$K_{Ge} = 3046393.90$

$$G_F(z) = \frac{1 + z^{-1}}{1 + A_F z^{-1}} \qquad \text{volts/volt,} \qquad (49)$$

$$K_F = \frac{245.68T}{148T + 2}$$

$$A_F = \frac{148.27T - 2}{148.27T + 2}$$

$$G_{PA}(z) = 1 \qquad \text{amps/volt.} \qquad (50)$$

3.2.3 Application of ARW to Discrete System

To complete the digital design it was necessary to replace the existing analog ARW with a discrete design. Observe the design of the existing analog filter/integrator and summing amplifier shown in Figure 81 (to understand the operation of the analog ARW design more fully). Table 1 delineates the values of the components in the schematic. The operational amplifier has three functions. It conditions the feedback signal from the rate gyro to filter out noise which may exist in the sensor and to filter out high frequency noise from the antenna platform. It sums the amplified command with the feedback signal to generate the error signal. Lastly, it provides an initial compensation signal using a proportional and integral controller.

The diodes clamp the output to 0.6 Volts and effectively cause the capacitor to hold its charge if the output exceeds 0.6 Volts. It is these diodes that perform the ARW function. A model of the design (shown in Figure 82) reduces to the model shown in Figure 83. This matches the block diagram of 72.

In the discrete design, a CAW implementation (as discussed in Section 3.1.1) was chosen to perform the ARW compensation. The integrator/filter is a proportional-integral controller; thus, there are three variations on using the CAW. The ARW compensation can be implemented around the entire controller including both the proportional and integral portion, around the integrator and its gain, or only around the integrator itself. Figures 84 - 86 illustrate each of these approaches respectively.

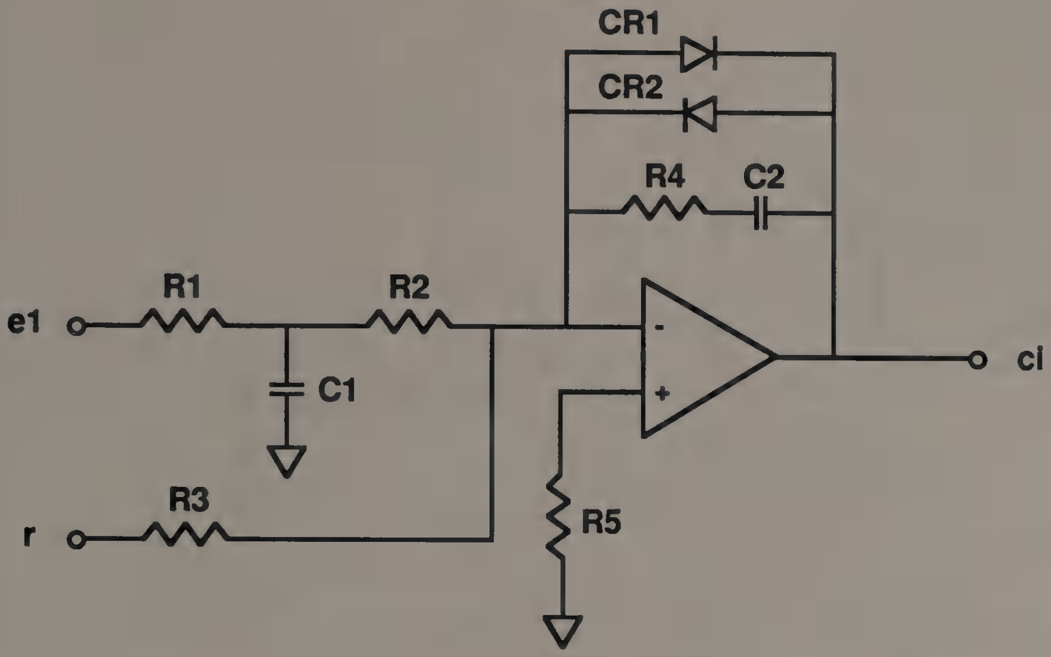


Figure 81. Track Radar Rate Servo Integrator/Filter/Summing Amp Schematic Design

Table 1. Component Values for Integrator/Filter

COMPONENT	VALUE
R1	28.7K
R2	28.7K
R3	95.3K
R4	93.1K
R5	28.7K
C1	47uF
C2	47uF
CR1	Vb = 0.6 volts
CR2	Vb = 0.6 volts

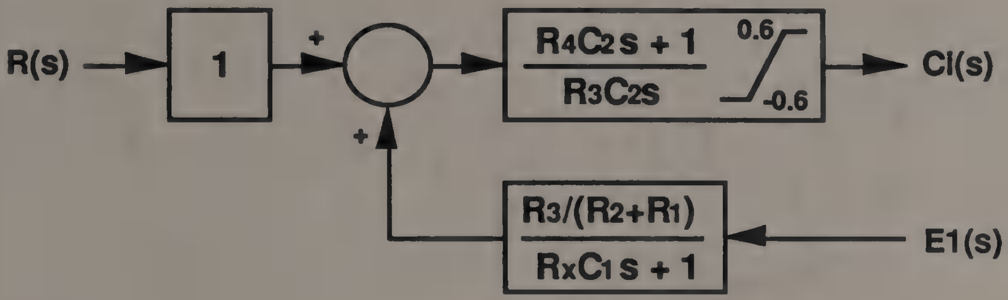


Figure 82. Track Radar Rate Servo Integrator/Filter/Summing Amp Block Diagram

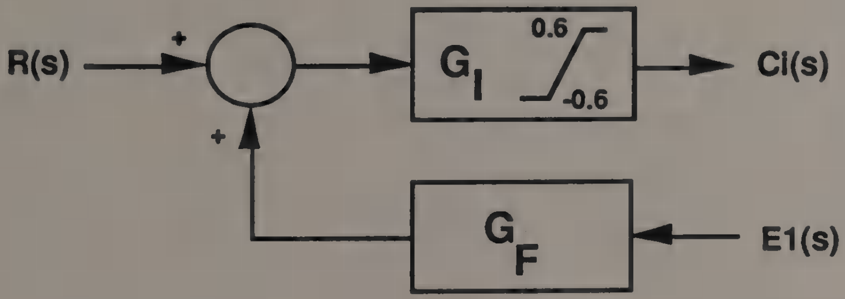


Figure 83. Track Radar Rate Servo Integrator/Filter/Summing Amp Model

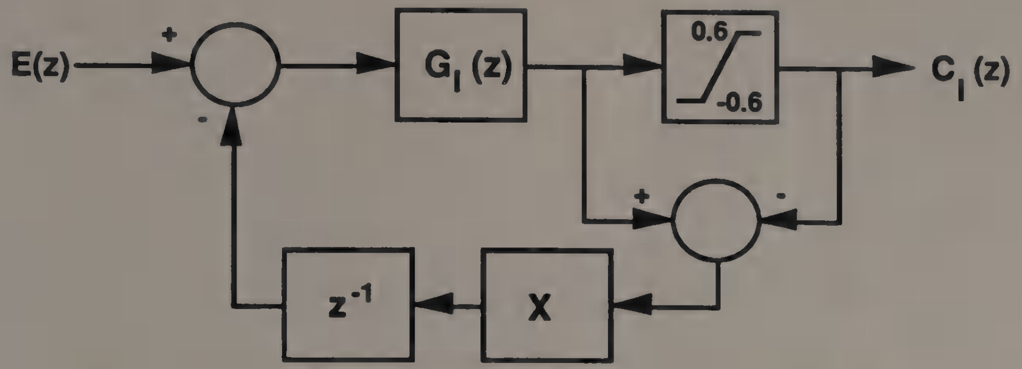


Figure 84. Discrete Design of Integrator/Filter (CAW around entire G_I)

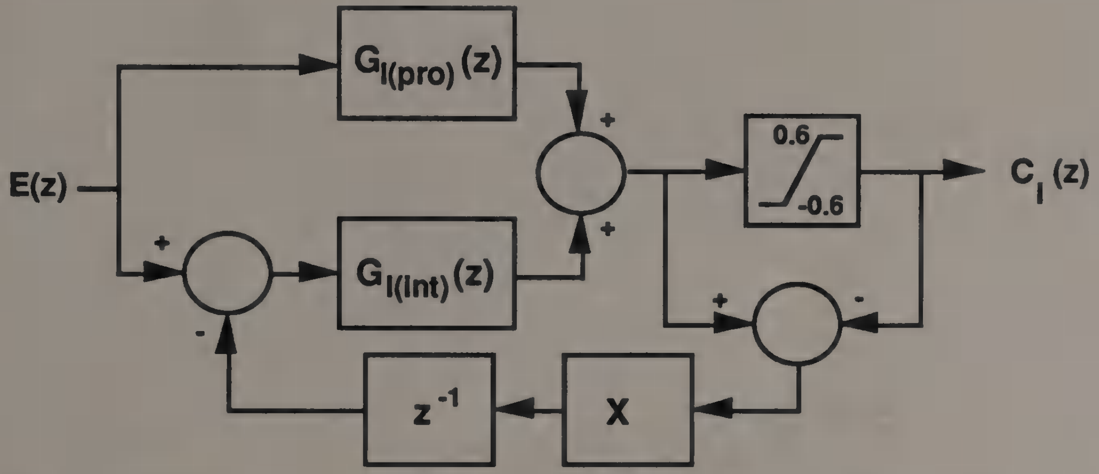


Figure 85. Discrete Design of Integrator/Filter (CAW around integrator and its gain)

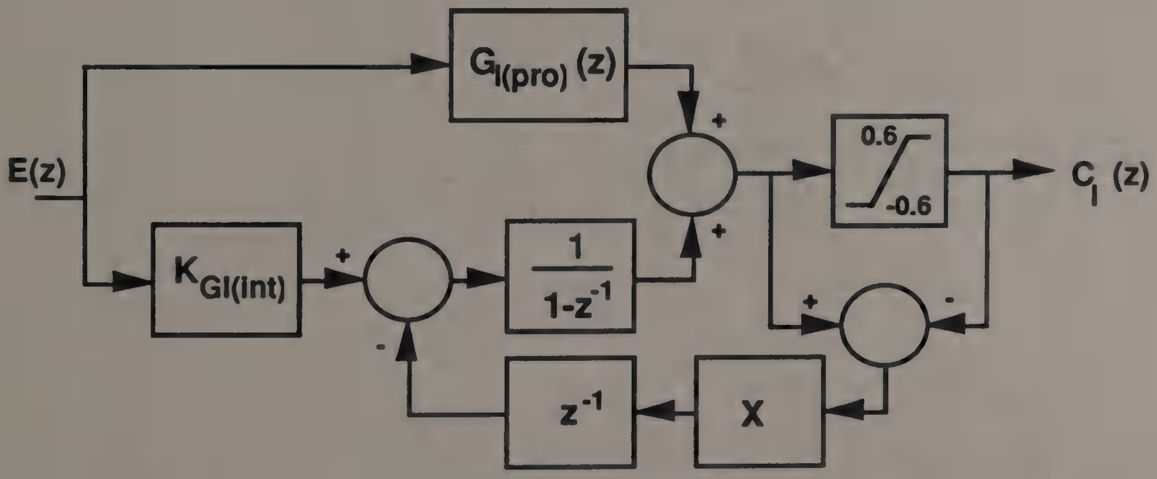


Figure 86. Discrete Design of Integrator/Filter (CAW around integrator only)

The design of Figure 84 was chosen, for a first attempt, with the CAW implemented around all of G_I (both the integral and proportional portions). The transfer function of the inner loop, T_{ic}, using this implementation is

$$T_{ic}(z) = \frac{K_t*X(1+A_t z^{-1})}{1 + z^{-1}(K_t*X-1) + z^{-2}(K_t*X*A_t)} , \tag{51}$$

where

$$K_t = \frac{10.5*(10.74*T+2)}{(2*10.74)} * \frac{z + A_t}{z - 1} , \tag{52}$$

$$A_t = \frac{(10.74*T-2)}{(10.74*T-2)} . \tag{53}$$

Although this approach is proposed by Glattfelder [3], as discussed in section 2, it proved unworkable for this application. The problem is the same as that discussed in the simple examples of Sections 3.1.1 and 3.1.2: the additional delay within the feedback makes the closed-loop transfer function of the inner loop of second order. For all values of $X>0$ and $0<T<1$, there is at least one root $z = +1$. It is not possible, therefore, to create a stable inner loop design using this implementation.

The next attempt incorporated the design of Figure 85 with the CAW implemented around the integrator portion of G_I along with its gain. The transfer function of the inner loop, T_{ic}, using this implementation is

$$T_{ic}(z) = \frac{K_t*T*X(1+z^{-1})}{1 + z^{-1}(K_t*T*X-1) + z^{-2}(K_t*T*X)} . \tag{54}$$

where

$$K_t = \frac{10.5*T}{2} . \tag{55}$$

This approach was found to be stable for reasonable values of X and T, and root locus plots of $T_{ic}(z)$ were plotted for values T and X. The maximum value of X possible before loop instability is plotted for values of T in Figure 87. Also plotted is the value of X vs. T for an inner loop response damping ratio of 0.9. This curve will be used in the next section to choose values of X and T for the final design. Note that a gain of at least ten is desirable in order for the loop to function correctly and this dictates a sample frequency of at least 180 Hz. This bounds the lower limit of the sample period T. The final system sample period chosen later will be based on this and other factors.

