NORTHERN ILLINOIS UNIVERSITY

PD, PI, AND PID CONTROLLER DESIGNING SOFTWARE USING MATLAB

A Thesis Submitted to the
University Honors Program
In Partial Fulfillment of the
Requirements of the Baccalaureate Degree
With Upper Division Honors

Department of Electrical Engineering

by Bradley S. Mathis

DeKalb, Illinois

14 May, 1994

Student Name: Bradley S. Mathis

Approved By: James P. Bohs

Department of: Electrical Engineering

Date: 5-5-94

HONORS THESIS ABSTRACT
THESIS SUBMISSION FORM

AUTHOR: Bradley S. Mathis

THESIS TITLE: PD, PI, and PID Controller Designing Software Using MATLAB

ADVISOR: Dr. James Bobis ADVISOR'S DEPT: ELE

DISCIPLINE: Control Systems YEAR: 1994

PAGE LENGTH: 66 BIBLIOGRAPHY: Yes ILLUSTRATED: Yes

PUBLISHED: No

COPIES AVAILABLE: Hard Copy

ABSTRACT:

The main purpose of this project is the design and programming of software which will design PD, PI and PID controllers for a plant transfer function, given a user specified desired output phase margin and error constant as design parameters. The controllers to be designed are restricted to controllers in series with the controlled process, with unity feedback. Frequency domain analysis techniques are used as the basis for the controller design algorithm, the methods of which are derived from the reference texts listed, and outlined in this document. The project software is written in MATLAB, to take advantage of many of its existing controls-based frequency domain analysis functions, and to provide a medium for full exploitation and analysis of the software's design results.

Acknowledgements

I would like to thank Dr. James P. Bobis, for invaluable assistance and support while working on this project, and for introducing me to the wonderful world of analog and digital control systems. I thank Dr. Benjamin C. Kuo, author of two very clearly written texts from which a majority of my ideas and design procedures are adopted. Thanks to The MathWorks, creators of MATLAB, and especially John N. Little and Alan J. Laub, authors of the Control Systems Toolbox User's Guide, for providing ample information and examples to clearly illustrate software functions used in this project. Finally, I would like to thank all those not mentioned who provided any source of information, ideas, or moral support.

This project is dedicated to my wife Michelle, the largest source of support. Without her endless patience and tolerance, this project would not have been possible.

Table of Contents

Table of Illustrations

Figures:

Tables

Abstract

The main purpose of this project is the design and programming of software which designs PD, PI and PID controllers for a plant transfer function, given a user specified desired output phase margin and error constant as design parameters. The controllers to be designed are restricted to controllers in series with the controlled process, with unity feedback. Frequency domain analysis techniques are used as the basis for the controller design algorithm, the methods of which are derived from the reference texts listed, and outlined in this document. The project software is written in MATLAB, to take advantage of many of its existing controls-based frequency domain analysis functions, and to provide a medium for full exploitation and analysis of the software's design results.

Introduction

1. Controllers

Controllers and control systems exist in a virtually infinite variety, both in type of application and level of sophistication, however, any control system can be described by its inputs, components, and objectives. For example, the temperature control system in a house is a system that uses the current temperature as its input, a thermostat and heat source as its components, and constant temperature regulation as its objective. This type of control is referred to as a *process control* or *regulator system*, and its main objective is to hold a controlled variable at or near a constant value. Another type of control system is a servomechanism, which is designed so that the output variable follows the input variable as closely as possible as the input changes. Examples of a servomechanism include the power steering control in an automobile, and the rotor mechanism in an antenna rotor.

Three types of controllers that are popular in industry, and often used in controls classes as a basis for learning control systems, are the PD, PI, and PID controllers, which can be used as servomechanisms or process controllers. Manufacturers of process controllers use the PID almost to the exclusion of other controllers, because of its flexibility (Van de Vegte). Any of the controller coefficients can be zeroed, so that the PID controller can function as a PD, PI, or P controller. In order to use any of these types of controllers, the coefficients of the controllers must be found, a process commonly referred to as *tuning*. There is a certain level of difficulty to tuning a PD, PI, or PID controller analytically, and many users tune the controllers by a trial-and-error method. This requires time, and a lot of experience. With the widespread availability of personal computers, control systems can now be modeled by the computer, and many of the classical analytical control methods can now be quickly simulated without regard to the numerous amount of calculations required, which were previously done by hand. The software proposed in this report is designed to calculate the coefficients of the PD, PI, and PID controllers using the algorithms and design methodologies of these classical methods.

There are two basic design methodologies for designing controllers: designing in the time domain, and in the frequency domain. To design in the time domain given the plant transfer function, root loci and/or root contour plots are used to find optimum placement of controlled system roots, as a variable or variables are are changing. Due to the subjective nature of choosing root locations using root loci or root contours, this seems inefficient and difficult to translate into a workable software algorithm. To design in the frequency domain, the only requirements are magnitude and phase Bode plots of the plant transfer function, an understanding of phase margin specifications, and how the desired controller affects the plant transfer function response in the frequency domain. Although this design method is somewhat subjective (as is any design process), it seems better suited for a computer algorithm. Since the desired output phase margin would be specified by the user in the software proposed, most of the subjective process is left to the user. Therefore, the algorithm need only calculate the original phase margin of the system, and the controller coefficients that would result in the desired phase margin and error constant. It is for this reason that the frequency domain design methodology is chosen over the time domain design approach.

2. Frequency Domain Analysis and Design

In order to devise a design algorithm that is suitable for programming, the analysis and design methodology must be understood. One of the fundamental means by which to increase stability in the output response of a plant transfer function is to increase the amount of phase at the phase margin, that is, the frequency where the magnitude response of the function is of unity gain. However, increasing phase has an effect of damping system response as well, therefore increasing rise time, so too much added phase may be undesired. Too little phase margin may result in excessive overshoot or oscillation in the output response, and may lead to an unstable response with small changes in the plant transfer function (due to system aging, wear, or inaccuracies in plant modeling). Therefore, care and understanding must be exercised when choosing a desired phase margin. Since most plant systems can be modeled by a second order transfer function, and since second order prototypes are used as a basis for understanding higher-order systems, a design rule-of-thumb for second

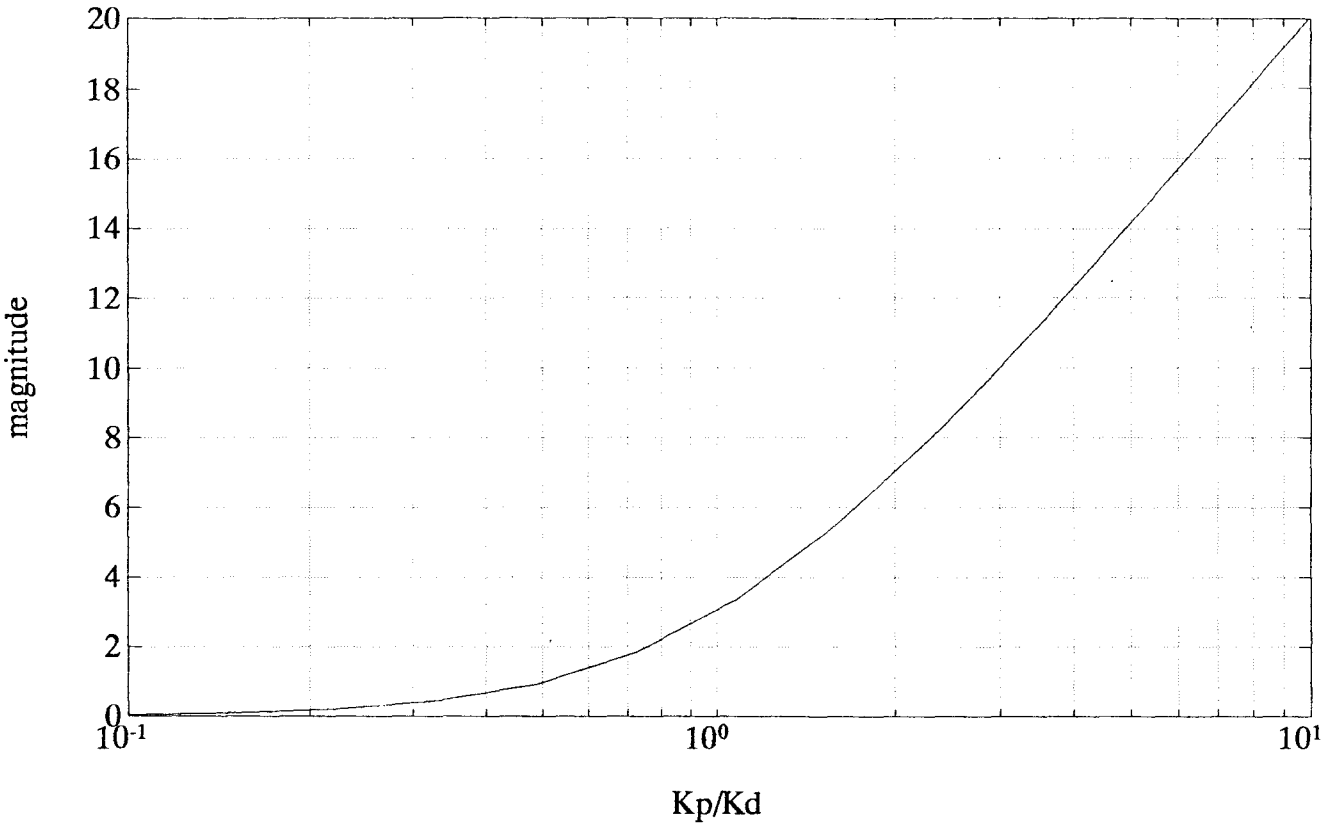
order systems is used for finding a suitable phase margin. This rule is to design the controller to provide 45 degrees of phase margin, which (for a second order system) would result in less than 5% overshoot. For higher order systems, systems with time delay or other non-linear systems, the overshoot would probably be greater, however, designing for 45 degrees of phase margin is a sensible starting point.

In the frequency domain, the PD controller has the effect of adding 90 degrees of phase to the overall response above a critical frequency (Kp/Kd), as well as adding 20dB per decade of gain to the magnitude response above this frequency (figure 1). This has the effect of increasing the bandwidth of the overall system, which translates into a decrease in rise time, as well as providing phase margin for stability. The negative effect of the PD is to increase system overshoot and oscillation for some values of Kp/Kd, and to increase settling time if Kp/Kd is too near to the origin.

The PI controller degrades the phase of the system by 90 degrees below a critical frequency (Ki/Kp), as well as attenuates the magnitude by 20log(Ki) dB above the critical frequency (figure 2). The magnitude attenuation results in increased system stability, less overshoot and oscillation, but also increases rise time due to the damping nature of integration. The PI controller also, more importantly, increases the type of the system by one, therefore improving steady-state error. For example, if the system is a type one (one pole at the origin), a PI controller will increase the ramp-error constant to infinity, decreasing the steady-state error to zero. Since the PI degrades the phase response of the system near and below the critical frequency, Kp and Ki must be chosen such that the frequency Ki/Kp is as far to the left as bandwidth allows so the existing phase margin is not degraded, and rise time is not severely affected.

The PID controller combines the advantages of the PD and PI controllers, by increasing bandwidth and phase margin, resulting in a decrease in rise time, overshoot, and oscillation, and overall stability, if the coefficients Kp, Ki, and Kd are chosen carefully.

Figure 1: PD Controller Frequency Response (Kp = 1)



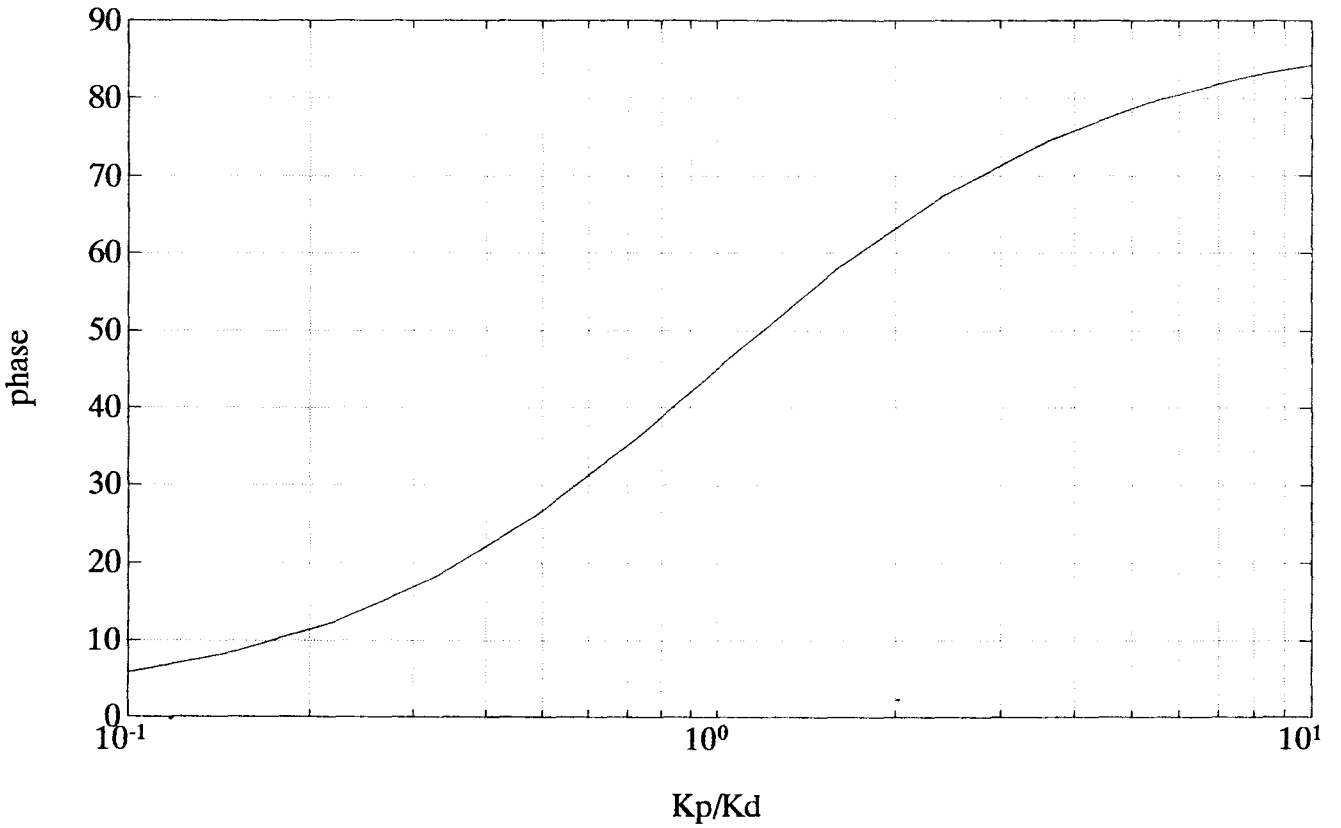
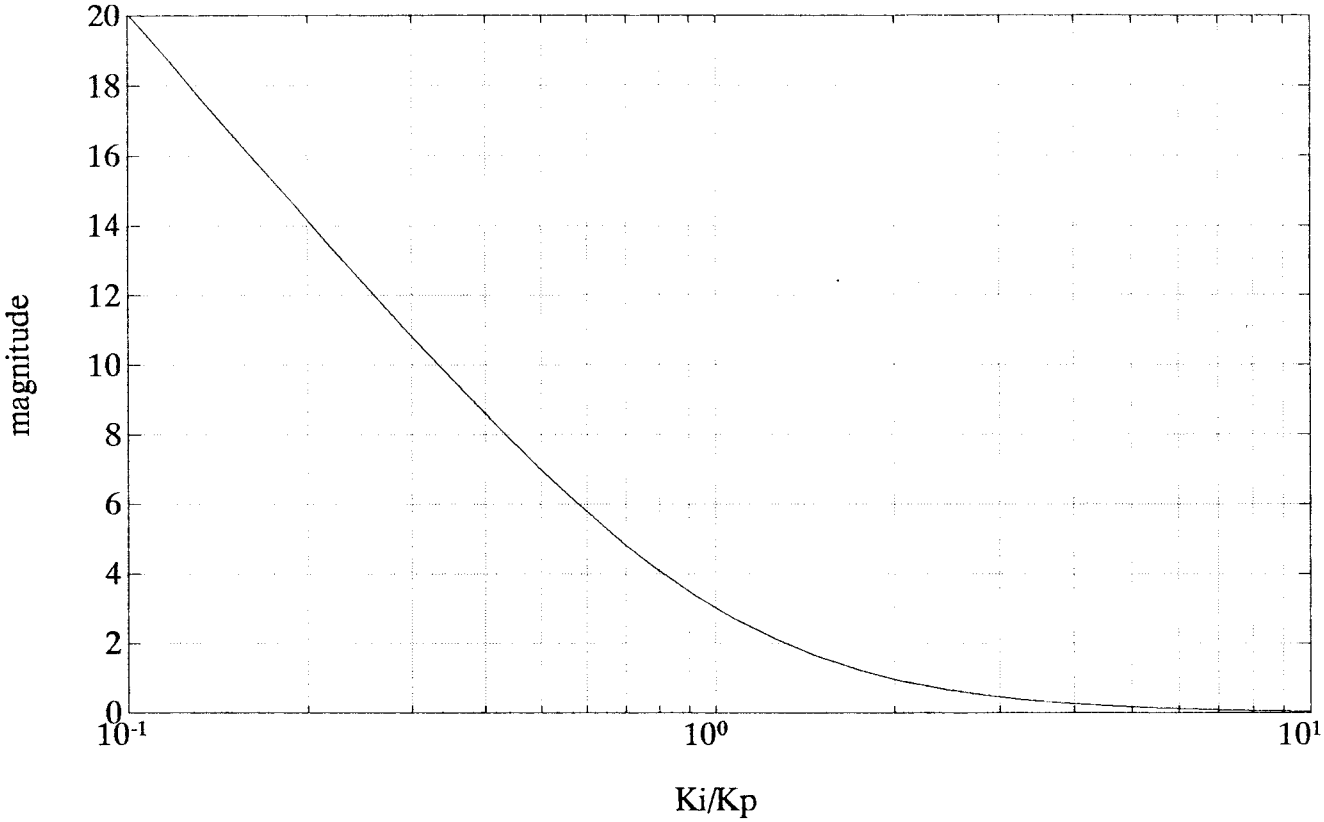
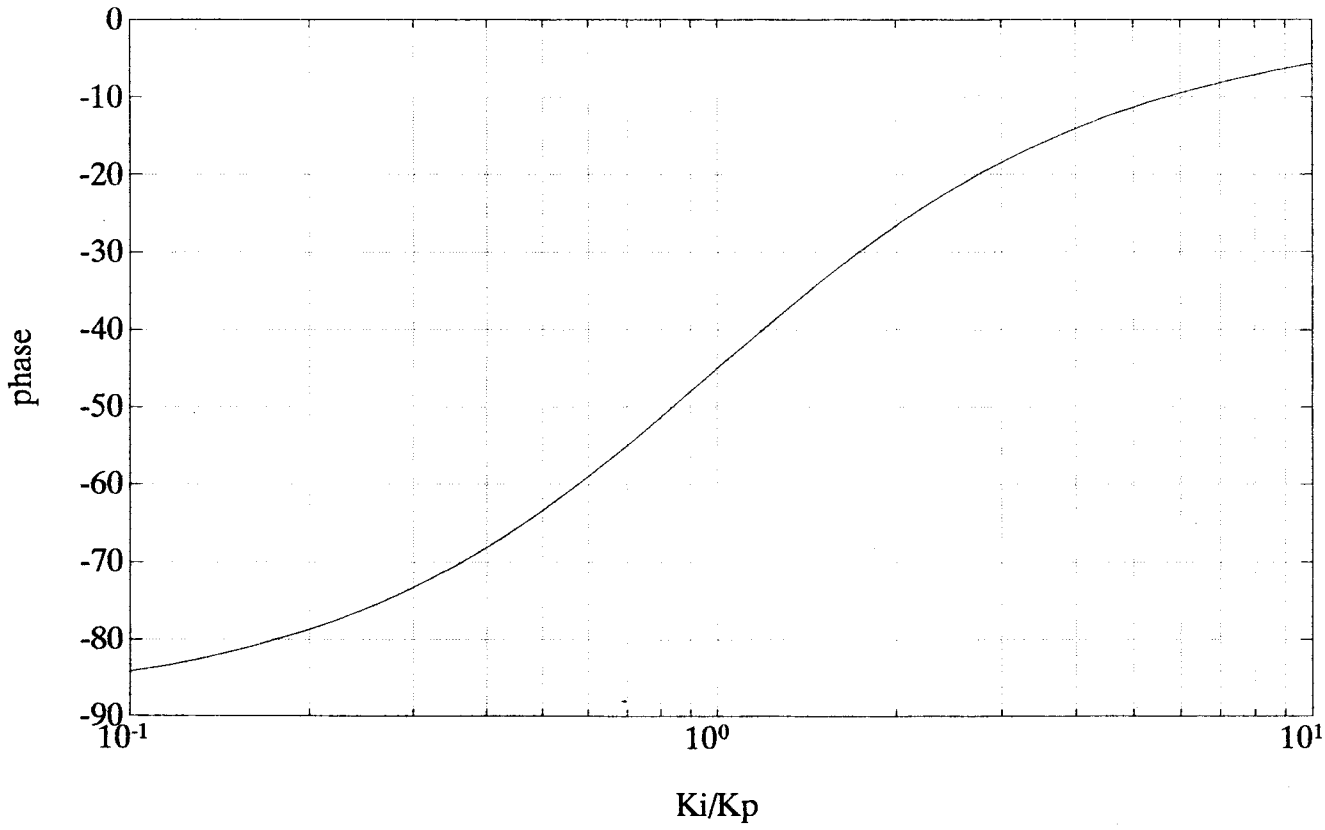


Figure 2: PI Controller Frequency Response (Ki = 1)





3. Introduction to MATLAB

Before any software can be designed or programmed, at least a basic understanding of the syntax and formats of the language must be learned. MATLAB, which stands for *matrix laboratory*, is a matrix oriented computing environment designed for high-performance numeric computation. It is an interactive system, as well as a programming language, designed to solve many numeric problems in a fraction of the time required to program in a language such as C, Fortran, or Pascal. MATLAB features a variety of matrix-manipulating functions, as well as families of application-specific functions contained in *toolboxes.*. The controller designing software designed and programmed for this project utilizes many of the functions in the Control Systems Toolbox.

While the basic data element of MATLAB is the matrix, the way in which the matrix is interpreted is where the power of MATLAB lies. If given the vector matrix: [1 2 3], it can be interpreted as: 1) a vector matrix; 2) a matrix of polynomial coefficients in descending powers of x; 3) a matrix containing control characters to pass to another function; etc. The user can create any function accepting matrices as input, and interpret and manipulate those matrices however they may see fit.

Most MATLAB functions, especially those that manipulate matrices as data, must fit a particular format. The function call requires three parts: the function name, the function's input parameters, and the function's output parameters, and requires the syntax format in figure 3. If so desired, the function can be programmed so that it need not have output parameters, however, it must have at least one input parameter to be a function. The function itself has two parts: the function declaration, and the function body (processing routine using arithmetic operators and/or other MATLAB functions), is saved as a file with a **.m** extension commonly referred to as an **.m** file, and requires the format in the function example in figure 4. This figure gives an overall example of how a MATLAB function, and how MATLAB itself is structured (note that comments are denoted by %).

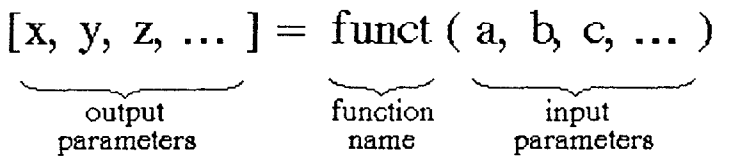


Figure 3: MATLAB Function Call

```
function r = rank(x,tol)
%RANK      Rank.  K = RANK(X) is the number of singular values  of   X
%      that are larger than MAX(SIZE(X)) * NORM(X) * EPS.
%      K = RANK(X,tol) is the number of singular values of  X that
%      are larger than tol .
s = svd(x);
if (nargin = = 1)
      tol = max(size(x)) * s(1) * eps;
end
r = sum(s > tol);
```

Figure 4: MATLAB Function Example

Applications

The PD, PI, and PID designing software is designed with the intent to be used as a teaching/learning aid, especially for the ELE380, ELE480, and ELE481 controls classes. The intent arises from the project advisor's need and desire for such software, and from a lack of understandable and easy to use design software (or any PD, PI, and PID design software, for that matter). However, the software designed is not limited to this use. It is fully functional, and may be used as a tool in real design applications.