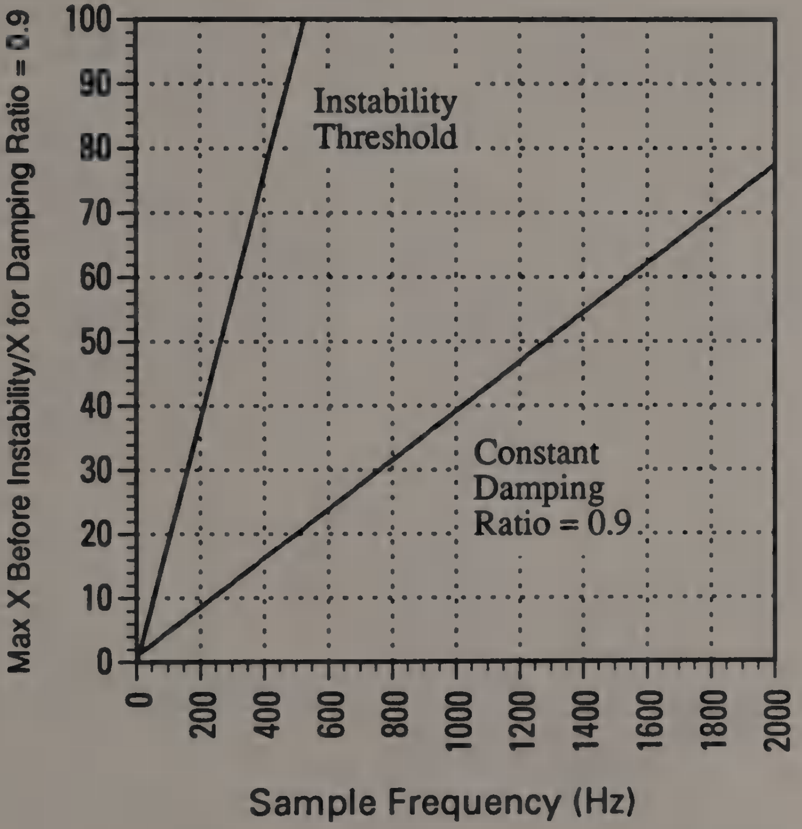


Figure 87. Maximum Gain X (to ensure Inner Loop Stability) and Gain X (for damping = 0.9) vs. Sample Frequency (integrator and its gain)

It is also possible to implement the CAW around only the integrator portion and not its gain as shown in Figure 86. For this implementation, the transfer function of the inner loop, T_{ic}, is

$$T_{ic}(z) = \frac{K_t*T*X(1+z^{-1})}{1 + z^{-1}(K_t*T*X-1) + z^{-2}(K_t*T*X)} \tag{56}$$

where

$$K_t = \frac{T}{2}. \tag{57}$$

The maximum value of X possible before loop instability is plotted for values of T in Figure 88 along with X vs. T for an inner loop response damping ratio of 0.9. In this case, gain of at least ten requires a sample frequency of at least 10 Hz.

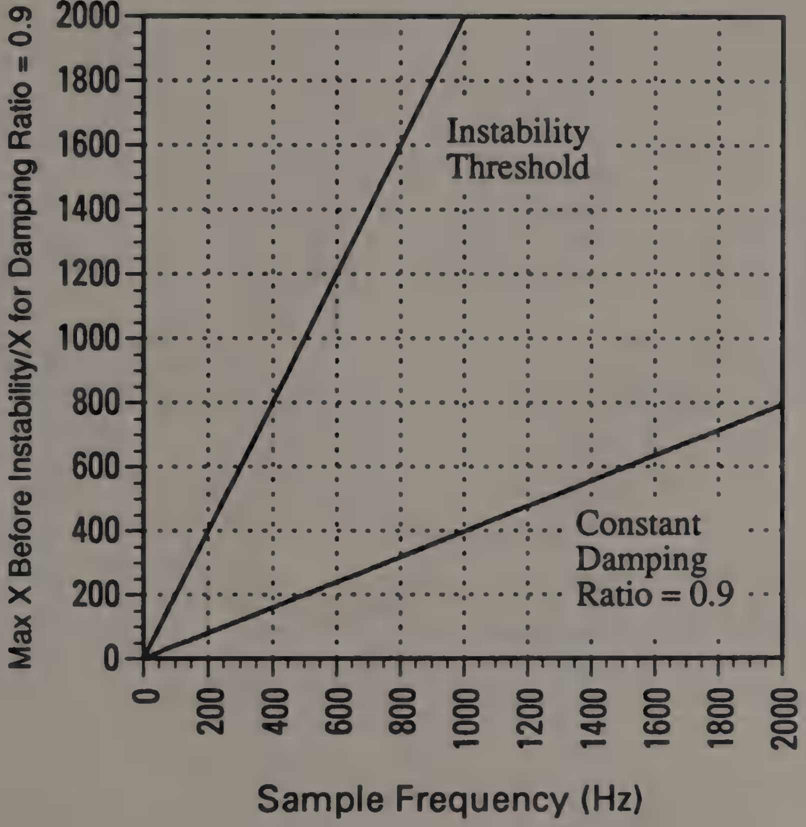


Figure 88. Maximum Gain X (to ensure Inner Loop Stability) and Gain X (for damping = 0.9) vs. Sample Frequency (integrator only)

The next section will discuss the process employed for selecting the system sample period and the use of the above plots to select the optimum feedback gain X.

3.2.4 Selecting Sample Period and Feedback Gain

The process for selecting the inner loop feedback gain, X, is to determine the desired sample frequency from the closed-loop bandwidth and to then choose the X from the previous plots that meets a damping ratio of 0.9. This will allow the inner loop to respond fast enough to affect the CAW compensation properly but will limit the overshoot of the inner loop to avoid chatter.

Figure 89 shows the linear open-loop frequency response for the original continuous system. The closed loop bandwidth is determined from the crossover point and is approximately 50 Hz. The sample rate for the digital design was chosen (using criteria proposed by Franklin, Powell, and Workman [12]) to be approximately 20 times the closed-loop bandwidth of the system. This establishes a sample frequency of approximately 1000 Hz and a sample period, T, of 0.001 seconds.

Having established a sample frequency, T, the selection of the feedback gain for the CAW is made from Figures 87 and 88. For implementing the CAW around the integrator and its gain, the X gain along the 0.9 damping ratio curve is approximately 38. For implementing the CAW around the integrator only, the X gain is approximately 387. It is not surprising that these two numbers differ by a factor of about ten. The only difference in the two implementations is whether the integrator's gain of 10.5 is included in the inner loop as part of the controller, or included in the inner loop as part of the feedback gain. Either way, the final design of the inner loop is essentially identical.

Given that both are viable approaches, the first approach was chosen (CAW around integrator and its gain) for the final design. A sample period of T = 0.001 and a feedback gain X = 38 was selected. The performance of the completed discrete design to a step input is shown in Figure 91. The output, *y[kT]*, and the integrator/filter control signal, *ci[kT]* is shown. The design offers less that 1% overshoot and settles out in approximately 0.37 seconds.

Figure 92 depicts the integrator/filter signal at a time scale to show the proper damping of the signal. Minimal overshoot is present and the signal does not go below zero, therefore, no chatter will result.

The output and integrator/filter signals for damping ratios of 0.85 and 0.70 are shown in Figures 93-97. When the damping is below 0.85, chatter begins to shown in the system. The choice of the gain and sample period to yield a damping ratio of 0.9 is thus determined to be a good one.

One of the significant advantages of the ARW compensation, as mentioned in Section 3.1.1 is that it creates a system that is very well behaved with regard to overshoot. The overshoot remains very consistent regardless of the magnitude of the step change commanded to the system. This is illustrated for this system in Figure 97 for step rate inputs of 0.3, 0.5, 0.8, 1.0 and 1.5 radians/second.

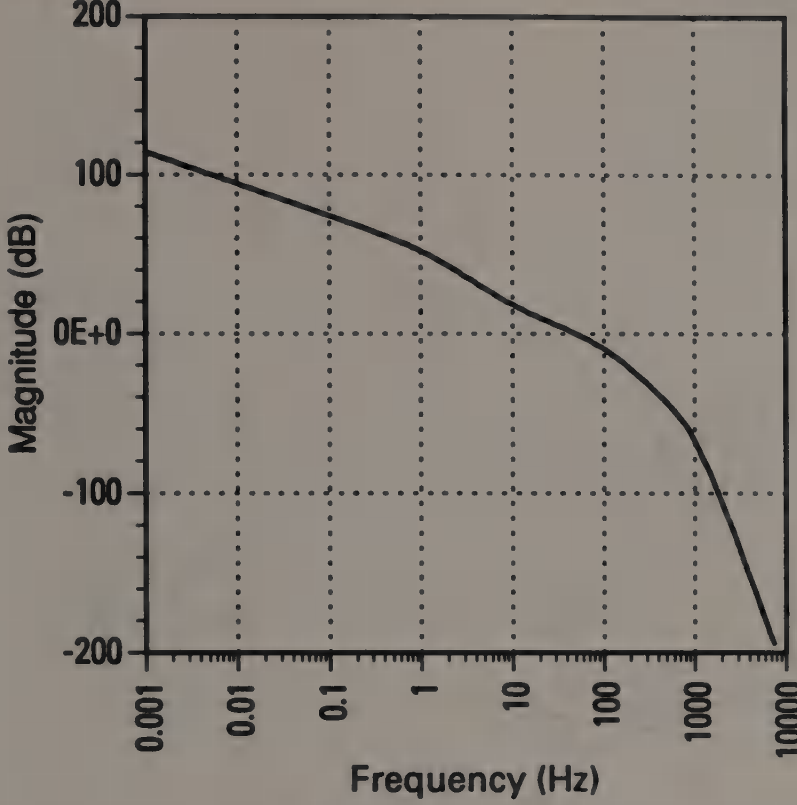


Figure 89. Linear Closed Loop Frequency Response (Gain) Continuous Track Rate Servo

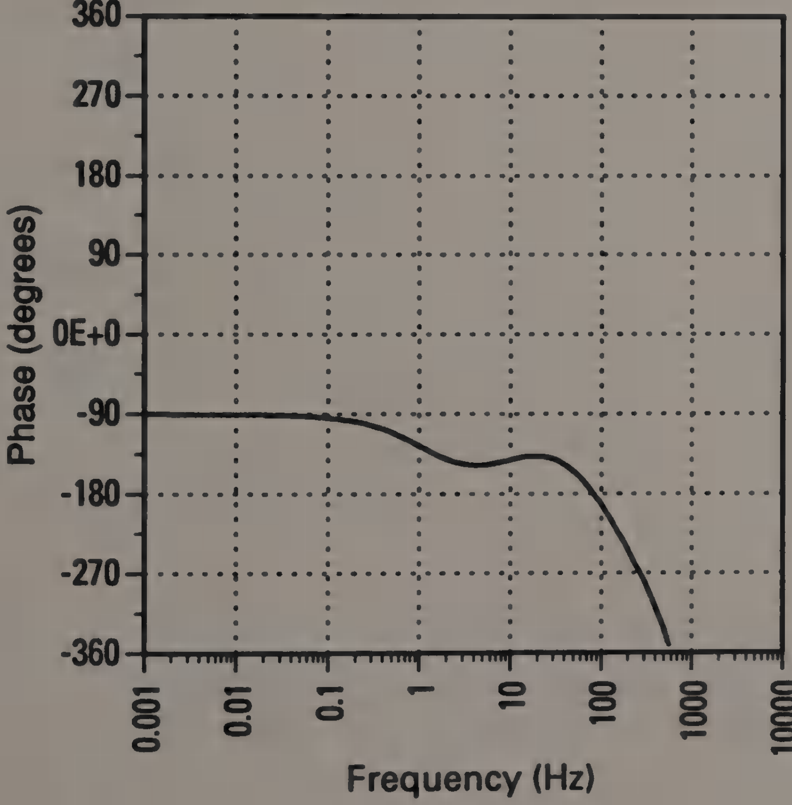


Figure 90. Linear Closed Loop Frequency Response (Phase) Continuous Track Rate Servo