Objectives

The objectives of this project are:

1) To design and program software that reliably and accurately design PD, PI, and PID controllers, given a desired phase margin to be met, and an optional error constant;
2) To design the software such that it designs the controllers in frequency domain;
3) To design the software using MATLAB commands, or a combination of MATLAB commands and C code if necessary, to take full advantage of MATLAB's frequency analysis functions, and subsequently to provide a medium for full exploitation of the software's design results;
4) To design the software such that it is consistent with the format of most MATLAB functions requiring a minimal amount of user interface, yet so they remain easy to use and understand;
5) To thoroughly document the software consistent with other MATLAB reference manuals, and to provide the user with enough information and examples to become comfortable with its use;
6) To program the PID designing software such that it makes use of the PD and PI designing software as subroutines.

Appended objectives:

7) To design the software such that it retains accuracy, yet accomplishes its task in a minimum amount of time;
8) To program the PID designing software such that a unit-step response graph of the uncontrolled system (if stable) and of the controlled system are plotted.

Procedure

1. Algorithm design

In order to design a PD controller in the frequency domain, Kuo proposes procedures as follows:

1) *Since the proportional-control gain constant Kp can be combined with the series gain of the system, the zero-frequency gain of the PD controller, or Kp, can be regarded as unity.* This means that Kp is effectively divided out of the controller equation, and incorporated into the transfer function gain constant, leaving the controller equation $G_c(s) = (1 + (Kd/Kp)s)$. For this software, Kp can be chosen given error constant information, or can be chosen to be 1, and the user alerted.

2) *Place the corner frequency of the controller (Kp/Kd) such that an effective improvement of the phase is realized at the new gain-crossover frequency.* Or, choose Kp/Kd such that the desired phase margin, or best possible phase margin, is met.

The trick to designing the PD controller using these procedures is knowing which values of Kp/Kd to try. Since most of the negative changes in the magnitude and phase responses of the Bode plots of a plant transfer function occur within the boundaries of the largest and smallest pole frequencies, it seems reasonable to vary the frequency Kp/Kd within this range to find the optimum value. This observation gives a starting value of Kp/Kd for a suitable algorithm. Since Kp = 1 for this procedure, only Kd need be evaluated.

Since Bode plots are logarithmic, it also seems reasonable that the frequency Kp/Kd must be varied in an exponential manner to produce linear results on a Bode plot. This observation gives a method to incrementing the frequency Kp/Kd to find the optimum value. Since greater accuracy requires smaller increments, and faster process time requires larger increments, the incrementation amount will be determined through a trial and error experimentation process, to find reasonable accuracy for a short process duration.

To design the PI controller, Kuo proposes:

1) *Determine the phase margin and gain margin of the uncompensated system using Bode plots.*

2) *For a certain specified phase margin, find a new gain-crossover frequency w_c' corresponding to this desired phase margin.* That is, find the point on the phase plot where the phase is the desired phase margin, and locate the frequency where that occurs.

3) *Lower the magnitude of curve of the uncompensated system down to 0dB at the new gain-crossover frequency by :*
$Kp = 10^{(-|Gp(wc')|dB/20)}$. This equation determines Kp.

4) *Select Ki such that the the corner frequency Ki/Kp is far below the new gain-crossover frequency.* $Ki = (w_c' / 10) Kp$. This locates the frequency Ki/Kp a decade away from the new gain-crossover frequency, so the PI controller does not significantly degrade the new phase margin at w_c'.

5) *If Ki is selected based on an error constant, determine Kp by using a range of values.* Or, if an error constant must be met, choose Ki based on the error constant, and choose Kp such that the desired phase margin, or best possible phase margin, is met.

If there is no error constant to be met, the algorithm for finding the coefficients Kp and Ki are the equations given above. If an error constant must be met, Ki is chosen by the conditions necessary to satisfy the error constant, and Kp is varied through a range of values to find an optimum value. This algorithm parallels the PD design algorithm, where Kp replaces Kd and Ki replaces Kp in the PD algorithm. Also, in order to design the PI using the PD algorithm, the plant transfer function denominator must be multiplied by 1/s, so the PI equation is of the form $G_c(s) * s = (Ki + Kp\ s)$.

Finally, to design the PID controller, Kuo's proposal is:

1) *Design the PD controller with Kp = 1.*

2) *Incorporate the PD controller into the plant transfer function, and design the PI controller.*

3) The coefficients for the PID controller are given by:

$$Kp_{PID} = Kp + Kd * Ki$$
$$Ki_{PID} = Ki$$
$$Kd_{PID} = Kd * Kp$$

where Kd is found from the PD controller design, and Kp and Ki are found from the PI controller design.

2. Software Design

In order to maintain consistency with other MATLAB functions, the function **bode.m** was examined. This function was chosen as a programming model because a) it is a typical example of a frequency-based controls function, and b) its input and output requirements are similiar to those desired in the controller designing software being programmed. (The function **bode.m** will be referred to in this report; it or any other MATLAB function can be referenced by loading the **.m** file into any word processor.)

When analyzing **bode.m**, the large function header was noted. This header gives a brief summary of the function's use, how it must be called, and what parameters it requires. This header also doubles as the function's help information, when the line **help bode** is entered at the MATLAB command line prompt. Therefore, to stay consistent with general function formats, and to provide as much documentation as possible for the functions, the function headers must fit this format, and provide as much information as possible. However, the function description is limited to one screen (any more text would be lost at the top of the screen when calling the help function), so it also must be as compact and concise as possible.

A second observation made concerning the function **bode.m** was that it limited the number and type of input and output arguments, in an effort to require the user to input the required data to be processed and output the resultant data. These input and output parameter checks were modified to fit the format of the functions being designed. Aside from the parameter checks that were directly adapted from the function, **bode.m** was otherwise used as a general function format reference, and no other direct software adaptations were made.

3. PD function: designpi.m

The PD function was designed and programmed first, since the PI controller could utilize the same algorithm. In following the PD algorithm noted above, a number of existing functions were used to achieve the design goal, and the software algorithm is as follows:

1) *Determine Kp for a given error constant.* If no error constant is given, set Kp = 1.

2) *Determine the magnitude of the smallest non-zero root of the denominator, and initialize loop parameters.* This provides the starting point for a choice of Kd.

3) *Increment the magnitude of Kd.* This provides a new value for Kd, based on its previous value. The magnitude of Kd was tentatively incremented by 0.02, to provide 50 values of Kd per decade of change, however, this figure may change based on the accuracy of the phase margin found, and the time required for the function to run.

4) *Calculate the phase margin of the controlled system with the new Kd.* This is implemented by the functions **conv.m**, which multiplies the numerator ploynomial by the controller polynomial; **bode.m**, which determines the frequency response of the controlled system; and **margin.m**, which determines the phase margin of the system from the frequency response.

5) *Repeat steps 3 and 4 until the desired, or maximum attainable phase margin is found.* Since there may be two possible values of Kd to satisfy the desired phase margin, both must be kept. Therefore, after finding the first value of Kd, the steps 3 and 4 must be repeated to find the second value of Kd. If the desired phase margin is not met, the value of Kd which gives the largest possible phase margin is saved.

6) *Return the values of Kp, Kd, and the controlled system phase margin.* If there are two values of Kd, then two element matrices of Kp, Kd, and phase margin are returned, where corresponding values have identical indices.

4. PI function: designpi.m

The PI function differs from the PD function in that it must design the PI controller one of two different ways, depending on whether or not an error constant is given. If no error constant is given, the algorithm is:

1) *Determine the frequency of the desired phase margin (or of the maximum possible phase margin if the desired does not exist) from the frequency response of the uncontrolled system.* This is done using the function **bode.m** and a loop to check for the existence of the desired phase margin.

2) *Determine the magnitude M of the function at the new frequency.* This is found using the data from the previous use of **bode.m**.

3) *Kp = 10^(-|log10(M)|).*

4) *Ki = Kp * (w_c' / 10).* Locates Ki one decade away.

5) *Return the values of Kp, Ki, and the controlled system phase margin.*

If an error constant is specified, Kp and Ki are found using the PD algorithm with the following modifications:

1) *The PD controller variables Kp and Kd are replaced with the PI variables Ki and Kp.* That is, Kp_{PI} replaces Kd_{PD} and Ki_{PI} replaces Kp_{PD}.

2) *The plant transfer function must be multiplied by 1/s.* Or, the plant transfer function denominator must be multiplied by s.

5. PID function: pid.m

In order to satisfy one of the given objectives, the PID controller must make use of the PD and PI functions. Fortunately, Kuo's PID design procedure designs the PD and PI separately, so this is easily incorporated into the following algorithm:

1) *Design the PD controller with Kp = 1.* This is done by specifying no error constant when calling the function **designpd.m**. If the PD controller meets the desired phase margin indicated, the user is prompted to continue with the PI design, or stop.

2) *Incorporate the PD controller into the plant transfer function.*

3) *Design the PI controller.* The PI controller is not designed if the PD controller meets the desired phase margin, and the user answers "no" to the continue prompt.

4) *Plot step response if user specifies.* If the user answers "yes" to the plot prompt, a step response of the controlled system is plotted, along with a plot of the uncontrolled system (if stable), and the PD response (if stable and if it meets phase margin specifications). To plot these functions, the plot range needs to be calculated. This is done by examining the individual plot ranges. If the individual ranges differ by a large margin, a compromise range must be found, so as not to lose the integrity of either plots. If this is not possible, then the integrity of the controlled system plot outweighs the uncompensated system plot, and that range is used.

5) *Return the values Kp, Ki, Kd, and the controlled system phase margin.* The coefficients for the PID controller are given by:

$$Kp_{PID} = Kp + Kd * Ki$$
$$Ki_{PID} = Ki$$
$$Kd_{PID} = Kd * Kp$$

where Kd is found from the PD controller design, and Kp and Ki are found from the PI controller design. Since the functions **designpd.m** and **designpi.m** can return multiple values of the controller coefficients, the largest of these values is chosen (which generally give a wider bandwidth). Similiarly, for step 1 above, the largest value of Kd is incorporated into the PD function to design the PI controller.

Modifications

The functions were first tested for all possible combinations of incorrect input and output parameters, and the error responses checked and modified. When the responses were found to be satisfactory, the functions were next tested with a plant transfer function Kuo uses in several of his textbook examples:

$$Gp_1(s) = \frac{100}{s\,(1 + 0.1s)(1 + 0.2s)}$$

When testing the PD design function with a desired phase margin of 45 degrees, is was immediately noted that the function required a considerable amount of time to calculate its values, approximately 3 minutes (the function was run on a 386-based computer with math co-processor; actual computational times may vary). Upon examination of the results of the computation, the returned values were accurate compared to Kuo's solutions. Kuo's value of Kd was 0.20, which gave a phase margin of 17.98 degrees; the software results gave a value of Kd as 0.2048, and a phase margin of 18.02 degrees. The magnitude increment value was then changed to 0.05, and tested. This change resulted in a faster computation time (approximately one minute), but unacceptable phase margin results, where the phase margin was accurate only to within approximately plus or minus 7 degrees. The magnitude increment was tested with values of 0.04, 0.03, and 0.025. The value 0.03 was finally chosen as the magnitude increment value, providing a suitable compromise between accuracy (approximately plus or minus 2 degrees) and speed (approximately 2 minutes - comparable to the MATLAB function **rlocus.m**). The magnitude increment of the PI design function was also adjusted to this value.

The PD, PI, and PID controllers were next tested with the plant transfer function:

$$Gp_2(s) = \frac{2500}{s\,(s + 25)}$$

which Kuo uses as in an example to design a phase-lead controller. This function was chosen to test the error constant function of the PD controller.

Upon testing with the error constant equal to 10, the results yielded a phase margin within 1 degree of the desired, and coefficients yielding the desired error constant.

Next, the PID controller was tested with the plant transfer function:

$$Gp_3(s) = \frac{100}{s^2 + 10s + 100}$$

which Kuo uses as a design problem. The design requirements specified are: less than 2% overshoot, less than 0.02 second rise time, and ramp-error constant of 100. Upon testing, the function returned coefficients for the PID controller which resulted in overshoot and rise time less than the required amounts, however, the returned phase margin was infinite. Upon analysis of the frequency response plots of the controlled system, and observance of the phase margin for each iteration of the function, the problem was traced to the results of the **bode.m** function call. When the values of Ki reached a certain point during the iteration process, the **bode.m** function generated a magnitude plot which had no 0dB crossing, and therefore no phase margin value. When the **margin.m** function was then called, it interpreted no phase margin as infinite phase margin. This problem was corrected by increasing the frequency range over which the **bode.m** function operated. This correction was made to the **designpd.m** and **designpi.m** functions, and also resulted in a decrease in function computation time.

The final plant transfer function used for testing was:

$$Gp_4(s) = \frac{22.104}{s^3(s + 0.9216)}$$

a transfer function derived from an adaptation of Loberg's ball and beam senior design project. Although it is not expected the PID controller would stabilize this system since the PID controller only adds 90 degrees of phase, this function was used to test whether or not the function could handle a high-order system, particularly one that has several poles at the origin. The function could not handle this system, and produced spurious results. The problem was again traced to the operation of the **bode.m**

function, where the phase plot returned was shifted so that it was plottable within the range of -180 degrees to +180 degrees. Therefore, any function of type two or greater would be shifted at by at least 360 degrees. This problem was corrected by calculating the input function type, and then subtracting 360 degrees for every time the plot was shifted.

After testing the functions for operation, some final issues needed to be resolved. The problem of the inclusion of an error constant in the PD designing function was an ongoing concern, since one of the objectives of the function design was to include the error constant as a design parameter. However, the function as written could only handle ramp-error constants, or greater. Since the steady-state error equation for a ramp-error constant, or any greater error constant is $e_{ss} = R/K_x$ (the error constant equation used by the function), and the steady-state error equation for a step-error constant is $e_{ss} = R/(1 + K_p)$, the function could not handle a step error constant. This issue was resolved by removing the provisions for handling any error constant, setting Kp = 1, and including information for the user on how to design for a desired steady-state error. This solution was chosen so the function would remain easy to use, follow Kuo's design principle, and so the function would not require more calculations to slow the process time down further. The function could still design for a desired error constant, but the user must incorporate this error-constant into the plant transfer function. This modification was made only to the PD design function, since the PI controller adds a pole to the origin of the plant transfer function. The smallest error constant that could be used in the PI design is a ramp-error constant, and therefore the step-error equation would not apply.

Another concern is the inclusion of a feature that tests the input transfer function to determine if the control system to be designed is useful. This concern is raised by Bateson's design process, where he includes equations to test the usefulness of PD and PI controllers, given the plant transfer function. After careful consideration, these features were not included in the software design. Since the software is designed to optimize

the input transfer function, any controller that would degrade the input function would not be returned as output (the coefficients returned would be zero, or essentially zero). Also, it is assumed that the software user knows what he is doing, so the user would not attempt to design a controller where it would not be useful.

The last ongoing concern was the generation of the step response plots. As written originally, once the PID controller finished its coefficient calculations, the user was then prompted as to whether or not a step response plot was desired. Since the design objectives require minimal user interface, and consistency with other MATLAB functions, the function was modified such that no plot prompt is given. If no left-hand output arguments are specified, the function calculates the controller coefficients and plots the step responses with the coefficient and phase margin information. If the output parameters are specified, the coefficients and phase margin are returned through the parameters, and no plot is generated. This modification reduces user interaction with the function, and is consistent with **bode.m** and other MATLAB functions.

Results

The PD, PI, and PID controllers were finally tested using the four aforementioned plant transfer functions, and the results are presented in tables 1 through 4, and in figures 5 - 12. The results are compared to Kuo's results or design requirements wherever possible.

Testing the functions using:

$$Gp_1(s) = \frac{100}{s\,(1 + 0.1s)(1 + 0.2s)}$$

design a controller to stabilize the system represented by Gp1(s). The input (desired) phase margin was given to be 55 degrees, and no error constant was specified.

Results:

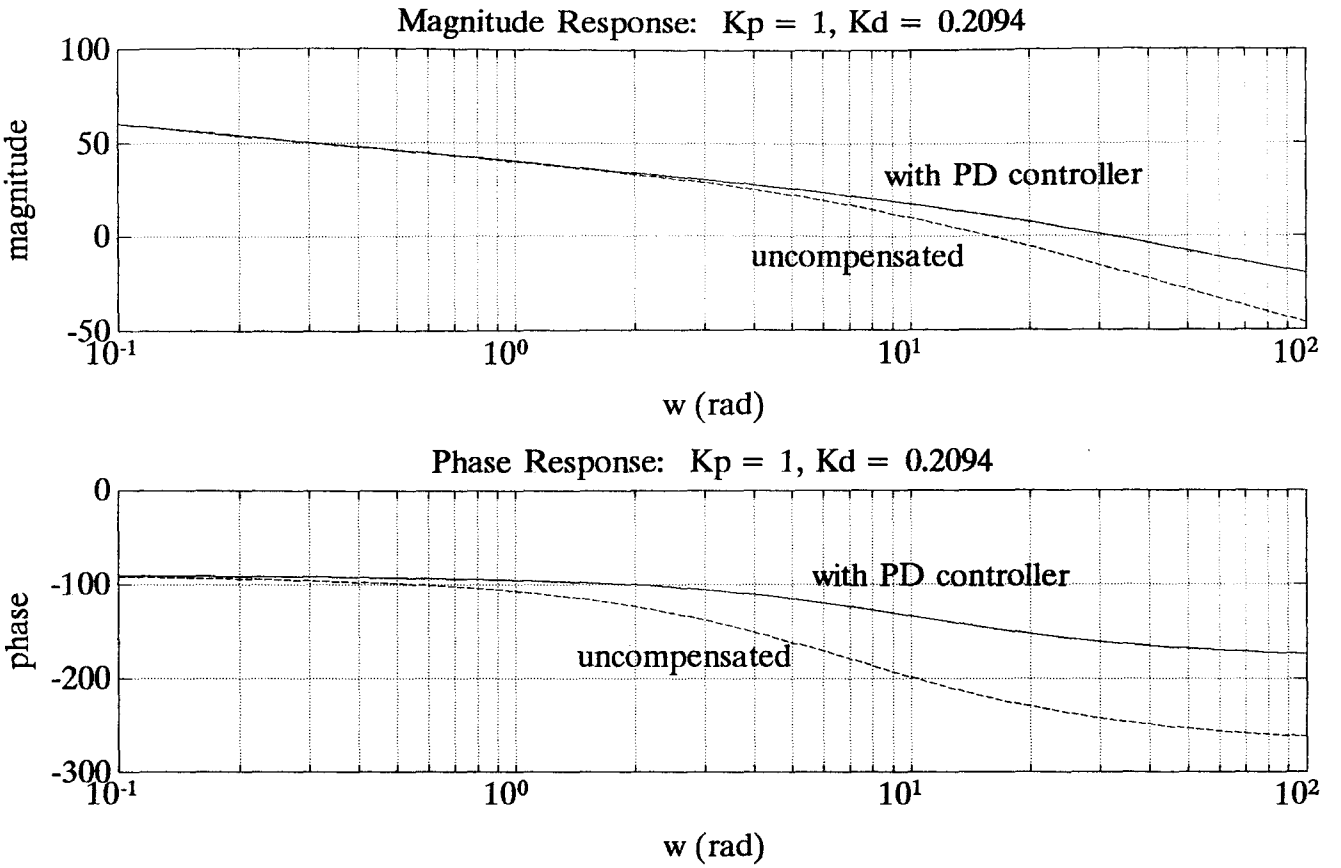
	Example				Computed				
Controller	Kp	Ki	Kd	Phase Margin	Kp	Ki	Kd	Phase Margin	Run Time
uncomp.	1	0	0	-40.31	1	0	0	-40.24	0
PD	1	0	0.20	17.98	1	0	0.209	18.67	38.79sec
PI	0.0178	0.0036	0	55	0.0181	0.0030	0	55.825	2.41sec
PID	0.0735	0.007	0.035	56	0.0685	0.0339	0.0129	56.51	40.93sec

Table 1: Software Test Results Using $Gp_1(s)$

As shown in table 1, the software design results parallel Kuo's example results, for all controllers. As would be expected for any design software, certain amounts of variation will exist, since designing any controller is a subjective process, however, the end results of the design process meet the requirements, and compare to those given by Kuo. The computation times given in table 1 are used only as a basis of comparison and analysis, and will vary depending on processor setup used.

By examining the magnitude and phase responses in figure 5, the PD controller increases the bandwidth of the system, as well as increases the phase of the system for high frequencies by 90 degress, as expected and

Figure 5: PD Testing Results Using $Gp_1(s)$



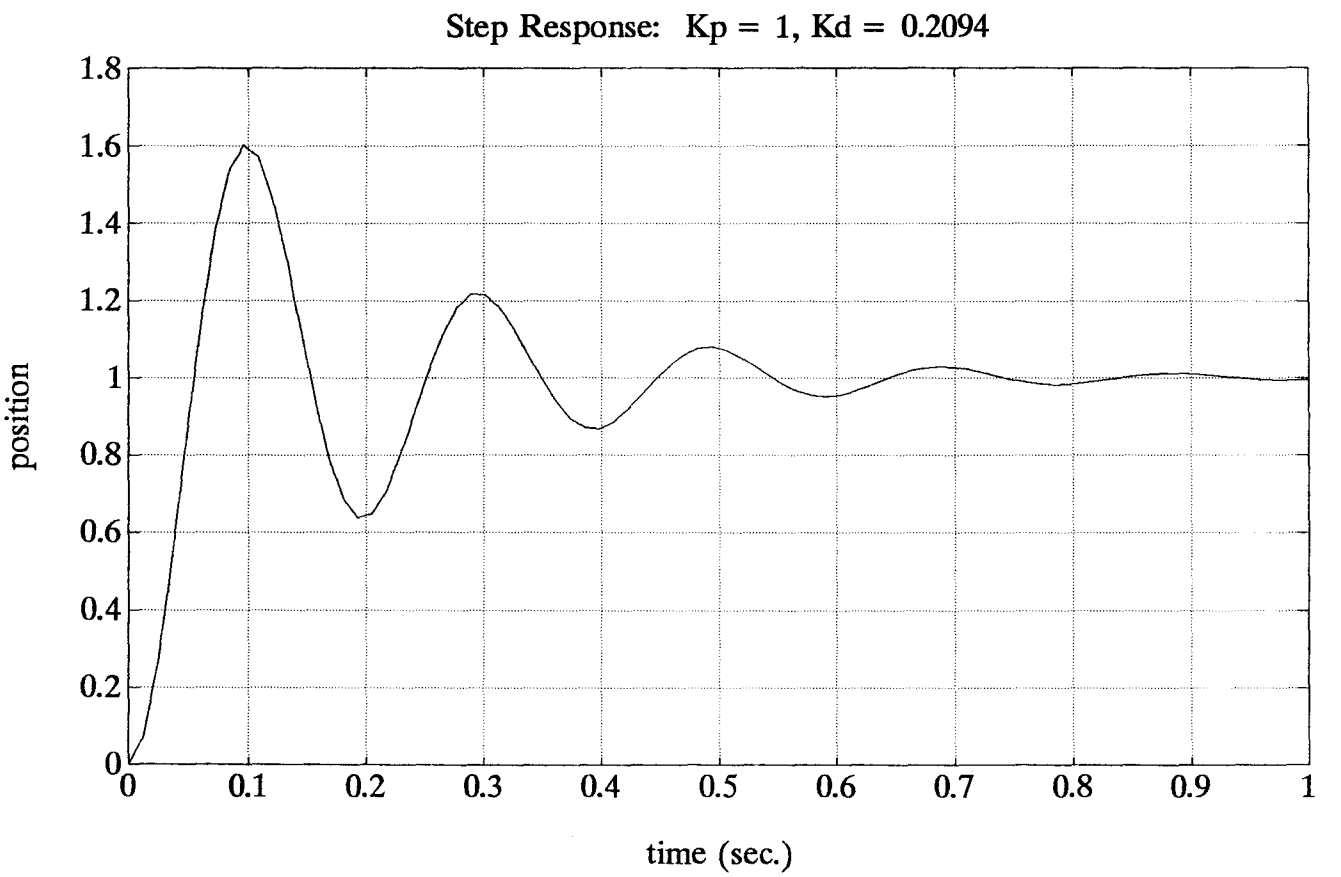
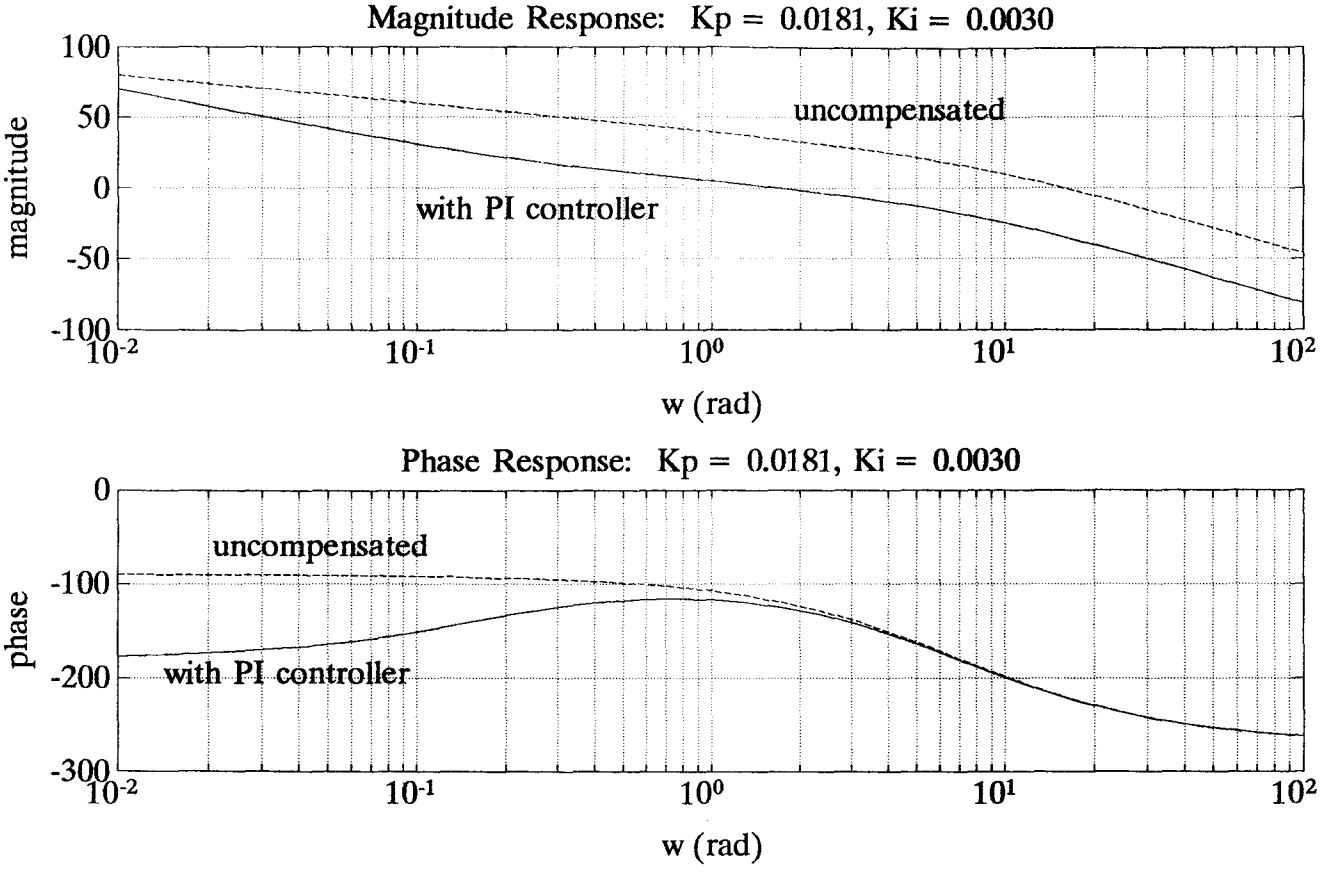