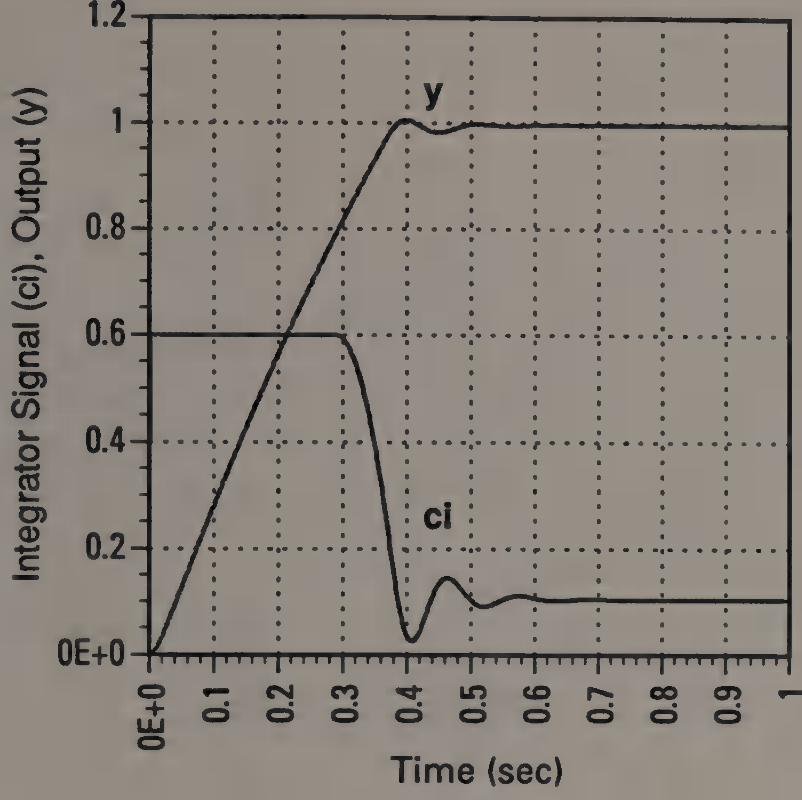


Figure 91. Step Response of Optimized Discrete Track Rate Design

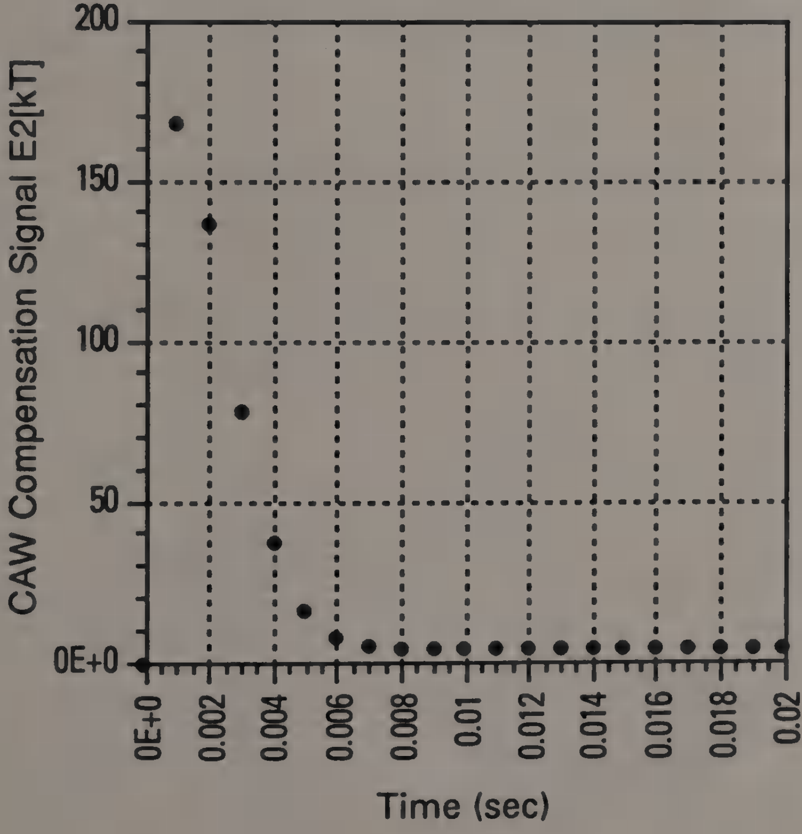


Figure 92. Integrator/Filter Signal for Optimized Discrete Track Rate Design

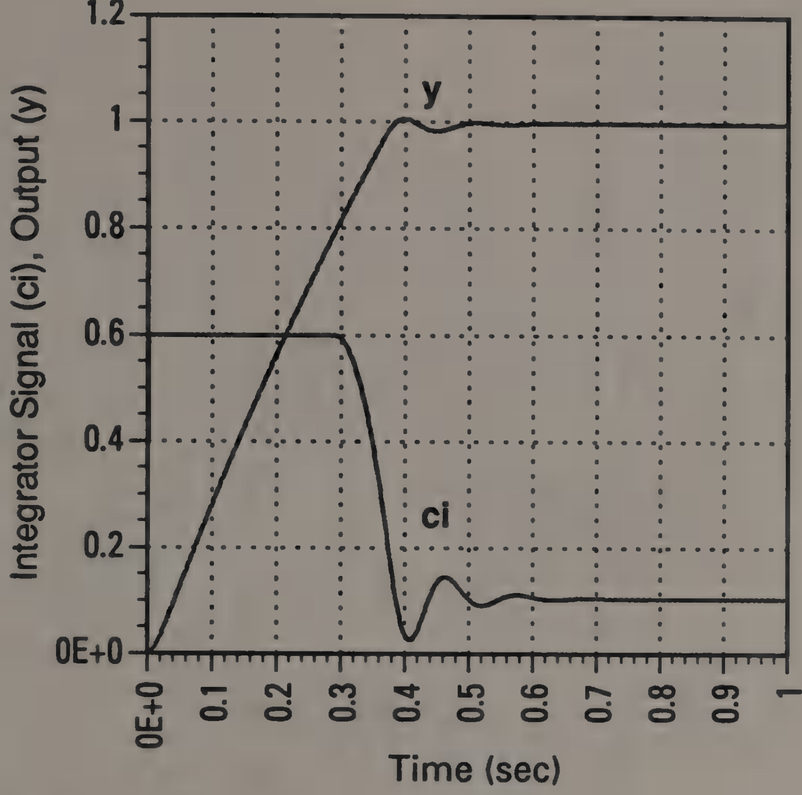


Figure 93. Step Response of Discrete Track Rate Design (damping = 0.85) Output y[kT] and Integrator/Filter Signal ci[kT]

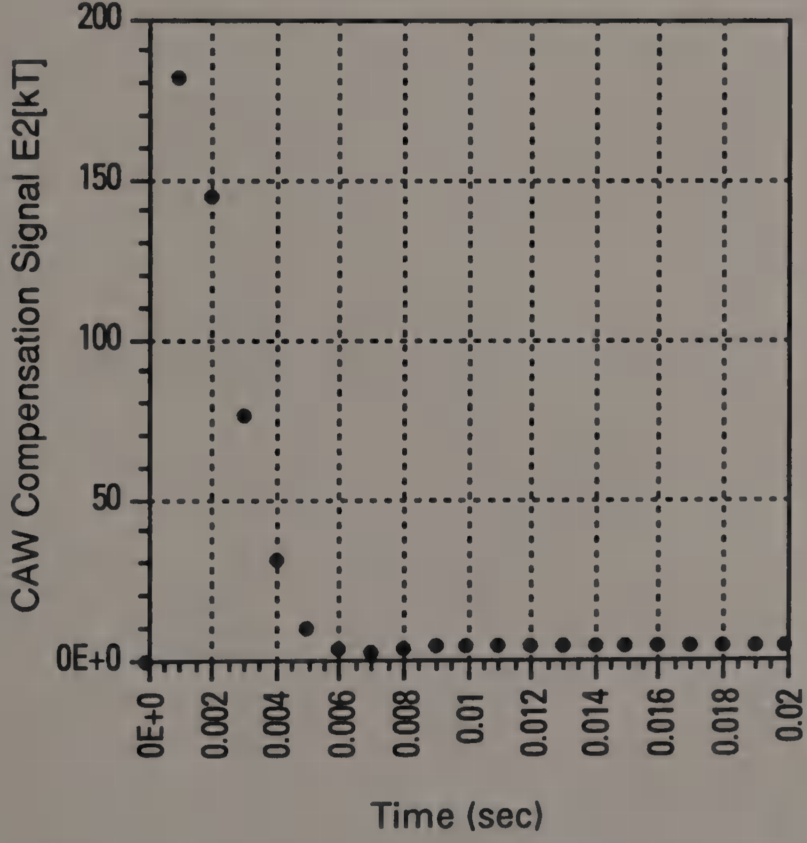


Figure 94. Step Response of Discrete Track Rate Design (damping = 0.85) CAW Compensation Signal eh[kT]

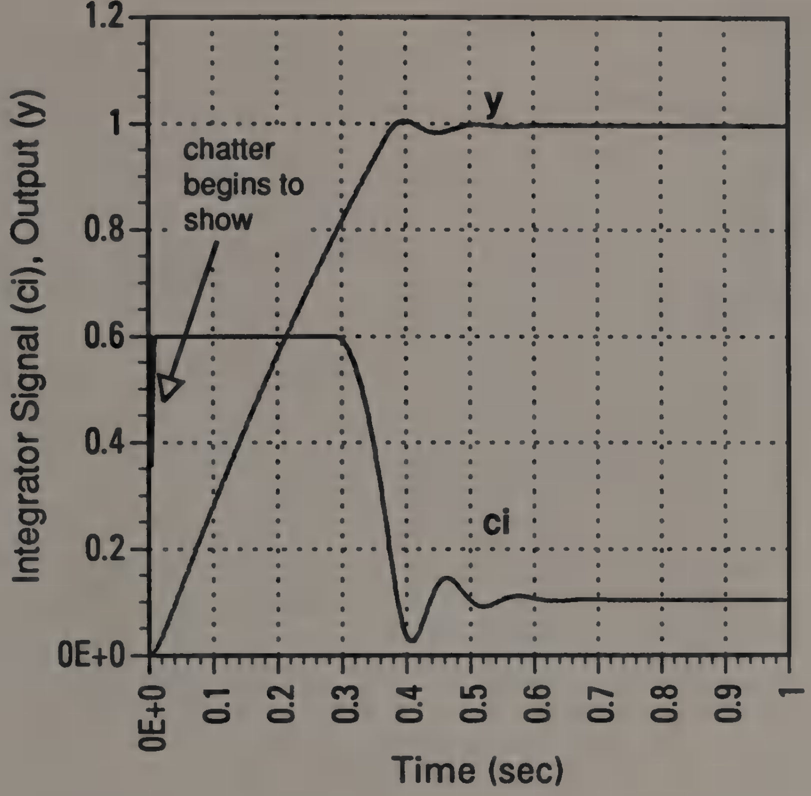


Figure 95. Step Response of Discrete Track Rate Design (damping = 0.70) Output y[kT] and Integrator/Filter Signal ci[kT]

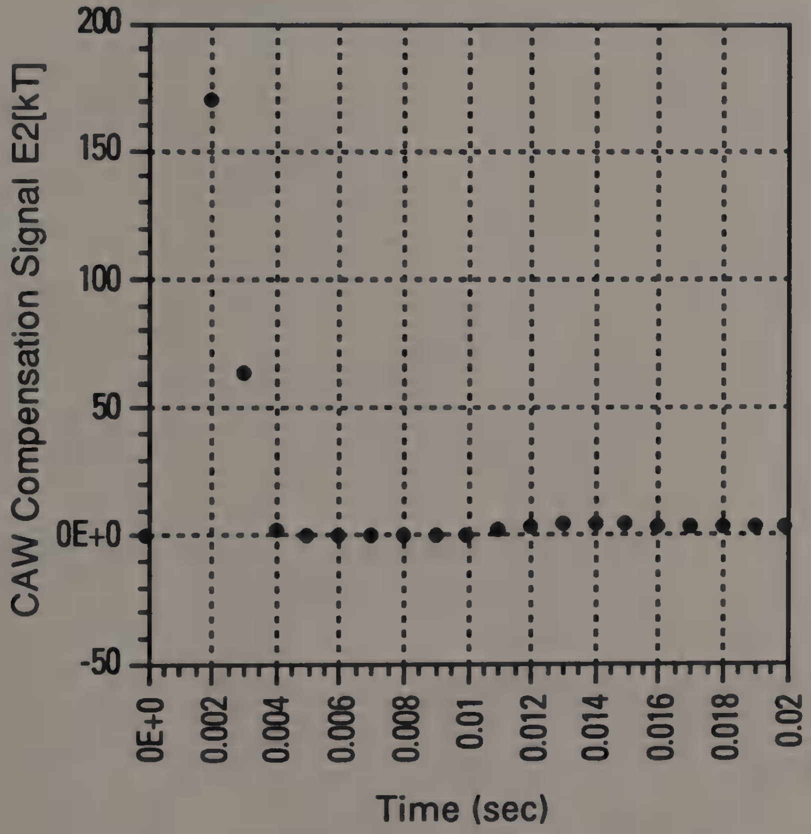


Figure 96. Step Response of Discrete Track Rate Design (damping = 0.70) CAW Compensation Signal eh[kT]

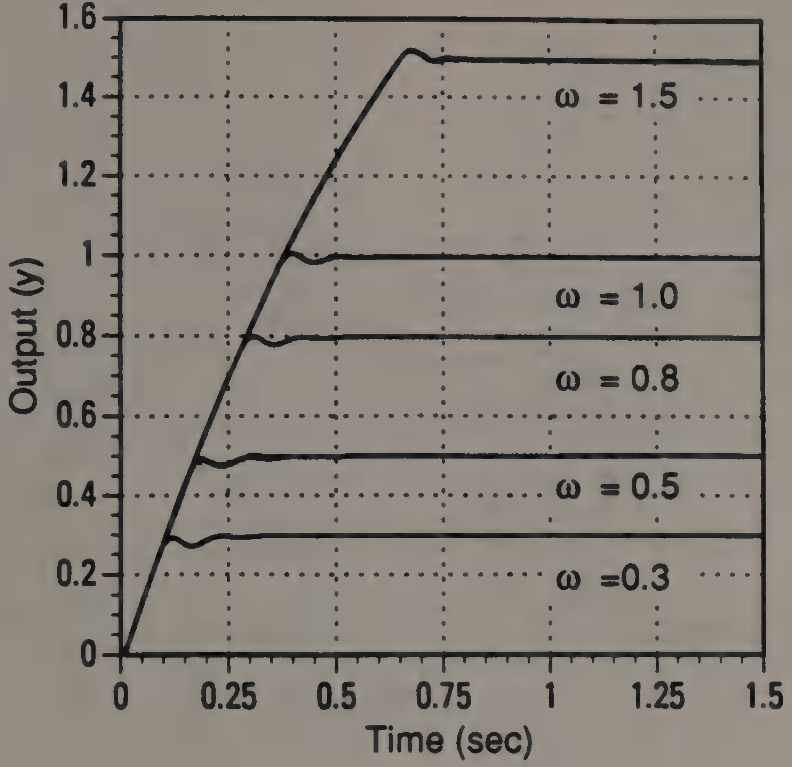


Figure 97. Track Rate Output, Steps of ω=0.3, 0.5, 0.8, 1.0, 1.5 rad/sec

CHAPTER 4
CONCLUSIONS

Nearly all systems have some type of control input saturation, and often it is the dominant nonlinearity. The problem of plant input saturation has been considered via the study of three examples along with the associated consequence - reset-windup of the controller. Several schemes exist for enhancing the stability and performance of continuous-time systems subject to reset-windup. Two schemes were investigated in detail: the conventional antireset-windup (CAW) implementation and the override signal (OS) implementation. These schemes can have significant effect on reducing the overshoot and settling time in a continuous-time system with reset-windup caused by saturation.