Figure 6: PI Testing Results Using $Gp_1(s)$



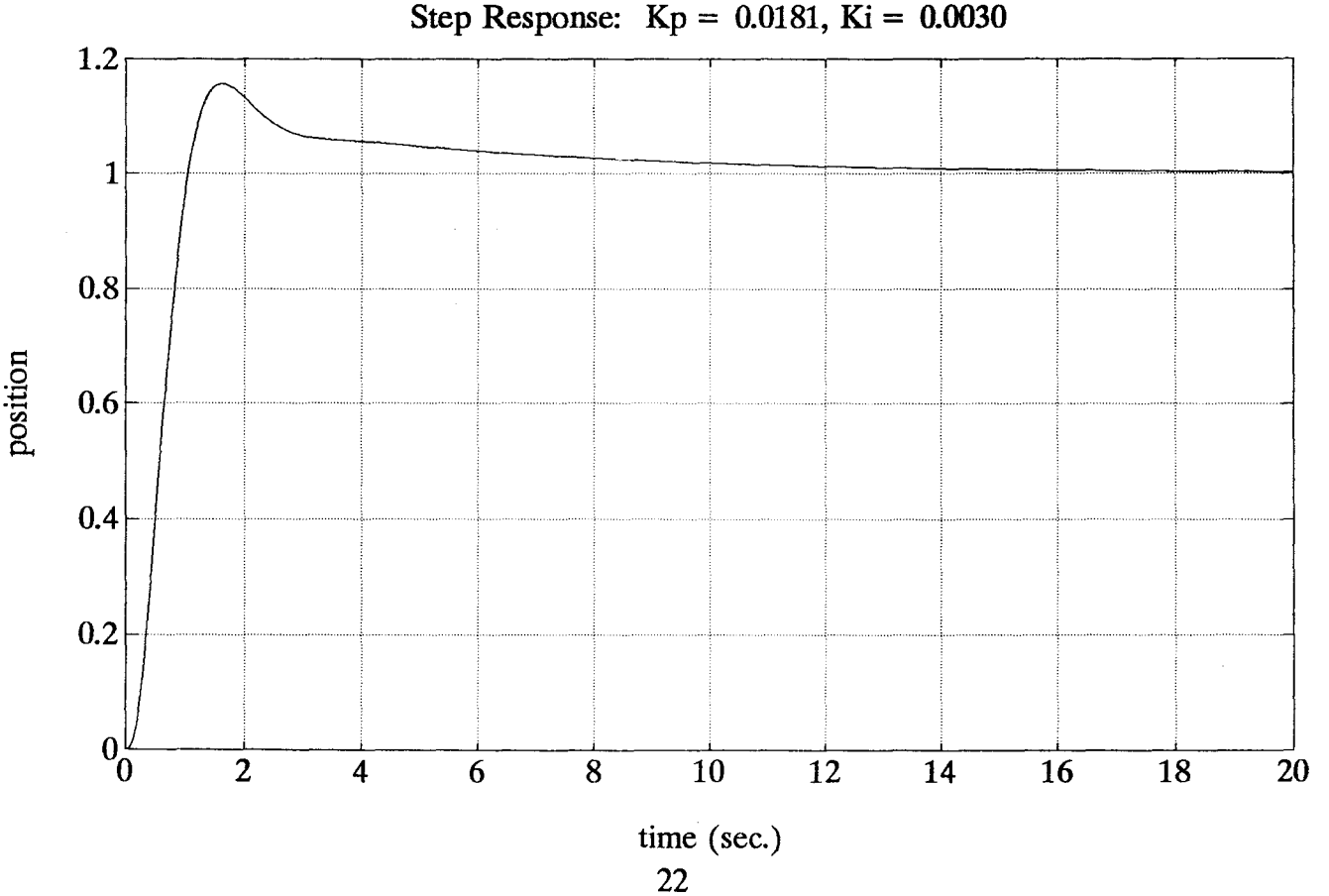
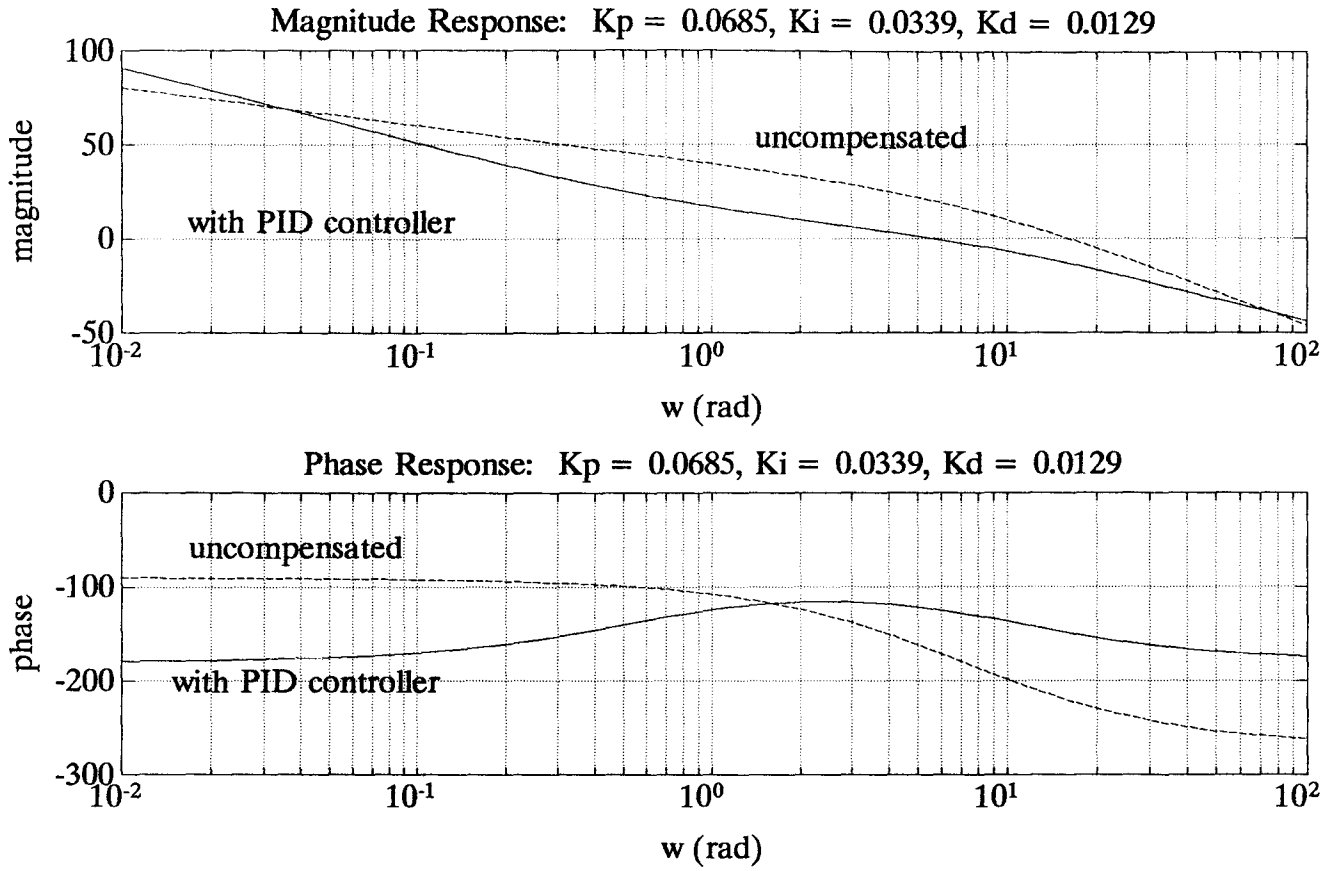


Figure 7: PID Testing Results Using Gp1(s)