These schemes can also be applied to discrete systems but care must be taken in applying these schemes in order to avoid chatter in the system. Chatter results from overshoot in the antireset-windup compensation signal if the signal overshoots the control point and drives the system in and out of saturation. The paper discusses the development of discrete implementations of the CAW and the OS and establishes design criteria for avoiding chatter and developing a good discrete design. The criteria establishes a method of analyzing the system based on the discrete response of the inner loop (the CAW compensation loop).

The criteria developed shows how to choose the sample period, T, and the feedback gain within the inner loop so that the overshoot of the CAW compensation signal is adjusted to be minimal. For a first order controller, the criteria developed is to adjust the response of the inner loop for a second order damping ratio of 0.9. This is done through judicious selection of T and the feedback gain. In theory, both the discrete CAW and OS should be applicable in systems with controllers of higher order; further effort could investigate the schemes for systems with more complex controllers.

It was also determined that the CAW and the OS implementations, while appearing to be different in their operation are, in fact, the same. Both operate on the method of an inner loop (the ARW compensation loop) which modifies the control signal to avoid saturation by adjusting the control signal to keep the input to the saturation at, or close to, the saturation point. The schemes do not allow the control signal to drive far into saturation which keeps the windup of the controller small.

The design of an ARW compensation scheme using the CAW implementation was demonstrated for an existing analog control system for a tracking antenna. A controller was designed using "design by discrete equivalent" and the bilinear transform and the sample time and feedback gain values were determined to produce a usable design. The design demonstrates performance equal to, or better than, the original design.

REFERENCES

1. Franklin, G.F., J.D. Powell, and M.L.Workman, *Digital Control of Dynamic Systems*. Addison-Wesley Publishing Company, Reading MA, 1990, pg 512.

2. Doyle, J.C., R.S. Smith, and D.F. Enns, "Control of Plants with Input Saturation", *Proceedings of the 1987 American Control Conference*, June 1987, pg. 1034.

3. Glattfelder, A.H. and W. Schaufelberger, "Stability Analysis of Single Loop Control System with Saturation and Antireset-Windup Circuits", *IEEE Transactions on Automatic Control*, Vol AC-28, No. 12, December, 1983, pg. 1074.

4. Buckley, P.S., "Designing Override and Feedforward Controls - I", *Control Engineering*, Vol. 18, No. 8, 1971, pg. 48.

5. Doyle, J.C., R.S. Smith, and D.F. Enns, "Control of Plants with Input Saturation", *Proceedings of the 1987 American Control Conference*, June 1987, pg. 1035.

6. Glattfelder, A.H., W. Schaufelberger and J. Todtli, "Diskrete Proportional-Integral-Differential Refler mit Anti-Windup Massnahmen", *SGA Bulletin*, No. 1, pp. 12-22; No. 2, pp. 19-28, 1983.

7. Glattfelder, A.H. and W. Schaufelberger, "Stability Analysis of Single Loop Control System with Saturation and Antireset-Windup Circuits", *IEEE Transactions on Automatic Control*, Vol AC-28, No. 12, December, 1983, pg. 1075.

8. Buckley, P.S., "Designing Override and Feedforward Controls - I", *Control Engineering*, Vol. 18, No. 8, 1971, pp. 48-51.

9. Doyle, J.C., R.S. Smith, and D.F. Enns, "Control of Plants with Input Saturation", *Proceedings of the 1987 American Control Conference*, June 1987, pp. 1034-1039.

10. Zames, G., "On the Input-Output Stability of Time-Varying Nonlinear Feedback Systems, Part II: Conditions Involving Circles in the Frequency Plane and Sector Nonlinearities", *IEEE Transactions on Automatic Control*, Vol AC-11, 1966, pp. 465-476.

11. Glattfelder, A.H. and W. Schaufelberger, "Stability Analysis of Single Loop Control System with Saturation and Antireset-Windup Circuits", *IEEE Transactions on Automatic Control*, Vol AC-28, No. 12, December, 1983, pp. 1074-1081.

12. Franklin, G.F., J.D. Powell, and M.L.Workman, *Digital Control of Dynamic Systems*. Addison-Wesley Publishing Company, Reading MA, 1990, pg. 512.

13. Franklin, G.F., J.D. Powell, and M.L.Workman, *Digital Control of Dynamic Systems*. Addison-Wesley Publishing Company, Reading MA, 1990, pg. 483.

14. Slivinsky, C. and J. Borninski, "Control System Compensation and Implementation with the TMS32010", Texas Instruments, Dallas, TX, 1990.

15. Katz, P., *Digital Control Using Microprocessors*, Prentice-Hall, 1981.

APPENDIX A
DESIGN BY DISCRETE EQUIVALENT

The most basic of control systems consists of a plant and a controller. The plant is that which is controlled and may be a motor for positioning of a radar antenna; an aircraft control surface actuator; a chemical process flow-control valve, or countless other mechanisms. The controller is that which accepts the commands for controlling the plant and generates control signals which make the plant behave dynamically in some desired manner with a desired level of performance. In most analog control systems, this controller is a compensation network, or compensator, designed with op-amps and discrete components. In a digital control system, this compensator is a control algorithm which determines a command to be applied to the plant at discrete intervals (every sample time).

There are many reasons why one might wish to transform or convert an analog, or continuous, compensator into a digital, or discrete, one. The first reason is that there are a number of design techniques which have been developed for design of control systems in the continuous, or s-plane; Bode, root locus, and Nyquist are three examples. These techniques are well established, many designers are comfortable using them, and many design tools have been developed around them. A designer, therefore, can design the compensator for a given system in the continuous plane and then convert it to a discrete controller for implementation in a digital computer.

Secondly, there are many existing analog control systems which have been designed and in operation for a long time. This is particularly true in the military where a weapons systems is designed to be in service for as many as twenty or thirty years. As these systems are upgraded, or replaced, the existing analog control systems are often replaced with digital ones. Oftentimes, the performance need not be significantly changed: the basic theoretical design of the control system can be retained but must be modified for implementation with a computer.

The process of converting a system expressed as a set of transfer functions in the s-plane, to a system expressed as a set of transfer functions in the z-plane is "design-by-discrete equivalents", or "design-by-transform methods". Design by discrete equivalent is a viable and much used method.

A number of techniques have been developed to transform a continuous system into a discrete one. These techniques are often referred to as discrete transform methods. Given a transfer function, H(s), (as shown in Figure 98) with an input signal *e(s)* and output signal *u(s)*, the discrete equivalent of this transfer function can be realized by sampling the input signal *e(t)* to produce a signal *e[kT]* and passing this sampled signal through a discrete version, $H_d(z)$, of the analog compensator.

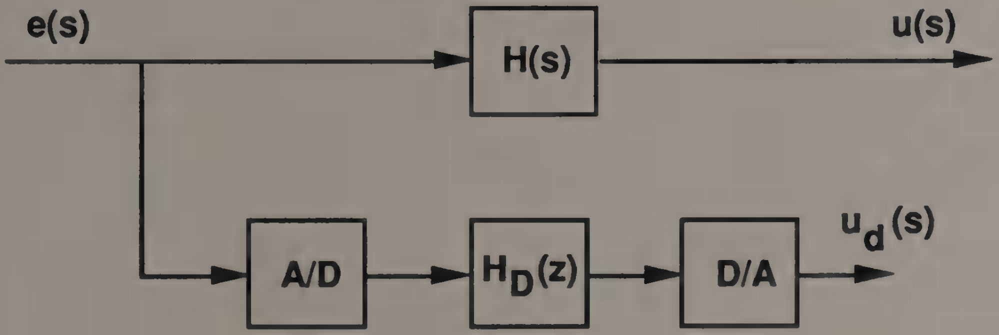


Figure 98. Continuous Transfer Function and Discrete Equivalent

The control signal is then passed through a D/A converter to produce a new $u_d(s)$. If the sample rate, T, is infinite (or sufficiently high) and the conversion technique used on the analog compensator yields a discrete compensator which asymptotically (in T) represents its analog counterpart, then $u_d(s)$ will equal *u(s)*. Since it is impossible to make T infinite, and because all transform techniques convert the analog compensator to a digital one with varying degrees of accuracy, it is not possible to get $u_d(s)$ to be exactly equal to *u(s)*. It is therefore necessary to select a discrete filter $H_D(z)$ such that $u_d(s)$ approximates *u(s)* as closely as possible. Mathematically, the objective is to select $H_D(z)$ such that

$$H_D(e^{j\omega T}) \approx H(j\omega) \tag{58}$$

for $\omega < a$ where $a >> \omega_b$, the bandwidth of the compensator. To achieve this, it is necessary that

$$T >> \frac{\pi}{a}. \tag{59}$$

If both of these conditions are met, $u_d(t)$ will be approximately equal to *u(t)*. Given that $u_d(t)$ will be approximately equal to *u(t)* if conditions (58) and (59) are met, it is desirable to develop transform techniques to accomplish condition (58).

Condition (59) is met by selecting a sufficiently high sample rate [13]. The selection of the sample rate is the result of a tradeoff of several factors. It is desirable to have T, the sample period, as small as possible to get the most accurate representation of the continuous filter. However, cost is the motivation to have the sample period large. The larger the sample time, the slower the sample rate and the slower the computer necessary for a given control function. Lower sample rates also require slower A/D and D/A converters which translates to lower cost.

Several techniques exist to transform H(s) to $H_D(z)$. Some of the more popular techniques are:

- Step-Invariant method
- Bilinear method
- Bilinear with pre-warping method
- Pole-Zero mapping method
- Mapping differentials method
- Impulse-Invariance method.

For the purpose of this thesis, the bilinear transform will be used to create a discrete design of the controller and the step-invariant transform will be used to create a discrete model of the plant. The pole-zero transform method was also investigated for the controllers with similar results.

Bilinear Transform Method

The Bilinear transform, known also as Tustin's Method, approximates a given transfer function H(s) with a discrete equivalent $H_D(z)$ by replacing each s in the transfer function by

$$s = \frac{2}{T} * \frac{(z-1)}{(z+1)}. \tag{60}$$

This approximation is a map from the s-plane to the z-plane. Tustin's method exactly maps the stable region of the s-plane (the left-half of the plane) into the stable region of the z-plane (the interior of the unit circle). This substitution maps low analog frequencies into approximately the same digital frequencies but can produce a highly nonlinear mapping for the high frequencies [14]. Although some distortion results at higher frequencies since the entire $j\omega$-axis of the s-plane is mapped into the 2π-length of the unit circle, it provides a close approximation to the analog compensator and is the most commonly used technique [15]. The bilinear method is often supplemented with a technique called prewarping which attempts to correct for this distortion at the critical frequency of the compensator. However, the bilinear transform without the prewarping was found to be sufficient for the designs within this thesis. An example of the bilinear transform is as follows.

Assume a first-order filter as before, where

$$H(s) = \frac{a}{s+a}. \tag{61}$$

Substitution of equation (60) into equation (61) gives the discrete equivalent of equation (61),

$$H_D(z) = \frac{a}{\frac{2\,(z-1)}{T(z+1)} + a} \tag{62}$$

which reduces to

$$H_D(z) = \frac{az+a}{\left(a + \frac{2}{T}\right) z + \left(a - \frac{2}{T}\right)} . \tag{63}$$

This equivalent filter is, in most cases, a sufficient discrete representation of the original filter.

<u>Step-Invariant Method</u>

The Step-Invariant method, also known as the Hold-Equivalence method, approximates a given transfer function P(s) with a discrete equivalent $P_D(z)$ by the transform

$$P_D(z) = (1-z^{-1})\, Z\left\{\mathfrak{I}^{-1}\left\{\frac{P(s)}{s}\right\}\right\} , \tag{64}$$

where Z is the Z-transform and $\mathfrak{I}^{-1}$ is the inverse-Laplace transform. This transform method is derived from the use of a zero-order hold to approximate a continuous input signal. An example of the Step-Invariant transform is as follows. Assume a first-order filter such as

$$P(s) = \frac{a}{s+a} . \tag{65}$$

From equation (65) it can be seen that

$$\frac{P(s)}{s} = \frac{a}{s(s+a)} = \frac{1}{s} - \frac{1}{s+a} . \tag{66}$$

The inverse-Laplace transform of equation (66) is

$$\mathfrak{I}^{-1}\left\{\frac{P(s)}{s}\right\} = \mathfrak{I}^{-1}\left\{\frac{1}{s}\right\} - \mathfrak{I}^{-1}\left\{\frac{1}{s+a}\right\}$$

$$= 1 - e^{-at} . \tag{67}$$

The Z-transform of equation (67) is

$$Z\{1 - e^{-at}\} = \frac{1}{1-z^{-1}} - \frac{1}{1-e^{-aT}z^{-1}} ,$$

which results in

$$Z\{1 - e^{-at}\} = \frac{(1-e^{-aT}z^{-1}) - (1-z^{-1})}{(1-z^{-1})(1-e^{-aT}z^{-1})} . \tag{68}$$

Multiplying (68) by $(1-z^{-1})$ yields the Step-Invariant equivalent of equation (65):

$$P_D(z) = \frac{1-e^{-aT}}{z-e^{-aT}} . \tag{69}$$

APPENDIX B
ANALYTICAL SIMULATIONS
FOR CONTINUOUS AND DISCRETE SYSTEMS

Eight simulations were developed to study the systems discussed in the thesis and to generate the results presented. The eight simulations fall into two categories; continuous-time simulations and discrete-time simulations and are as follows:

<u>Continuous-Time Simulations</u>

- Simple Example (Section 2.1)
- Doyle and Smith Continuous System (Section 3.1.1.1)
- Complex Compensator Continuous System (Section 3.1.1.4)
- Glattfelder and Schaufelberger Continuous System (Section 3.1.2.1)
- Track Rate Servo Continuous System (Section 3.2.1)

<u>Discrete-Time Simulations</u>

- Doyle and Smith Discrete System (Section 3.1.1.2)
- Glattfelder and Schaufelberger Discrete System (Section 3.1.2.3)
- Track Rate Servo Discrete System (Section 3.2.2)

All simulations with the exception of the Track Rate Servo Continuous System simulation, were written in THINK Pascal™ 3.0 using an Apple Macintosh IIsi. The first four continuous simulations use a state-space representation for the system blocks and the fourth order Runge-Kutta method for computation of the system states. The Track Rate Servo Continuous System simulation was developed on a Sun Workstation™ using the

Matlab simulation tool Simulab™. The listings for all but the Track Rate Continuous System simulation are presented at the end of this appendix. The three discrete simulations were developed using standard difference equations for computation of the systems states at each sample time. The listing for all of the discrete simulations are presented at the end of this appendix.