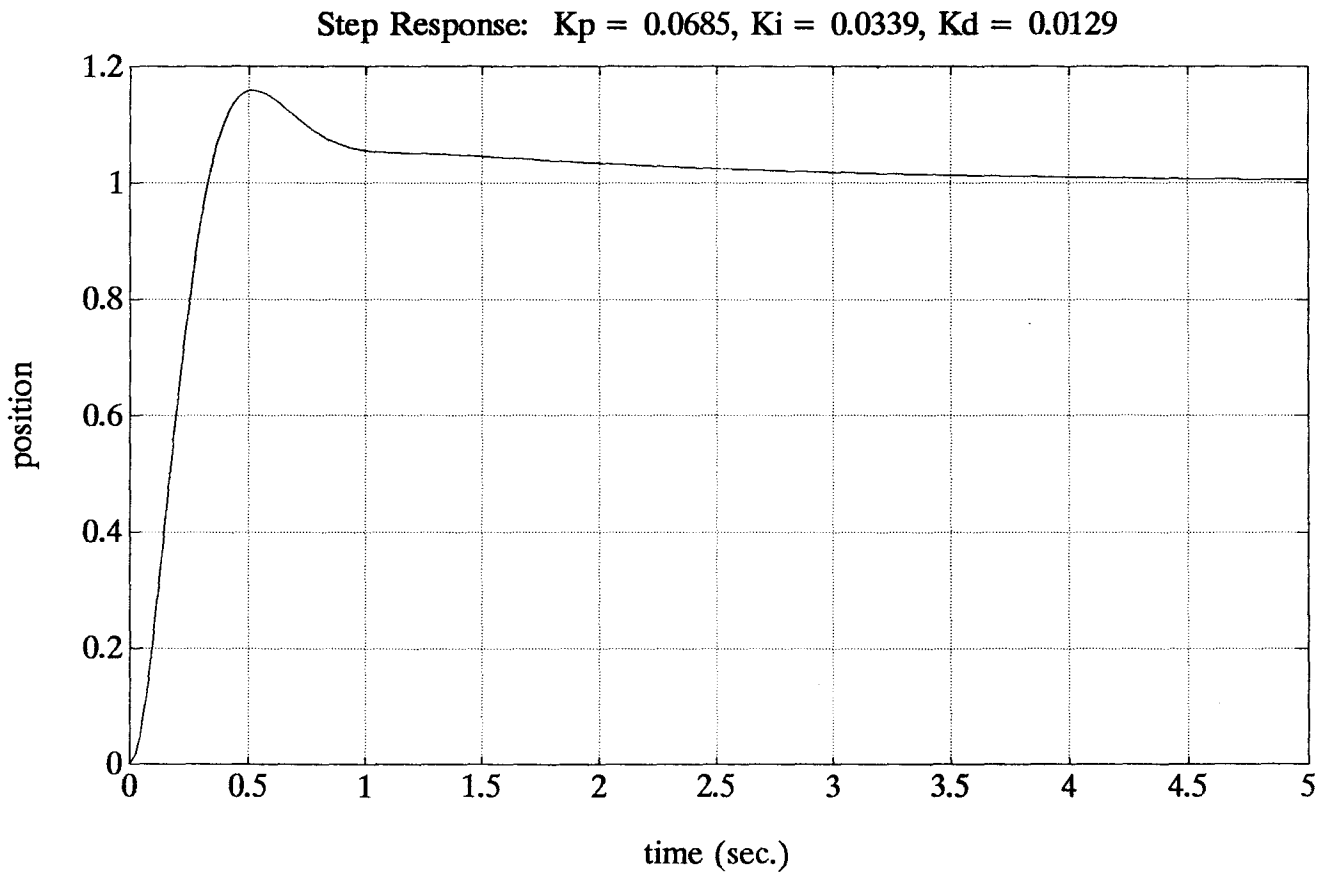
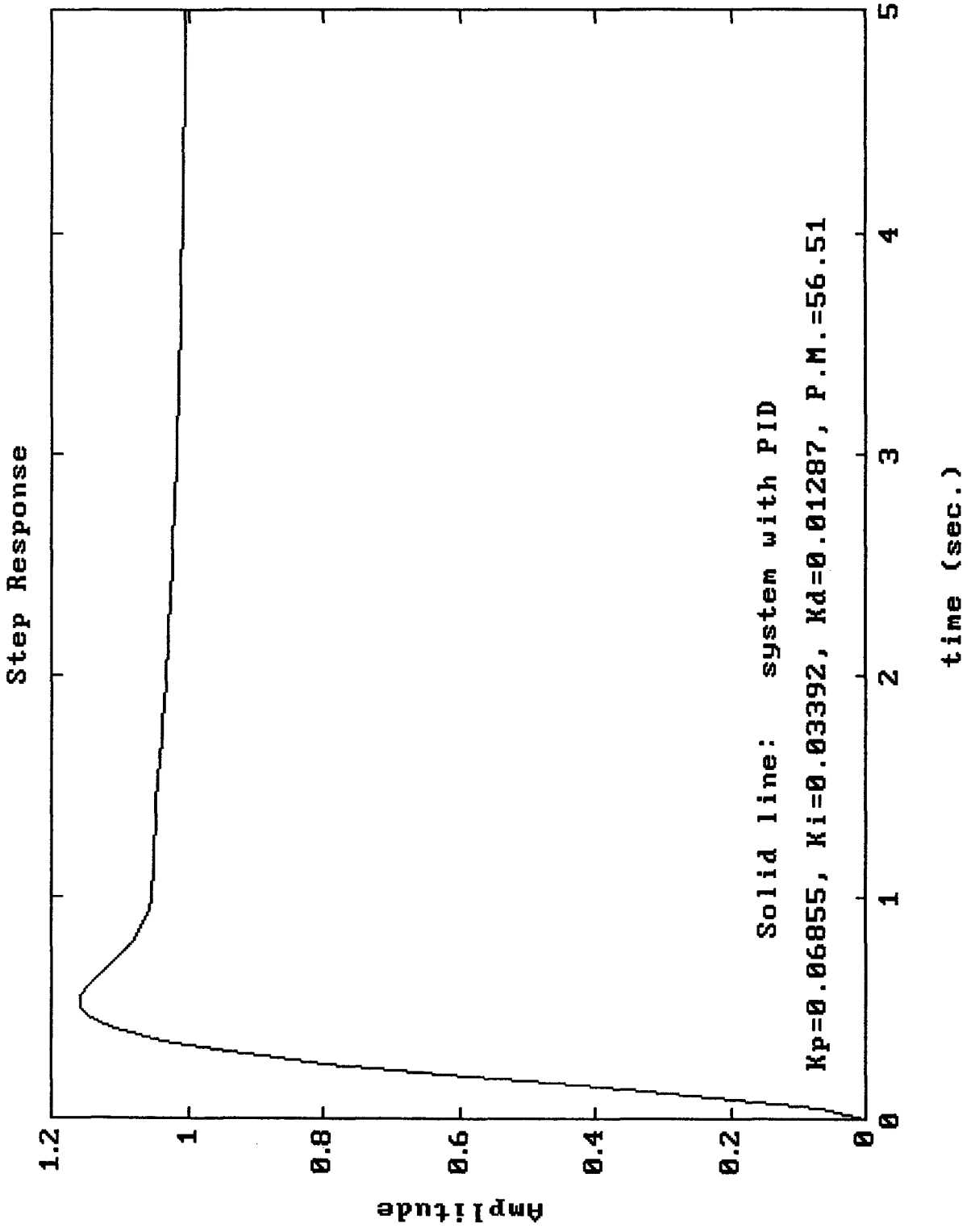


Figure 8: Actual Plot of **pid.m** Results with $Gp_1(s)$



outlined earlier, thus stabilizing the unstable system. The step response in figure 5 shows that although there is a considerable amount of overshoot and oscillation in the system, the system has been stabilized.

The magnitude and phase responses in figure 6 show the reduction in gain the PI controller produces, and the degradation of phase for lower frequencies by 90 degress, as expected and denoted earlier. The step response shows a marked improvement in stability over the PD controlled system, but a much larger rise time.

The step response in figure 7 shows the benefit of using the PID controller over the individual PD and PI controllers for this system. The improved stability given by the PI controller is also accompanied by the smaller rise time of the PD controller; the PID controller combines the advantages of both, as predicted and outlined earlier. By examining the PID magnitude and phase responses, the frequency domain effects of the PID on the system can be seen.

Finally, the output plot generated by the invocation of the **pid.m** function without lefthand output arguments in figure 8, gives identical results as the step response created using the results of the **pid.m** function with output arguments. Also note that since the input function was not stable, and since the PD controller did not meet the input specifications (to design a controller fo 55 degrees of phase margin), these plots were not generated. Thus, the functions **designpd.m**, **designpi.m**, and **pid.m** performed as expected using input function $Gp_1(s)$, and meet the design requirements.

The next function used for testing was:

$$Gp_2(s) = \frac{2500}{s\,(s+25)}$$

and was used to test the plot generation option when the **pid.m** function is called, and a PD design satisfies the input requirements. The desired input phase used was 45 degrees.

Figure 9: Actual Plot of **pid.m** Results with $Gp_2(s)$

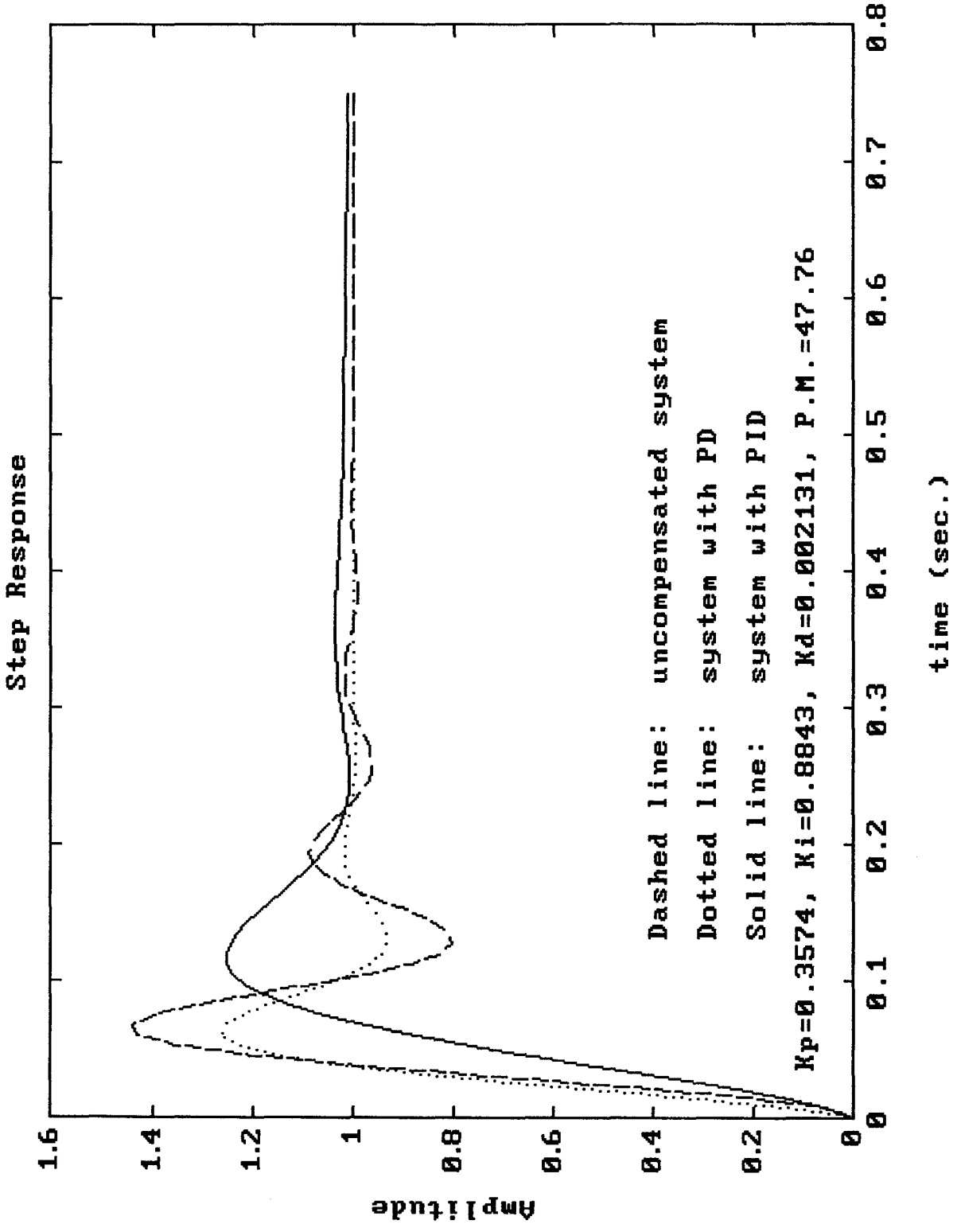
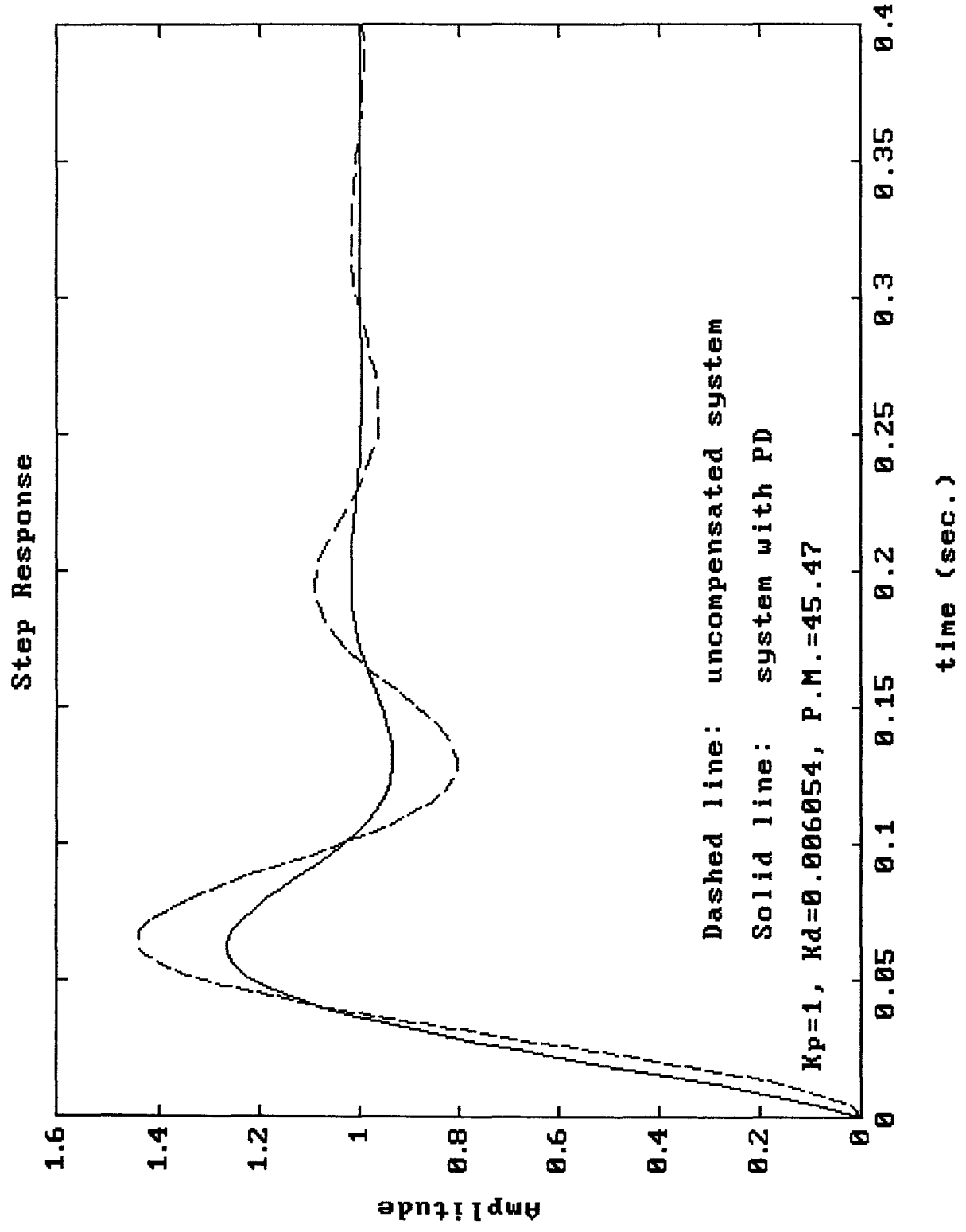