CONTINUOUS-TIME SIMULATIONS

PROGRAM LISTINGS

```
program Simple_Example_of_Windup;


{*******************************************}
{                                           }
{   R.J. Spangenberger     08 Aug 1991      }
{                                           }
{   4th Order Runge-Kutta Continuous Simulation }
{   of Simple Windup Example                }
{                                           }
{*******************************************}



{program constants}
 const
   runga_kutta_interval = 0.01;                    { time step for each runge-kutta interval }
   printing_interval = 1;                          { print out every printing_interval computed state }
   run_time = [illegible];                         { run time of simulation in seconds }
   max_integrators = 4;                            { number of integrators in system }
   limit = 1;                                      { limit value of saturation block }
   X_gain = 10;                                    { gain of feedback in inner loop }

{program type definitions}
 type
   mat = array[1..max_integrators] of real;        { matrix of integrators }

{program variables}
 var
   rk_step_matrix:  array[1..4] of real;           { four runge-kutta constants }
   integrator_inputs, e, integrator_states, y: mat; { inputs, states and temporary variables }
   system_input: real;                             { command into system }
   t: real;                                        { time (seconds) }
   outfile: text;                                  { output file for data }
   [illegible], yy, r, e2, e1, e3, [illegible], u, [illegible] real; { auxiliary variables }



 function matrix_add (a, b: mat): mat;
  { add matrices a and b }
   var
    i: integer;
 begin
  for i := 1 to max_integrators do
    matrix_add[i] := a[i] + b[i];
 end;


 function matrix_multiply (a, b: mat): mat;
  { multiply matrices a and b }
   var
    i: integer;
 begin
  for i := 1 to max_integrators do
    matrix_multiply[i] := a[i] * b[i];
 end;



 function matrix_multiply_constant (a: mat; b: real): mat;
  { multiply matrix a by the value b }
   var
    i: integer;
 begin
  for i := 1 to max_integrators do
    matrix_multiply_constant[i] := a[i] * b;
 end;

 function matrix_divide_constant (a: mat; b: real): mat;
  { divide matrix a by the value b }
   var
    i: integer;
 begin
  for i := 1 to max_integrators do
    matrix_divide_constant[i] := a[i] / b;
 end;
```

```
procedure  set_rk_step_matrix;
{ initialize runge-kutta step constants }
begin
  rk_step_matrix[1]  :=  0.5;
  rk_step_matrix[2]  :=  0.5;
  rk_step_matrix[3]  :=  1;
  rk_step_matrix[4]  :=  2;
end;


procedure  initialize_integrator_outputs;
{ initialize the integrator states }
  var
    i:  integer;
begin
  for i := 1 to  max_integrators  do
    integrator_states[i]  :=  0;
  ee := 0;
  e1  := 0;
  e2  := 0;
  e3  := 0;
  yy  := 0;
  r := 0;
  s := 0;
  u := 0;
  c := 0;
end;


function  clamp  (input,  limit:  real):  real;
 {clamp  output  to  limit}
  var
    output:  real;
begin
  if  input  >  limit  then
    output  :=  limit
  else  if  input  <  -limit  then
    output  :=  -limit
  else
    output  :=  input;
  clamp  :=  output;
end;


function  compute_system_input  (t:  real):  real;
{ compute  input  to  system }
begin
  compute_system_input  :=  1;
end;


procedure  run_simulation  (var  t:  real);
{ run  simulation  from  t  =  0  to  t  =  run_time }
  var
    i:  integer;
    intermediate_value:  mat;
    temp_value:  real;
begin
  for i := 1 to 4 do
    begin
     { compute  system  command }
      system_input  :=  compute_system_input(t);

      {compute  auxiliary  variables}
      ee  :=  system_input  -  yy;
      u  :=  integrator_states[3];
      c  :=  clamp(u,  100000);
      yy  :=  (0.05  *  integrator_states[4])  +  (0.5  *  c);


      {compute  integrator  inputs  from  states}
      integrator_inputs[1]  :=  (-800  *  ee)  +  (-40.1  *  integrator_states[1])  +  (-404  *  integrator_states[2]);
      integrator_inputs[1]  :=  integrator_inputs[1]  +  (-40  *  integrator_states[3]);
      integrator_inputs[2]  :=  integrator_states[1];
      integrator_inputs[3]  :=  integrator_states[2];

      integrator_inputs[4]  :=  c;
      { update states through each runge-kutta minor step }
      case i of
        1:
          begin
            e := integrator_inputs;
            y := integrator_states;
            intermediate_value := matrix_multiply_constant(integrator_inputs, runga_kutta_interval * rk_step_matrix[i]);
            integrator_states := matrix_add(y, intermediate_value);
            t := t + (runga_kutta_interval / 2);
          end;
        2:
          begin
            integrator_states := matrix_multiply_constant(integrator_inputs, 2);
            e := matrix_add(e, integrator_states);
            intermediate_value := matrix_multiply_constant(integrator_inputs, runga_kutta_interval * rk_step_matrix[i]);
            integrator_states := matrix_add(y, intermediate_value);
          end;
        3:
          begin
            integrator_states := matrix_multiply_constant(integrator_inputs, 2);
            e := matrix_add(e, integrator_states);
            intermediate_value := matrix_multiply_constant(integrator_inputs, runga_kutta_interval * rk_step_matrix[i]);
            integrator_states := matrix_add(y, intermediate_value);
            t := t + (runga_kutta_interval / 2);
          end;
        4:
          begin
            integrator_states := matrix_add(e, integrator_inputs);
            temp_value := runga_kutta_interval / 6;
            integrator_states := matrix_multiply_constant(integrator_states, temp_value);
            integrator_states := matrix_add(y, integrator_states);
          end;
      end; {case}
    end; {for}
  end;


{main program}
begin
  { open output file }
  rewrite(outfile, 'Simple Example');

  { put header in output file }
  writeln(outfile, 't , r , e , u , c , y');

  { initialize runge-kutta step constants }
  set_rk_step_matrix;

  { set t = 0 }
  t := 0;

  { initialize states }
  initialize_integrator_outputs;

  { run simulation }
  while not (t > run_time - (runga_kutta_interval / 2)) do
    begin
      compute_aux_variables(t);
      if (trunc(t / 0.01) mod 10) = 0 then
        writeln(outfile, t : 10 : 6, ' , ', system_input : 10 : 6, ' , ', ee : 10 : 6, ' , ', yy : 10 : 6);
    end;
end.
```

```
program Doyle_CAW_Continuous;


{*************************************}
{                                     }
{   R.J. Spangenberger    08 Aug 1991 }
{                                     }
{   4th Order Runge-Kutta Continuous Simulation }
{   of Doyle and Smith System with CAW  }
{                                     }
{*************************************}



{program constants}
 const

   max_integrators = 2;                        { number of integrators in system }
   limit = 1;                                  { limit value of saturation block }
   X_gain = 10;                                { gain of feedback in inner loop }


{program type definitions}
 type
   mat = array[1..max_integrators] of real;    { matrix of integrators }

{program variables}
 var
   rk_step_matrix: array[1..4] of real;        { four runge-kutta constants }
   integrator_inputs, x, integrator_states, y: mat;   { inputs, states and temporary variables }
   system_input: real;                         { command into system }
   t: real;                                    { time (seconds) }
   outfile: text;                              { output file for data }
   ee, yy, r, e2, e1, e3, x, u, c: real;       { auxiliary variables }


 function matrix_add (a, b: mat): mat;
 { add matrices a and b }
   var
    i: integer;
 begin
   for i := 1 to max_integrators do
     matrix_add[i] := a[i] + b[i];
 end;


 function matrix_multiply (a, b: mat): mat;
  { multiply matrices a and b }
   var
    i: integer;
 begin
   for i := 1 to max_integrators do
     matrix_multiply[i] := a[i] * b[i];
 end;



 function matrix_multiply_constant (a: mat; b: real): mat;
  { multiply matrix a by the value b }
   var
    i: integer;
 begin
   for i := 1 to max_integrators do
     matrix_multiply_constant[i] := a[i] * b;
 end;

 function matrix_divide_constant (a: mat; b: real): mat;
  { divide matrix a by the value b }
   var
    i: integer;
 begin
   for i := 1 to max_integrators do
     matrix_divide_constant[i] := a[i] / b;
 end;
```

```
procedure  set_rk_step_matrix;
{ initialize runge-kutta step constants }
begin
  rk_step_matrix[1]  :=  0.5;
  rk_step_matrix[2]  :=  0.5;
  rk_step_matrix[3]  :=  1;
  rk_step_matrix[4]  :=  2;
end;


procedure  initialize_integrator_outputs;
{ initialize the integrator states }
  var
    i: integer;
begin
  for i := 1 to  max_integrators  do
    integrator_states[i]  :=  0;
  ee := 0;
  e1  := 0;
  e2  := 0;
  e3  := 0;
  yy  :=  0;
  r  := 0;
  a  := 0;
  u  := 0;
  c  := 0;
end;


function  clamp  (input,  limit:  real):  real;
 {clamp output to limit}
  var
    output:  real;
begin
  if input > limit then
    output  := limit
  else if input < -limit then
    output  := -limit
  else
    output := input;
  clamp  := output;
end;


function  compute_system_input  (t:  real):  real;
{ compute input to system }
begin
  compute_system_input  := 1;                               { input = step at t=0 }
end;


procedure  run_simulation  (var  t:  real);
{ run simulation from t = 0 to t = run_time }
  var
    i:  integer;
    intermediate_value:  real;
    temp_value:  real;
begin
  for i := 1 to 4 do
    begin
      { compute system command }
      system_input  :=  compute_system_input(t);            { compute input to system }
      r  :=  system_input;

      {compute  auxiliary  variables}
      ee  := r - yy;
      e2  := ee;
      e1  := e2 - e3;
      u  :=  integrator_states[1];
      c  := clamp(u,  1);
      a := u - c;
      e3  := X_gain * a;
      yy  :=  (0.05  *  integrator_states[2])  +  (0.5  *  c);

      { compute  integrator  inputs  from  states }
      integrator_inputs[1]  :=  (2  *  e1)  -  (0.1  *  integrator_states[1]);
      integrator_inputs[2]  :=  c;
      { update states through each runge-kutta inner step }
      case i of
        1:
          begin
            e := integrator_inputs;
            y := integrator_states;
            intermediate_value := matrix_multiply_constant(integrator_inputs, runge_kutta_interval * rk_step_matrix[i]);
            integrator_states := matrix_add(y, intermediate_value);
            t := t + (runge_kutta_interval / 2);
          end;
        2:
          begin
            integrator_states := matrix_multiply_constant(integrator_inputs, 2);
            e := matrix_add(e, integrator_states);
            intermediate_value := matrix_multiply_constant(integrator_inputs, runge_kutta_interval * rk_step_matrix[i]);
            integrator_states := matrix_add(y, intermediate_value);
          end;
        3:
          begin
            integrator_states := matrix_multiply_constant(integrator_inputs, 2);
            e := matrix_add(e, integrator_states);
            intermediate_value := matrix_multiply_constant(integrator_inputs, runge_kutta_interval * rk_step_matrix[i]);
            integrator_states := matrix_add(y, intermediate_value);
            t := t + (runge_kutta_interval / 2);
          end;
        4:
          begin
            integrator_states := matrix_add(e, integrator_inputs);
            temp_value := runge_kutta_interval / 6;
            integrator_states := matrix_multiply_constant(integrator_states, temp_value);
            integrator_states := matrix_add(y, integrator_states);
          end;
      end; {case}
    end; {for}
  end;


{ main program }
begin
  { open output file }
  rewrite(outfile, 'Doyle CAW X=10');

   { put header in output file }
  writeln(outfile, 't , r , e , e2 , e3 , e1 , u , c , a , y');

   { initialize runge-kutta step constants }
  set_rk_step_matrix;

  { set t = 0 }
  t := 0;

  { initialize states }
  initialize_integrator_outputs;

  { run simulation }
  while not (t = run_time - (runge_kutta_interval / 2)) do
    begin
      run_simulation(t);
      if (trunc(t / 0.001) mod 100) = 0 then
        writeln(outfile, t : 10 : 6, ' , ', r : 10 : 6, ' , ', ee : 10 : 6, ' , ', yy : 10 : 6, ' , ');
    end;
end.
```

```
program Doyle_CAW_Complex_Continuous;


{***************************************}
{                                       }
{   R.J. Spangenberger    08 Aug 1991   }
{                                       }
{   4th Order Runge-Kutta Continuous Simulation    }
{   of Doyle and Smith System with Complex Compensator  }
{                                       }
{***************************************}



{program constants}
  const
    runge_kutta_interval = 0.001;              { time step for each runge-kutta interval }
    printing_interval = 1;                     { print out every printing_interval computed steps }
    run_time = 20;                             { run time of simulation in seconds }
    max_integrators = 4;                       { number of integrators in system }
    limit = 1;                                 { limit value of saturation block }
    X_gain = 25;                               { gain of feedback in inner loop }

{program type definitions}
  type
    mat = array[1..max_integrators] of real;   { matrix of integrators }

{program variables}
  var
    rk_step_matrix: array[1..4] of real;       { four runge-kutta constants }
    integrator_inputs, e, integrator_states, y: mat;  { inputs, states, and temporary variables }
    system_input: real;                        { command into system }
    t: real;                                   { time (seconds) }
    outfile: text;                             { output file for data }
    ee, yy, r, e2, e1, e3, a, u, c: real;      { auxiliary variables }



  function matrix_add (a, b: mat): mat;
   { add matrices a and b }
    var
      i: integer;
  begin
    for i := 1 to max_integrators do
      matrix_add[i] := a[i] + b[i];
  end;


  function matrix_multiply (a, b: mat): mat;
   { multiply matrices a and b }
    var
      i: integer;
  begin
    for i := 1 to max_integrators do
      matrix_multiply[i] := a[i] * b[i];
  end;



  function matrix_multiply_constant (a: mat; b: real): mat;
   { multiply matrix a by the value b }
    var
      i: integer;
  begin
    for i := 1 to max_integrators do
      matrix_multiply_constant[i] := a[i] * b;
  end;

  function matrix_divide_constant (a: mat; b: real): mat;
   { divide matrix a by the value b }
    var
      i: integer;
  begin
    for i := 1 to max_integrators do
      matrix_divide_constant[i] := a[i] / b;
  end;
```

```
procedure  set_rk_step_matrix;
{ initialize runge-kutta step constants }
begin
  rk_step_matrix[1]  :=  0.5;
  rk_step_matrix[2]  :=  0.5;
  rk_step_matrix[3]  :=  1;
  rk_step_matrix[4]  :=  2;
end;


procedure  initialize_integrator_outputs;
{ initialize the integrator states }
  var
    i:  integer;
begin
  for i := 1 to  max_integrators  do
    integrator_states[i]  :=  0;
  ee := 0;
  e1  := 0;
  e2 := 0;
  e3 := 0;
  yy  := 0;
  r := 0;
  a := 0;
  u := 0;
  c := 0;
end;


function  clamp  (input,  limit:  real):  real;
 {clamp output to limit}
  var
    output:  real;
begin
  if input > limit then
    output  :=  limit
  else if  input  < -limit then
    output  :=  -limit
  else
    output  := input;
  clamp  := output;
end;


function  compute_system_input  (t:  real):  real;
{ compute input to system }
begin
  compute_system_input  :=  1;
end;


procedure  run_simulation  (var  t:  real);
{ run simulation from t = 0 to t = run_time }
  var
    i:  integer;
    intermediate_value:  mat;
    temp_value:  real;
begin
  for i := 1 to 4 do
    begin

      { compute system command }
      system_input  :=  compute_system_input(t);

      {compute  auxiliary  variables}
      r  :=  system_input;
      ee := r - yy;
      e2 := ee;
      e1  := e2 - e3;
      u  :=  integrator_states[3];
      c := clamp(u,  1);
      a := u - c;
      e3 := X_gain * a;
      yy  :=  (0.05  *  integrator_states[4])  +  (0.5  *  c);


      {compute  integrator  inputs  from  states}
      integrator_inputs[1]  :=  (800  *  e1)  +  (-40.1  *  integrator_states[1])  +  (-404  *  integrator_states[2]);
      integrator_inputs[1] := integrator_inputs[1] + (-40 * integrator_states[3]);
      integrator_inputs[2] := integrator_states[1];
      integrator_inputs[3] := integrator_states[2];

      integrator_inputs[4] := c;

      { update states through each runge-kutta minor step }
      case i of
        1:
          begin
            e := integrator_inputs;
            y := integrator_states;
            intermediate_value := matrix_multiply_constant(integrator_inputs, runga_kutta_interval * rk_step_matrix[i]);
            integrator_states := matrix_add(y, intermediate_value);
            t := t + (runga_kutta_interval / 2);
          end;
        2:
          begin
            integrator_states := matrix_multiply_constant(integrator_inputs, 2);
            e := matrix_add(e, integrator_states);
            intermediate_value := matrix_multiply_constant(integrator_inputs, runga_kutta_interval * rk_step_matrix[i]);
            integrator_states := matrix_add(y, intermediate_value);
          end;
        3:
          begin
            integrator_states := matrix_multiply_constant(integrator_inputs, 2);
            e := matrix_add(e, integrator_states);
            intermediate_value := matrix_multiply_constant(integrator_inputs, runga_kutta_interval * rk_step_matrix[i]);
            integrator_states := matrix_add(y, intermediate_value);
            t := t + (runga_kutta_interval / 2);
          end;
        4:
          begin
            integrator_states := matrix_add(e, integrator_inputs);
            temp_value := runga_kutta_interval / 6;
            integrator_states := matrix_multiply_constant(integrator_states, temp_value);
            integrator_states := matrix_add(y, integrator_states);
          end;
      end; {case}
    end; {for}
  end;


{main program}
begin
  { open output file }
  rewrite(outfile, 'Complex Compensator');

  { put header in output file }
  writeln(outfile, 't , r , e , u , c , y');

  { initialize runge-kutta step constants }
  set_rk_step_matrix;

  { set t = 0 }
  t := 0;

  { initialize the system }
  initialize_integrator_outputs;

  { run simulation }
  while not (t > run_time - (runga_kutta_interval / 2)) do
    begin
      compute_aux_variables(t);
      if (trunc(t / 0.001) mod 100) = 0 then
        writeln(outfile, t : 10 : 6, ' , ', system_input : 10 : 6, ' , ', ee : 10 : 6, ' , ', yy : 10 : 6);
    end;
end.
```

```
program Glattfelder_OS_Continuous;



{ * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * }
{                                                                             }
{   R.J. Spangenberger     08 Aug 1991                                        }
{                                                                             }
{   4th Order Runge-Kutta Continuous Simulation                               }
{   of Glattfelder and Schaufelberger OS System                               }
{                                                                             }
{ * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * }



{program constants}
 const
   runga_kutta_interval = 0.001;                { time step for each runge-kutta interval }
   printing_interval = 1;                       { print out every printing_interval computed state }
   run_time = 20;                               { run time of simulation in seconds }
   max_integrators = 2;                         { number of integrators in system }
   limit = 1;                                   { limit value of saturation block }
   K2_gain = 10;                                { gain of feedback in inner loop }

   STANDARD_LIMIT = 1.0;                        { value for uhi, ulo inputs }
   yo = 0;                                      { initial value of selected states }


{program type definitions}
 type
   mat = array[1..max_integrators] of real;     { matrix of integrators }

{program variables}
 var
   rk_step_matrix: array[1..4] of real;         { four runge-kutta constants }
   integrator_inputs, x, integrator_states, y: mat;  { inputs, states and temporary variables }
   system_input: real;                          { command into system }
   t: real;                                     { time (seconds) }
   outfile: text;                               { output file for data }
   r, e, e1, up, ul, temp, c, ll, yy: real;     { auxiliary variables }
   uhi, ulo: real;                              { uhi, ulo variables }


 function max (a, b: real): real;
 { return maximum of a or b }
 begin
   if a >= b then
     max := a
   else
     max := b;
 end;



 function min (a, b: real): real;
 { return minimum of a or b }
 begin
   if (a <= b) then
     min := a
   else
     min := b;
 end;




 function matrix_add (a, b: mat): mat;
 { add matrices a and b }
  var
    i: integer;
 begin
   for i := 1 to max_integrators do
     matrix_add[i] := a[i] + b[i];
 end;


 function matrix_multiply (a, b: mat): mat;
 { multiply matrices a and b }
  var
    i:  integer;
begin
  for i := 1 to  max_integrators  do
    matrix_multiply[i]  :=  a[i]  *  b[i];
end;



function  matrix_multiply_constant  (a:  mat;  b:  real):  mat;
 { multiply matrix a by the value b }
  var
    i:  integer;
begin
  for i := 1 to  max_integrators  do
    matrix_multiply_constant[i]  :=  a[i]  *  b;
end;


function  matrix_divide_constant  (a:  mat;  b:  real):  mat;
 { divide matrix a by the value b }
  var
    i:  integer;
begin
  for i := 1 to  max_integrators  do
    matrix_divide_constant[i]  :=  a[i]  /  b;
end;


procedure  set_rk_step_matrix;
{ initialize runge-kutta step constants }
begin
  rk_step_matrix[1]  :=  0.5;
  rk_step_matrix[2]  :=  0.5;
  rk_step_matrix[3]  :=  1;
  rk_step_matrix[4]  :=  2;
end;


procedure  initialize_integrator_outputs;
{ initialize the integrator states }
  var
    i:  integer;
begin
  integrator_states[1]  :=  0;
  integrator_states[2]  :=  -yo;
  r := 0;
  ee := 0;
  e1  := 0;
  up := 0;
  ui  := yo;
  temp  := 0;
  u := yo;
  ii := 0;
  yy  :=  -yo;
end;


function  clamp  (input,  limit:  real):  real;
 {clamp output to limit}
  var
    output:  real;
begin
  if input > limit then
    output  :=  limit
  else if input < -limit then
    output  :=  -limit
  else
    output  :=  input;
  clamp  :=  output;
end;


function  compute_system_input  (t:  real):  real;
{ compute input to system }
begin
  compute_system_input  :=  1;
end;
```

```
procedure run_simulation (var t: real);
{ run simulation from t = 0 to t = run_time }
  var
    i: integer;
    intermediate_value: mat;
    temp_value: real;
begin
  uhi := STANDARD_LIMIT;                                         { set uhi and ulo }
  ulo := -STANDARD_LIMIT;
  for i := 1 to 4 do
    begin
      system_input := compute_system_input(t);                   { compute input to system }
      r := system_input;

      { compute auxiliary variables }
      ee := r - yy;
      temp := max(((-c + ulo) * K2_gain), ee);
      e1 := min(((-c + uhi) * K2_gain), temp);
      c := integrator_states[1];
      ii := clamp(c, 1);
      yy := (0.05 * integrator_states[2]) + (0.5 * ii);

      { compute integrator inputs }
      integrator_inputs[1] := (2 * e1) - (0.1 * integrator_states[1]);
      integrator_inputs[2] := ii;

       { update states through each runge-kutta minor step }
      case i of
        1:
          begin
            e := integrator_inputs;
            y := integrator_states;
            intermediate_value := matrix_multiply_constant(integrator_inputs, runga_kutta_interval * rk_step_matrix[i]);
            integrator_states := matrix_add(y, intermediate_value);
            t := t + (runga_kutta_interval / 2);
          end;
        2:
          begin
            integrator_states := matrix_multiply_constant(integrator_inputs, 2);
            e := matrix_add(e, integrator_states);
            intermediate_value := matrix_multiply_constant(integrator_inputs, runga_kutta_interval * rk_step_matrix[i]);
            integrator_states := matrix_add(y, intermediate_value);
          end;
        3:
          begin
            integrator_states := matrix_multiply_constant(integrator_inputs, 2);
            e := matrix_add(e, integrator_states);
            intermediate_value := matrix_multiply_constant(integrator_inputs, runga_kutta_interval * rk_step_matrix[i]);
            integrator_states := matrix_add(y, intermediate_value);
            t := t + (runga_kutta_interval / 2);
          end;
        4:
          begin
            integrator_states := matrix_add(e, integrator_inputs);
            temp_value := runga_kutta_interval / 6;
            integrator_states := matrix_multiply_constant(integrator_states, temp_value);
            integrator_states := matrix_add(y, integrator_states);
          end;
      end; {case}
    end; {for}
end;


{main program}
begin
  { open output file }
  rewrite(outfile, '[illegible] CAW K2=10');

  { put header in output file }
  writeln(outfile, 't , r , e , e1 , c , i , y ');

  { initialize runge-kutta step constants }
  set_rk_step_matrix;

  { set t = 0 }
  t := 0;
  { initialize [illegible] }
  initialize_integrator_outputs;

  { run simulation }
  while not (t > run_time - (runga_kutta_interval / 2)) do
    begin
      compute_aux_variables(t);
      if (trunc(t / 0.001) mod 100) = 0 then
        writeln(outfile, t : 10 : 6, ' , ', r : 10 : 6, ' , ', ee : 10 : 6, ' , ', yy : 10 : 6);
    end;
end.
```