Figure 10: Actual Plot of **pid.m** Results with $Gp_2(s)$ (no PID)



Results:

	Computed			
Controller	Kp	Ki	Kd	Phase Margin
uncomp.	1	0	0	28.22
PD	1	0.0061	0	45.47
PID	0.3574	0.8843	0.0021	47.76

Table 2: Software Test Results Using $Gp_2(s)$

Figure 9 shows that all plots have been generated by responding "yes" to the "continue with PI design" prompt, as expected and as programmed. Note that the PD step response has decreased rise time and increased oscillation as compared to the PID step response, with equivalent overshoots, so the desired response is left up to the user. This is the purpose of generating both plots. Figure 10 is the resultant plot when the "continue" prompt is answered "no". Note that the PID response was not generated, the uncompensated and PD responses are identical to those in figure 9, and only the coefficients for a PD controller are returned. These results are in accordance to the software design and expectations.

The input transfer function:

$$Gp_3(s) = \frac{100}{s^2 + 10s + 100}$$

was used to test the error constant feature. In Kuo's problem, the controller is to be designed such that a ramp-error constant is 100, the maximum overshoot is less than 2%, and the rise time is less and 0.02 seconds. To meet the ramp error requirement, a PI or PID controller must be designed. For this example, the PID was designed, and the desired phase margin was specified as 180 degrees, since no phase requirements were given.

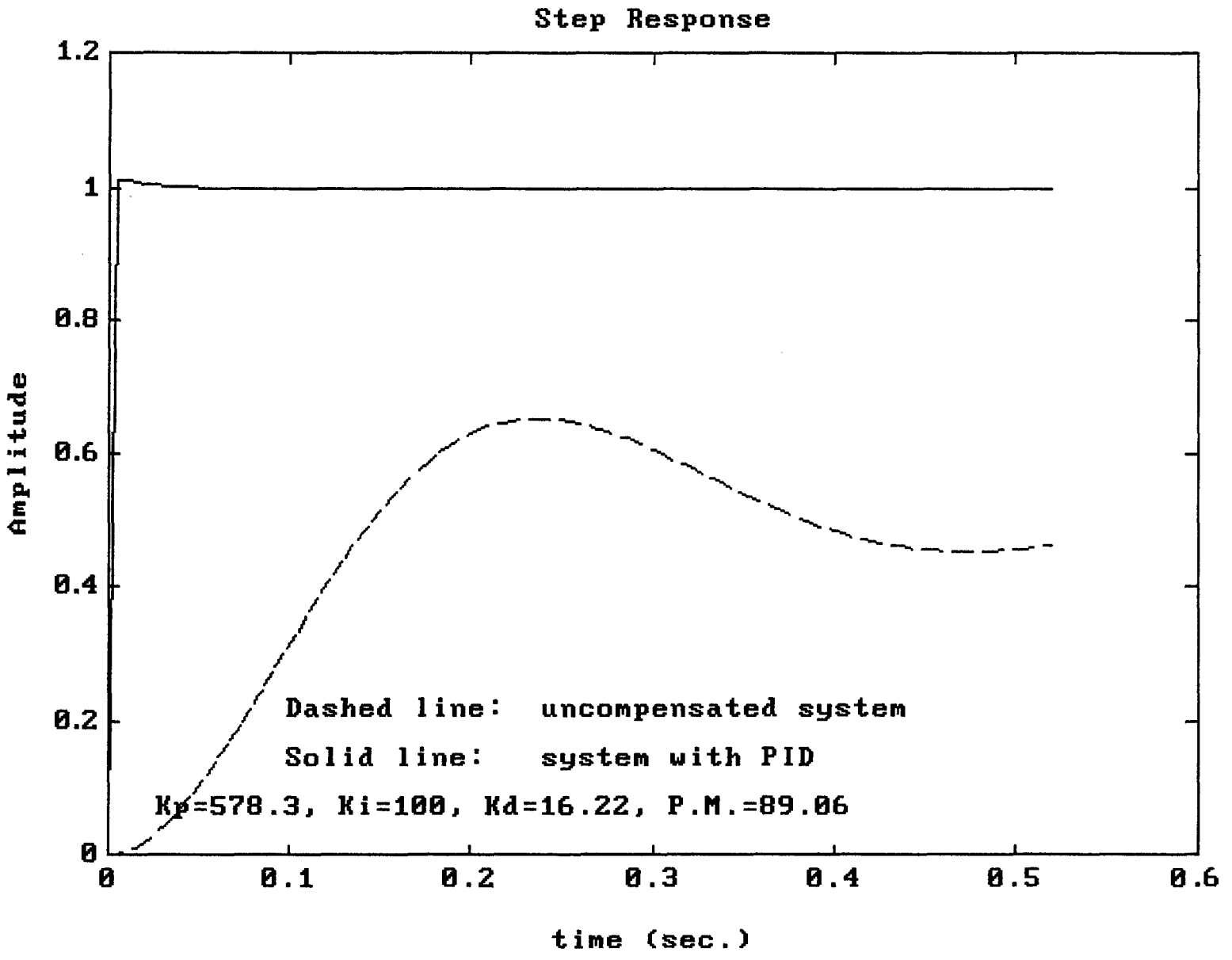


Figure 11: Actual Plot of **pid.m** Results with $Gp_3(s)$

Results:

	Computed			
Controller	Kp	Ki	Kd	Phase Margin
uncomp.	1	0	0	90.59
PID	578.3	100	16.22	89.06

Table 3: Software Test Results Using $Gp_3(s)$

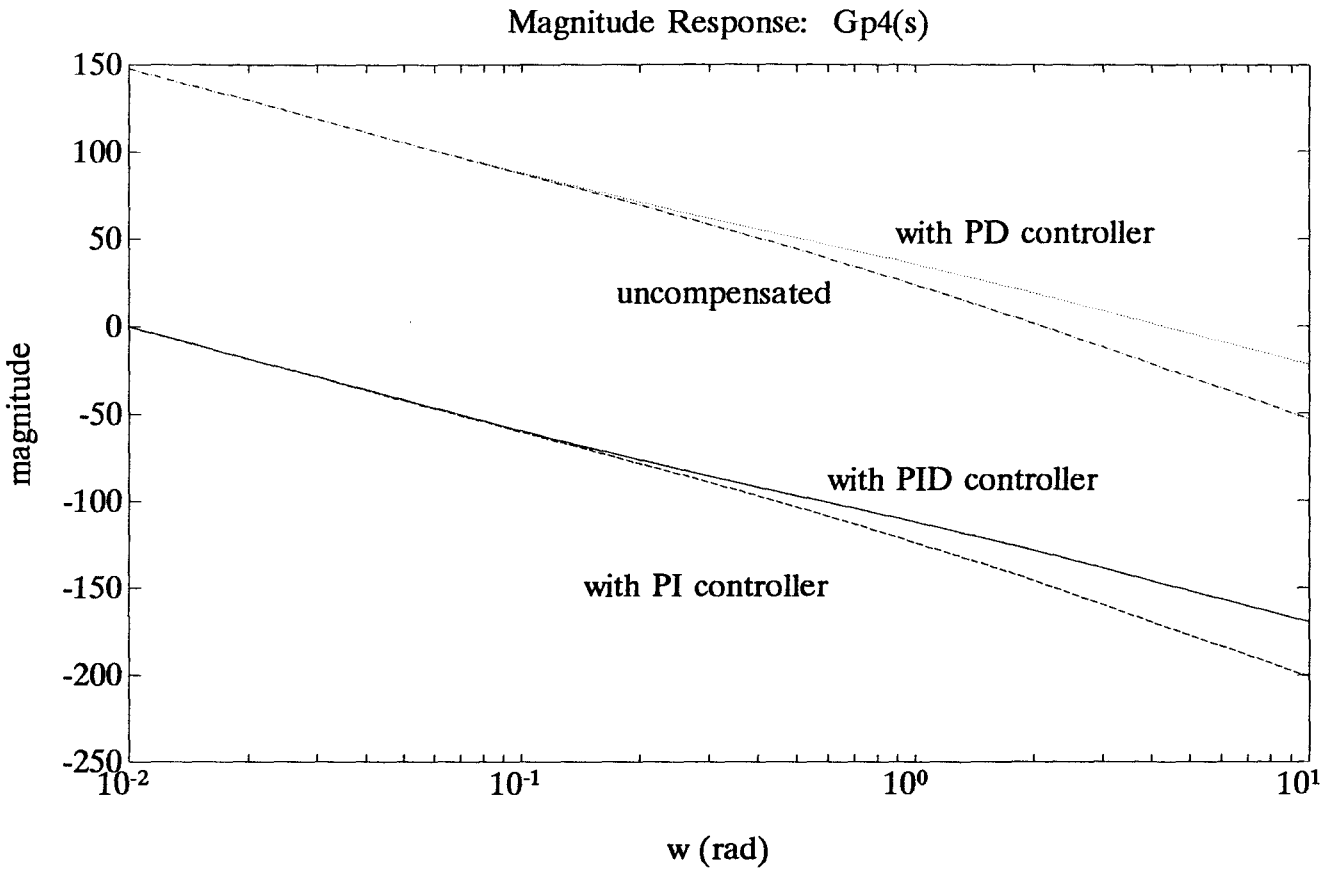
After generating the plot as in figure 11, analysis was performed on the plotted function to determine if the design met the requirements. This analysis could be done using the **ginput** function, which returns x-y coordinates of points picked off the graphics screen with a pointing device (such as a mouse), or more accurately, by using the actual plot data. Analysis performed on the plot data result in a rise time of less than 0.0027 seconds, and a maximum overshoot of 1.34%, both well below the design requirements. The error constant Kv = $\lim_{s>0}$ sGc(s) =100 with Ki =100, therefore all input requirements are met, and the error constant feature performs as designed.