<u>DISCRETE-TIME SIMULATIONS</u>

<u>PROGRAM LISTINGS</u>

```
program Doyle_CAW_Discrete (input, output);


{*************************************}
{                                     }
{  R.J. Spangenberger    08 Aug 1991  }
{                                     }
{  Discrete Time Simulation           }
{  of Doyle and Smith System with CAW }
{                                     }
{*************************************}


 const
  MAXX = 20000;                          { max number of time steps }
  INFINITY = 9999999;                    { constant for infinity }
  TO_FILE = 1;                           { 2=write to nowhere, 1=write to file,  0=write to screen }

 type
  real_type = array[0..MAXX] of real;    { states and auxiliary variables }
  ptr_type = ^real_type;                 { pointers to states and auxiliary variables }

 var
  r, y, e, e1, e2, e3, a, c, i: ptr_type;  { system states and auxiliary variables }
  limit: real;                             { limit value of saturation block }
  X_gain: real;                            { gain of inner feedback loop }

  start_time: real;                        { simulation start time }
  stop_time: real;                         { simulation stop time }
  T: real;                                 { sample time }
  number_of_iterations: integer;           { total number of iterations }

  outfile: text;                           { output file for data }
  filename: string[80];                    { output file name }



 procedure Initialize;
  { Initialize Variables }
   var
    k: integer;
 begin
  {open file}
  if TO_FILE = 1 then
   begin
    rewrite(outfile, filename);
    writeln(outfile, 'k,kT, r, e , e2,e3,e1 , c ,i , a , y ');
   end
  else if TO_FILE = 0 then
   writeln('k,kT, r, e , e2,e3,e1 , c , i , a , y ')
  else
   begin
    {nothing}
   end;

   {new dynamic variables}
  new(r);
  new(e);
  new(e1);
  new(e2);
  new(e3);
  new(c);
  new(i);
  new(a);
  new(y);

   {initialize variables}
  for k := 0 to MAXX do
   begin
    r^[k] := 1;
    e^[k] := 0;
    e1^[k] := 0;
    e2^[k] := 0;
    e3^[k] := 0;
    c^[k] := 0;
    i^[k] := 0;
    a^[k] := 0;
    y^[k] := 0;
    end;

   {set saturation limit}
  limit := 1.0;

   {set stop time}
  stop_time := 20;

   {compute number of iterations}
  number_of_iterations := round(stop_time / T);

   {if number of iterations exceeds allocation, indicate error}
  if ((stop_time / MAXX) ≥ T) then
    begin
      sysbeep(200);
      writeln('ERROR ... TOO MANY ITERATIONS');
    end;
end;



procedure ClearSpace;
 {dispose of dynamic variables}
begin
  dispose(r);
  dispose(e);
  dispose(e1);
  dispose(e2);
  dispose(e3);
  dispose(c);
  dispose(l);
  dispose(a);
  dispose(y);
end;



procedure Run_Simulation;
{Run simulation from 1 to MAXX iterations}
  var
    k: integer;
begin
   {compute iterations}
  for k := 1 to number_of_iterations do
    begin
      e^[k] := r^[k] - y^[k - 1];
      e2^[k] := e^[k];
      e1^[k] := e2^[k] - e3^[k - 1];
      c^[k] := ((2 / (0.1 + (2 / T))) * (e1^[k])) + ((2 / (0.1 + (2 / T))) * (e1^[k - 1]));
      c^[k] := c^[k] - ((0.1 - (2 / T)) / (0.1 + (2 / T)) * c^[k - 1]);

      if c^[k] ≥ limit then
        l^[k] := limit
      else if c^[k] ≤ -limit then
        l^[k] := -limit
      else
        l^[k] := c^[k];

      a^[k] := c^[k] - l^[k];
      e3^[k] := a^[k] * X_gain;

       {Step Invariant Plant Equation}
      y^[k] := (0.5 * l^[k]) + (((0.05 * T) - 0.5) * l^[k - 1]) + y^[k - 1];

      if TO_FILE = 1 then
        writeln(outfile, k : 5, ' , ', k * T : 10 : 6, ' , ', r^[k] : 10 : 6, ' , ', e^[k] : 10 : 6, ' , ', y^[k] : 10 : 6)
      else if TO_FILE = 0 then
        writeln(k : 5, k * T : 10 : 6, r^[k] : 10 : 6, e^[k] : 10 : 6, y^[k] : 10 : 6)
      else
        begin
         {nothing}
        end;
    end;
end;


begin
 { set xgain value }
 X_gain := 10;
 { set sample time }
 T := 0.001;

 { initialize output file name }
 filename := 'Doyle CAW X=10';

 { set initial conditions and initialize variables }
 Initialize;

 { run simulation }
 Run_Simulation;

 { close output file }
 close(outfile);

 { dispose of dynamic storage space }
 ClearSpace;
end.
```

```
program Glattfelder_OS_Discrete (input, output);

{*****************************************}
{                                         }
{   R.J. Spangenberger     08 Aug 1991    }
{                                         }
{   Discrete Time Simulation              }
{   of Glattfelder and Schaufelberger System with CAW  }
{                                         }
{*****************************************}

 const
   MAXX = 2000;                                { max number of time steps }
   INFINITY = [illegible];                     { constant for infinity }
   TO_FILE = 1;                                { 2=write to nowhere, 1=write to file,  0=write to screen }
   STANDARD_LIMIT = 1.0;                       { value for uhi, ulo inputs }
   yo = 0;                                     { initial value of selected states }


 type
   real_type = array[0..MAXX] of real;         { states and auxiliary variables }
   ptr_type = ^real_type;                      { pointers to states and auxiliary variables }


 var
   r, e, e1, ei, e2, c, i, y: ptr_type;        { system states and auxiliary variables }
   eh, el, s: ptr_type;
   k: integer;

   limit: real;                                { limit value of saturation block }
   uhi, ulo: real;                             { uhi, ulo variables }
   K2_gain: real;                              { gain of feedback in inner loop}

   start_time: real;                           | simulation start time }
   stop_time: real;                            { simulation stop time }
   T: real;                                    { sample time }
   number_of_iterations: integer;              { total number of iterations }

   outfile: text;                              { output file for data }
   filename: string[80];                       | output file name }



 function max (a, b: real): real;
 { return the max between a and b }
 begin
   if a >= b then
     max := a
   else
     max := b;
 end;



 function min (a, b: real): real;
  { return the min between a and b }
 begin
   if (a <= b) then
     min := a
   else
     min := b;
 end;



 procedure Initialize;
 { Initialize Variables }
   var
     k: integer;
 begin
   {open file}
   if TO_FILE = 1 then
     begin
       rewrite(outfile,  filename);
       writeln(outfile, 'k ,   kT  ,  r ,  e , e1, eh, et , el , s ,  ei ,  c , i , e2 ,  y');
     end
   else if TO_FILE = 0 then
     writeln(outfile, 'k ,   kT  ,  r ,  e , e1, eh, et , el , s ,  ei ,  c , i , e2 ,  y')
  else
    begin
     {nothing}
    end;

   { new dynamic variables }
  new(r);
  new(e);
  new(e1);
  new(ei);
  new(e2);
  new(c);
  new(i);
  new(y);

  new(eh);
  new(el);
  new(et);


   { initialize variables }
  for k := 0 to MAXX do
    r^[k] := 1;
  e^[0] := 0;
  e1^[0] := 0;
  et^[0] := 0;
  eh^[0] := 0;
  el^[0] := 0;
  ei^[0] := yo;
  e2^[0] := 0;
  c^[0] := yo;
  i^[0] := 0;
  y^[0] := -yo;

   { set saturation values }
  limit := STANDARD_LIMIT;
  uhi := STANDARD_LIMIT;
  ulo := -STANDARD_LIMIT;

   {sample time}
  T := 0.01;

   {set stop time}
  stop_time := 10;

   {compute number of iterations}
  number_of_iterations := round(stop_time / T);

    {if number of iterations exceeds allocation, indicate error}
  if ((stop_time / MAXX) > T) then
    begin
      sysbeep(200);
      writeln('ERROR ... TOO MANY ITERATIONS');
    end;
end;



procedure ClearSpace;
 {dispose of dynamic variables}
begin
  dispose(r);
  dispose(e);
  dispose(e1);
  dispose(ei);
  dispose(e2);
  dispose(c);
  dispose(i);
  dispose(y);

  dispose(eh);
  dispose(el);
  dispose(et);
end;



procedure Run_Simulation;
```

```
 { Run simulation from 1 to MAXX iterations }
   var
     k: integer;
 begin
 {compute iterations}
   for k := 1 to MAXX do
     begin

       e^[k] := r^[k] - y^[k - 1];
       eh^[k] := (uhi - c^[k - 1]) * K2_gain;
       el^[k] := (ulo - c^[k - 1]) * K2_gain;
       et^[k] := max(e^[k], el^[k]);
       e1^[k] := min(et^[k], eh^[k]);
       if (e1^[k] = eh^[k]) then
         [illegible] := 1
       else if (e1^[k] = el^[k]) then
         [illegible] := 1
       else
         [illegible] := 0;

       ei^[k] := ((2 / (0.1 + (2 / T))) * (e1^[k])) + ((2 / (0.1 + (2 / T))) * (e1^[k - 1]));
       ei^[k] := ei^[k] - ((0.1 - (2 / T)) / (0.1 + (2 / T)) * ei^[k - 1]);

       c^[k] := ei^[k];

        {Limit}
       if c^[k] > limit then
         l^[k] := limit
       else if c^[k] < -limit then
         l^[k] := -limit
       else
         l^[k] := c^[k];
       {End Limit}

       e2^[k] := c^[k] - l^[k];

       y^[k] := (0.5 * l^[k]) + (((0.05 * T) - 0.5) * l^[k - 1]) + y^[k - 1];


       if TO_FILE = 1 then
         writeln(outfile, k : 5, ' , ', k * T : 10 : 6, ' , ', r^[k] : 10 : 6, ' , ', e^[k] : 10 : 6, ' , ', y^[k] : 10 : 6)
       else if TO_FILE = [illegible] then
         writeln(k : 5, ' , ', k * T : 10 : 6, ' , ', r^[k] : 10 : 6, ' , ', e^[k] : 10 : 6, ' , ', y^[k] : 10 : 6)
       else
         begin
          {nothing}
         end;
     end;
 end;


begin
 { set K2 gain value }
  K2_gain := 10;

  { set sample time }
  T := 0.001;

  { initialize output file name }
  filename := 'Glattfelder OS X=10';

  { set initial conditions and initialize variables }
  initialize;

  { run simulation }
  Run_Simulation;

  { close output file }
  close(outfile);

  { dispose of dynamic storage space }
  ClearSpace;

end.
```

```
program Track_Rate_Simulation (input, output);

{*****************************************}
{                                         }
{  R.J. Spangenberger    06 Aug 1991      }
{                                         }
{  Discrete Time Simulation               }
{  of Track Rate Servo - Traverse Axis with CAW }
{                                         }
{*****************************************}


 const
   MAXDATA = 1000;                        { maximum number of iterations }
   INFINITY = 9999999;                    { infinity constant }
   SAMPLE_RATE = 1000;                    { simulation sample rate }
   SCREEN_OR_FILE = 1;                    { 0 = write to screen, 1 = write to file }
   NUM_BETWEEN_PRINTS = 10;               { number of iterations between output of data }
   X_gain = 10;                           { gain of inner feedback loop }

 type
   data_type = array[0..MAXDATA] of real; { states and auxiliary variables }
   ptr_type = ^data_type;                 { pointer to states and auxiliary variables }

 var
   k: integer;                            { iteration variable }
   sample_period: real;                   { sample period }

    {transfer function blocks}
   KCC_IN, KCC_OUT: ptr_type;
   GF_IN, GF_OUT: ptr_type;
   GI_IN, GI_OUT: ptr_type;
   GC_IN, GC_OUT: ptr_type;
   GPA_IN, GPA_OUT: ptr_type;
   GMRATE_IN, GMRATE_OUT: ptr_type;
   GMPOS_IN, GMPOS_OUT: ptr_type;
   GG_IN, GG_OUT: ptr_type;


    {data file}
   outfile:  text;



 procedure initialize;
 { initialize states and auxiliary variables }
   var
     i: integer;
 begin
    {set sample period}
   sample_period := 1 / SAMPLE_RATE;
    {new variables}
   new(KCC_IN);
   new(KCC_OUT);
   new(GF_IN);
   new(GF_OUT);
   new(GI_IN);
   new(GI_OUT);
   new(GC_IN);
   new(GC_OUT);
   new(GPA_IN);
   new(GPA_OUT);
   new(GMRATE_IN);
   new(GMRATE_OUT);
   new(GMPOS_IN);
   new(GMPOS_OUT);
   new(GG_IN);
   new(GG_OUT);
    {set system initial conditions}
   for i := 1 to MAXDATA do
     begin
       KCC_IN^[i] := 0.1;
       KCC_OUT^[i] := 0;
       GF_IN^[i] := 0;
       GF_OUT^[i] := 0;
       GI_IN^[i] := 0;
       GI_OUT^[i] := 0;
       GC_IN^[i] := 0;
       GC_OUT^[i] := 0;
```

```
      GPA_IN^[i] := 0;
      GPA_OUT^[i] := 0;
      GMRATE_IN^[i] := 0;
      GMRATE_OUT^[i] := 0;
      GMPOS_IN^[i] := 0;
      GMPOS_OUT^[i] := 0;
      GG_IN^[i] := 0;
      GG_OUT^[i] := 0;
    end;
   {open output file}
  rewrite(outfile, 'Track Rate Servo CAW  X = 52');
end;



procedure KDG (var x, y: ptr_type; k: integer);
{compute output of gain block}
  const
    KK = 5.1177;
begin
  y^[k] := x^[k] * KK;
end;



procedure GI (var x, y: ptr_type; k: integer);
{compute output of  integrator}
  var
    gain, a: real;
begin
  gain := ((10.5) * (10.74 * sample_period + 2)) / (10.74 * 2);
  a := (10.74 * sample_period - 2) / (10.74 * sample_period + 2);
  y^[k] := (gain * x^[k]) + (gain * a * x^[k - 1]) + (y^[k - 1]);
end;



procedure GC (var x, y: ptr_type; k: integer);
{compute output of compensator}
  var
    gain, a: real;
begin
  gain := (34.01 * 1024.59 * sample_period) / (1024.59 * sample_period + 2);
  a := (1024.59 * sample_period - 2) / (1024.59 * sample_period + 2);
  y^[k] := (gain * x^[k]) + (gain * x^[k - 1]) - (a * y^[k - 1]);
end;



procedure GPA (var x, y: ptr_type; k: integer);
{compute output of power amplifier}
  const
    gain = 1;
    limit = 10;
begin
  y^[k] := gain * x^[k];
  if y^[k] > limit then
    y^[k] := limit
  else if y^[k] < -limit then
    y^[k] := -limit;
end;



procedure GMRATE (var x, y: ptr_type; k: integer);
{compute output rate of  plant (motor) using step invariant transform}
  var
    A1, B1, AA, BB, CC, DD, EE, a, b, c, d, e, f: real;
begin
  a := [illegible];
  b := 1.181;
  c := 98.91;

  A1 := [illegible];
  B1 := 3.41047e-3;

  AA := a - (A1) + (B1);
  BB := (-a * exp(-c * sample_period)) - (a * exp(-b * sample_period)) + (A1);
  BB := BB + (A1 * exp(-c * sample_period)) - (B1) - (B1 * exp(-b * sample_period));
  CC := (a * exp(-b * sample_period - c * sample_period)) - (A1 * exp(-c * sample_period));
  CC := CC + (B1 * exp(-b * sample_period));
  DD := (-exp(-c * sample_period) - exp(-b * sample_period));
  EE := (exp(-b * sample_period - c * sample_period));

  y^[k] := (AA * x^[k]) + (BB * x^[k - 1]) + (CC * x^[k - 2]) - (DD * y^[k - 1]) - (EE * y^[k - 2]);
end;




procedure GMPOS (var x, y: ptr_type; k: integer);
{compute output position of  plant (motor)}
begin
  y^[k] := (x^[k]) + (y^[k - 1]);
end;



procedure GH (var x, y: ptr_type; k: integer);
{compute output of  gyro}
  const

    a = 6.406611384e12;
    b = 766.42564;
    c = 681377.11;
    d = 2668.2030;
    e = 3048381.8;

  var
    AA, BB, CC, DD, EE, FF, GG, HH, II: real;
    DZ5, DZ4, DZ3, DZ2, DZ1, DZ0: real;
    NZ5, NZ4, NZ3, NZ2, NZ1, NZ0: real;
    temp1,  temp2:  real;
begin

  AA := 4 + (2 * b * sample_period) + (sample_period * sample_period * c);
  BB := -8 + (2 * sample_period * sample_period * c);
  CC := 4 - (2 * b * sample_period) + (sample_period * sample_period * c);
  DD := 4 + (2 * d * sample_period) + (sample_period * sample_period * e);
  EE := -8 + (2 * sample_period * sample_period * e);
  FF := 4 - (2 * d * sample_period) + (sample_period * sample_period * e);
  GG := sample_period * sample_period * sample_period * sample_period;

  DZ4 := (AA * DD);
  DZ3 := (AA * EE) + (BB * DD);
  DZ2 := (AA * FF) + (BB * EE) + (CC * DD);
  DZ1 := (BB * FF) + (CC * EE);
  DZ0 := (CC * FF);

  NZ4 := (a * GG);
  NZ3 := 4 * (a * GG);
  NZ2 := 6 * (a * GG);
  NZ1 := 4 * (a * GG);
  NZ0 := (a * GG);

  temp1 := (NZ4 / DZ5 * x^[k]) + (NZ3 / DZ5 * x^[k - 1]) + (NZ2 / DZ5 * x^[k - 2]);
  temp1 := temp1 + (NZ1 / DZ5 * x^[k - 3]) + (NZ0 / DZ5 * x^[k - 4]);
  temp2 := -(DZ3 / DZ5 * y^[k - 1]) - (DZ2 / DZ5 * y^[k - 2]);
  temp2 := temp2 - (DZ1 / DZ5 * y^[k - 3]) - (DZ0 / DZ5 * y^[k - 4]);
  y^[k] := temp1 + temp2;
  y^[k] := x^[k] * 3.08;
end;



procedure GF (var x, y: ptr_type; k: integer);
{compute output of gyro feedback summer}
  var
    gain, a: real;
begin
  gain := (245.68 * sample_period) / (148 * sample_period + 2);
  a := (148 * sample_period - 2) / (148 * sample_period + 2);
  y^[k] := (gain * x^[k]) + (gain * x^[k - 1]) - (a * y^[k - 1]);
end;
```

```
  procedure Run_Simulation;
  { Run simulation from 1 to MAXDATA iterations }
    var
      k, i: integer;
      temp1, delta: ptr_type;
  begin
    new(temp1);
    new(delta);
    for i := 1 to MAXDATA do
      begin
        temp1^[i] := 0;
        delta^[i] := 0;
      end;
    for k := 5 to MAXDATA do
      begin

        { compute kcc block }
        KCC(KCC_IN, KCC_OUT, k);

        { output of summer }
        temp1^[k] := KCC_OUT^[k] - GF_OUT^[k - 1];

        { output of CAW summer }
        temp1^[k] := temp1^[k] - delta^[k - 1];

        { output of integrator/filter }
        GI(temp1, GI_OUT, k);

        { output of compensator }
        GC(GI_OUT, GC_OUT, k);


        { output of motor power amplifier }
        GPA(GC_OUT, GPA_OUT, k);

        { compute CAW difference }
        delta^[k] := GC_OUT^[k] - GPA_OUT^[k];

        { compute CAW feedback signal }
        delta^[k] := delta^[k] * X_gain;

        { compute motor rate }
        GMRATE(GPA_OUT, GMRATE_OUT, k);

        { compute motor position }
        GMPOS(GMRATE_OUT, GMPOS_OUT, k);

        { compute gyro output }
        GG(GMRATE_OUT, GG_OUT, k);

        { compute filter output }
        GF(GG_OUT, GF_OUT, k);

        { write data to output file }
        if SCREEN_OR_FILE = 1 then
          begin
            if (k mod NUM_BETWEEN_PRINTS) = 0 then
              writeln(outfile, k : 10, ' , ', ((k - 5) * sample_period) : 10 : 5, ' , ', GMRATE_OUT^[k] : 10 : 5);
          end
        else
          writeln(k : 10, ((k - 5) * sample_period) : 10 : 5, '    ', GI_OUT^[k] : 10 : 5, '    ', GC_OUT^[k] : 10 : 5);
      end;
  end;


begin
  { initalize system }
  writeln('Initializing . . . Please Wait');
  Initialize;

  { run simulation }
  writeln('Running . . . Please Wait');
  Run_Simulation;
end.
```

BIBLIOGRAPHY

Books

Franklin, G.F., J.D. Powell and M.L. Workman, *Digital Control of Dynamic Systems - Second Addition*, Addison-Wesley Publishing Company, Reading MA, 1990.

James, H.M., *Theory of Servo-Mechanism*, New York, NY: McGraw-Hill, 1947.

Katz, P., *Digital Control Using Microprocessors*, Englewood Cliffs, NJ: Prentice-Hall, 1981.

Kuo, B.C., *Automatic Control Systems*, Englewood Cliffs, NJ: Prentice-Hall, 1982.

Kuo, B.C., *Discrete-Data Control Systems*, Champaign, IL: Science Tech, 1947.

Lindorff, D.P., *Theory of Sampled-Data Control Systems*, New York, NY: John Wiley & Sons, Inc., 1965.

Mack, D.R., *The General Electric A Course - Engineering Analysis*, General Electric Company, Bridgeport, CT, 1983.

McDonald, A.C. and H. Lowe, *Feedback and Control Systems*, Reston, VA: Prentice-Hall, 1981.

Murphy, G.J., *Control Engineering*, Cambridge, MA: Boston Technical Publishers, Inc., 1965.

Ogata, K., *Modern Control Engineering*, Englewood Cliffs, NJ: Prentice-Hall, 1970.

Ogata, K. *Discrete-Time Control Systems*, Englewood, NJ: Prentice-Hall, 1987.

Siebert, W.M., *Circuits, Signals and Systems Lecture Notes*, Cambridge, MA: Massachusetts Institute of Technology, Dept. of Electrical Engineering and Computer Science, 1984.

Periodicals

Anderson, M.J. and W.J. Grantham, "Lyapunov Optimal Feedback Control of a Nonlinear Inverted Pendulum", *Transactions of the ASME*, Vol. 111, December, 1989, pp 554-558.

Buckley, P.S., "Designing Override and Feedforward Controls - I",*Control Engineering*, Vol. 18, No. 8, 1971, pp 48-51.

Buckley, P.S., "Designing Override and Feedforward Controls - II", *Control Engineering*, Vol. 18, No. 9, 1971, pp 82-85.

Doyle, J.C., R.S Smith and D.F Enns, "Control of Plants with Input Saturation Nonlinearities", *Proceedings of the 1987 American Control Conference*, June 1987, pp. 1034-1039.

Glattfelder, A.H. and W. Schaufelberger, "Stability Analysis of Single Loop Control System with Saturation and Antireset-Windup Circuits", *IEEE Transactions on Automatic Control*, Vol AC-28, No. 12, December, 1983, pp 1074-1081.

Glattfelder, A.H., W. Schaufelberger and J. Todtli, "Diskrete Proportional-Integral-Differential Refler mit Anti-Windup Massnahmen", *SGA Bulletin*, 1983, No. 1, pp. 12-22; No. 2, pp. 19-28.

Hermes, H. "Asymptotically Stabilizing Feedback Controls and the Nonlinear Regulator Problem", *SIAM Journal of Control and Optimization*, Vol. 29, No. 1, January, 1991, pp. 185-196.

Henson, M.A. and D.E. Seborg, "Input-Output Linearization of General Nonlinear Processes", *AIChE Journal*, Vol. 36, No. 11, November 1990, pp. 1753-1757.

Khambanond, T., A. Palazoglu and J.A Romagnoli, "The Stability Analysis of Nonlinear Feedback Systems", *AIChE Journal*, Vol. 36, No. 1, January 1990, pp. 66-74.

Nett, C.N. and J.A. Polley, "Nonlinear Digital Controllers", *Proceedings of the American Control Conference*, March, 1988, pp. 1671-1678.

Slivinsky, C. and J. Borninski, "Control System Compensation and Implementation with the TMS32010", Texas Instruments, Dallas, TX, 1990.

Tustin, A., "A Method of Analyzing the Behavior of Linear Systems in Terms of Time   Series", *JIEE*, Vol. 94, Part IIA, 1947, pp. 130-142.

Zames, G., "On the Input-Output Stability of Time-Varying Nonlinear Feedback Systems, Part II: Conditions Involving Circles in the Frequency Plane and Sector Nonlinearities", *IEEE Transactions on Automatic Control*, Vol AC-11, 1966, pp. 465-476.