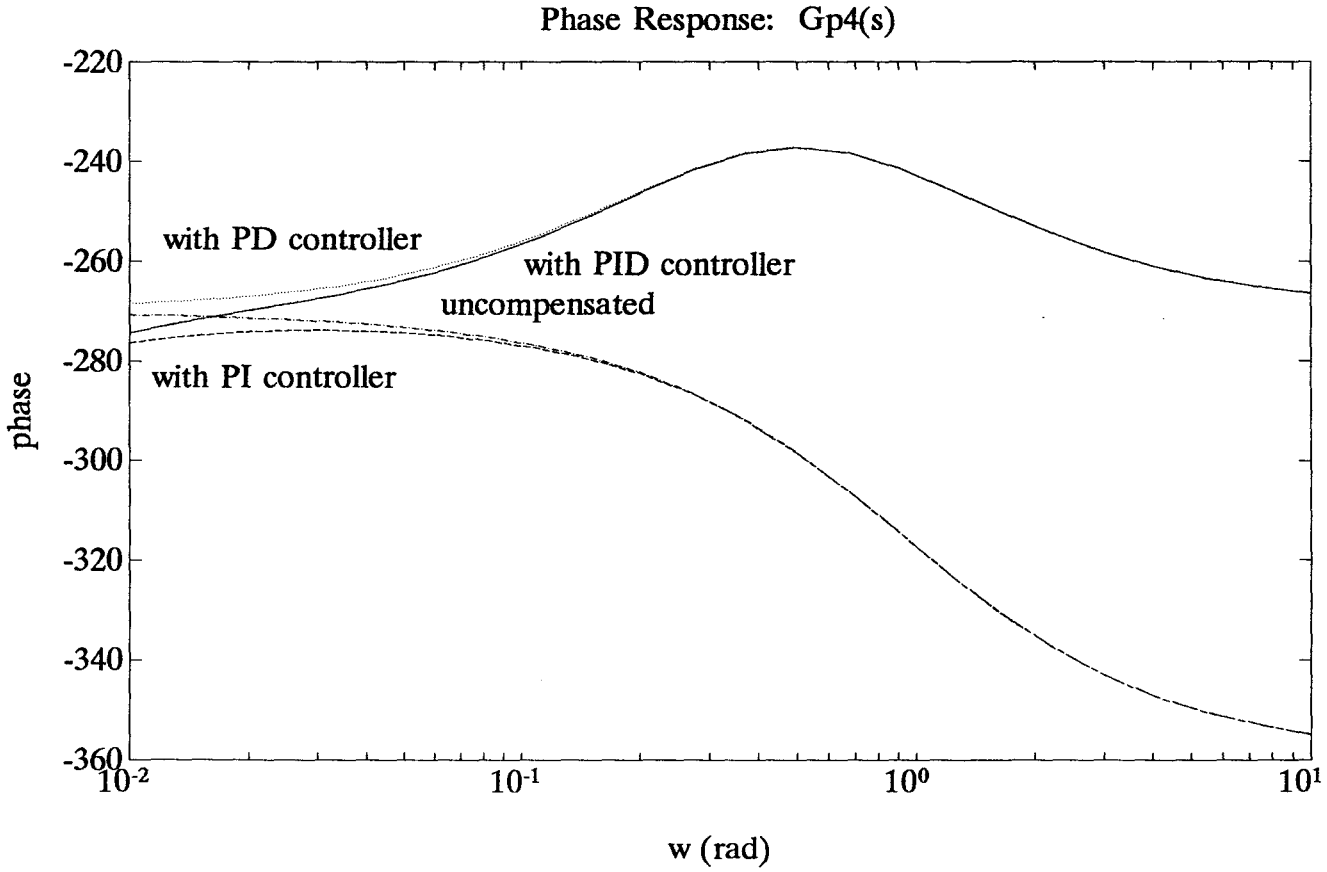
The final analysis performed used the transfer function:

$$Gp_4(s) = \frac{22.104}{s^3(s + 0.9216)}$$

to determine if the functions could handle an input function with several poles at the origin. Although the PD, PI, or PID functions cannot stabilize the system, the effects of the controllers on the frequency response of the uncompensated system should meet the expectations outlined. The desired phase input into the functions was the maximum 180 degrees.

Figure 12: Bode Plots of Systems with $Gp_4(s)$





Results:

	Computed			
Controller	Kp	Ki	Kd	Phase Margin
uncomp.	1	0	0	-156.4
PD	1	0	3.677	-81.52
PI	4.17E-8	4.17E-11	0	-96.38
PID	4.18E-8	4.17E-11	1.5E-7	-94.26

Table 4: Software Results Using $Gp_4(s)$

Table 4 clearly shows that although the PD controller is helpful in reducing negative phase margin, the PI and PID controller coefficients are impractical, and essentially zero, and therefore these controllers are impractical as well. This is to be expected, since the input transfer function is of order 4, and is a type 3, and any additional system poles would be unwanted. The magnitude and phase Bode plots in figure 12 graphically illustrate the impracticality; the PI controller offers little change over the uncompensated system, and the PID controller offers little change over the PD controlled system.

More importantly, the results in the phase margin column show there is no shift in phase by the function **bode.m**. If such a phase shift occurred, the phase margins would be shifted by 360 degrees, and would be positive values. Therefore, the phase correction in the software functions as expected, and as designed.

The tests documented above demonstrate that the software performs as designed, and meets the required design objectives.

Conclusion

By analyzing the results of the software designs, it is evident that PD, PI, and PID controllers can be successfully designed using computer simulation and evaluation techniques. The designs presented here use the MATLAB programming environment, in an effort to take full advantage of MATLAB's frequency analysis functions, and subsequently to provide a medium for full exploitation of the software's design results. Controller design is certainly not limited to this type of environment; controller design functions can be programmed using any language of any level, and any controller type can be designed using methods similiar to those presented here.

Other factors that can be included in controller design, and could certainly be included in these controller design functions, are provisions for time delay, sensitivity, multiple-input-multiple-output systems, systems represented in state-variable form, and non-linear systems. The MATLAB Control Systems Toolbox contains functions that convert systems in state-variable form to polynomial form, so any SISO system in state-variable form can be converted for use by the designed functions, and the designed controller could be converted to state-variable form as well. The control toolbox also includes functions that convert from digital domain to analog domain, and vice-versa, so that digital PD, PI, and PID controllers could be designed and analyzed using the software presented. Sections of non-linear systems can be represented by linear systems, and time-delays can be modeled in polynomial form, so the functions designed can manipulate these systems as well.

The analysis of the results of the designed functions is certainly not limited to the few examples mentioned. MATLAB includes a vast variety of data analysis functions, so that any desired analysis may be done. If an impulse response plot is more useful than a step response plot, it can be generated. In fact, MATLAB can generate any user-defined input response, and plot its output response. This makes MATLAB a very powerful analysis tool, and learning aid. It is for this reason that the controller design functions, specifically programmed to be designing tools and learning aids, were programmed and presented in this environment.

References

Bateson, Robert N. Introduction to Control System Technology. 4th ed. Toronto: MacMillan, 1993.

Ellis, George. Control System Design Guide. San Diego: Academic Press, 1991.

Kuo, Benjamin C. Automatic Control Systems. 6th ed. Englewood Cliffs, NJ: Prentice Hall, 1991.

Little, John N. and Alan J. Laub. Control System Toolbox (for use with MATLAB). User's Guide. Natick, MA: MathWorks, Inc., 1990.

Loberg, David. "Ball and Beam Position Control System." Design Project. DeKalb, IL: n.p., 1992.

Mathis, Bradley S. "Ball and Beam Position Control System." Honors Project. DeKalb, IL: n.p., 1993.

MATLAB. User's Guide. Natick, MA: MathWorks, Inc., 1991.

Van de Vegte, John. Feedback Control Systems. Englewood Cliffs, NJ: Prentice Hall, 1986.

Appendix 1: **designpd.m** - PD designing software

```
function [Kp,Kd,phaseout] = designpd(num,den,phasein)
%DESIGNPD - PD controller design
%   [KP,KD,PHASEOUT]=DESIGNPD(NUM,DEN,PHASEIN) calculates the coefficients
%   Kd and Kp, and output phase margin of:   Gc(s) = Kp + Kd*s, (see note).
%   for a PD controller in series with a plant with unity feedback,
%   yielding phase margin specified by PHASEIN. If the desired phase margin
%   is not attainable, or if PHASEIN is specified as 180 degrees, the
%   maximum phase margin attainable is returned. If the desired phase
%   margin is attainable, one or two values of Kd and Kp are returned,
%   depending on the nature of the transfer function. Vectors NUM and DEN
%   must contain the numerator and denominator coefficients of the open loop
%   plant transfer function in descending powers of s. Phasein must be a
%   scalar value expressed in degrees.
%
%   Note: It is assumed any error constant is incorporated into the plant
%   transfer function gain constant, and Kp is set to equal 1, for an actual
%   controller function of: Gc(s) = 1 + (Kd/Kp) s. See manual for details.
%
%   Warning: Results may be inaccurate for transfer functions with orders
%            greater than 5. This controller design function should be
%            used only as a guide.
%
%   See also: DESIGNPI and PID

%   Brad Mathis 4/23/94

%--------------------------------------
%  Input/output parameter check
%--------------------------------------

%  Check number of input arguments
error(nargchk(3,3,nargin));

%  Check number of output arguments
if ( nargout ~= 3 )
   error('Not enough output arguments')
   return;
end

%  Check validity of input arguments
[k,l] = size(num); [m,n] = size(den);
if (( k ~= 1 ) | ( m ~= 1 ))          % Check size of num, den
   error('Numerator and denominator inputs must be vectors in descending powers of s')
   return;
end
if ( l >= n )                         % Check order of num to den
   error('Order of numerator must be less than order of denominator')
   return;
end
```

```
[m,n] = size(phasein);                  % Check if phasein is scalar
if (( m ~= 1 ) | ( n ~= 1 ))
   error('Phase input must be a scalar in degrees')
   return;
end

%  Maximum value of phasein allowed = 180 degrees
if ( phasein > 180 )
   phasein = 180;
end

%-------------------------------------
%  Design PD controller
%-------------------------------------

disp('Designing PD controller...')

%-----------------
%  Compute Kp
%-----------------

%  Kp = 1 for no given error constant
kp = 1;

%-----------------
%  Compute Kd
%-----------------

%  Find minimum nonzero open loop transfer function root
tfroots = sort(abs(roots(den)));
for i = 1:1:length(tfroots)

   %  find system type
   systype = i-1;

   %  find minimum non-zero root
   if (tfroots(i) ~= 0)
      minroot = tfroots(i);
      break;
   end
end

%  Initialize loop parameters
phasem = zeros(1,500);                  % set magnitude using min.
magnitude = log10(kp/(minroot*10));        %     root value and kp
phasem(1) = -360;
phasem(2) = -359;
i = 2;
```

```
%  Loop to calculate Kd for optimum or desired phase margin
while (( phasem(i) < phasein ) & ( phasem(i) > phasem(i - 1) ) & fix(10*phasem(i)) ~=
fix(10*phasem(i-1) ) )
   kd_old = kd_current;                    % Save old kd
   magnitude = magnitude + 0.03;           % Increase magnitude
   kd_current = 10 ^ magnitude;            % Set kd value
   num2 = conv([kd_current kp],num);       % Multiply plant by PD
   w = freqint(num2,den,20);               % Find freq. interval
   w = logspace(log10(w(1)), log10(max(w)*100), 20);
   [mag,pha,w] = bode(num2,den,w);         % Find freq. response
   pha = pha - (360 * floor( (systype+2)/4 ) );  % Correct phase
   [Gm,phasem(i + 1),wcg,wcp] = margin(mag,pha,w);   % Check phase margin
   i = i + 1;
end

%  Save Kd, Kp value if optimum phase margin is met
if ( phasem(i) < phasem(i - 1) )
   phaseout(1) = phasem(i - 1);
   Kd(1) = kd_old; Kp(1) = kp;

%  Save Kd, Kp value if desired phase margin is met
else
   phaseout(1) = phasem(i);
   Kd(1) = kd_current; Kp(1) = kp;

   %  Loop to find second Kd for desired phase margin, if possible
   while (( phasem(i) > phasein ) & (fix(10*phasem(i)) ~= fix(10*phasem(i - 1))))
      kd_old = kd_current;                 % Save old kd
      magnitude = magnitude + 0.03;        % Increase magnitude
      kd_current = 10 ^ magnitude;         % Set kd value
      num2 = conv([kd_current kp],num);    % Multiply plant by PD
      w = freqint(num2,den,20);            % Find freq. interval
      w = logspace(log10(w(1)), log10(max(w)*100), 20);
      [mag,pha,w] = bode(num2,den,w);      % Find freq. response
      pha = pha - (360 * floor( (systype+2)/4 ) );  % Correct phase
      [Gm,phasem(i + 1),wcg,wcp] = margin(mag,pha,w);  % Check phase margin
      i = i + 1;
   end

   %  Save second Kd, Kp value for desired phase margin if it exists
   if (( phasem(i - 1) > phasein - 1 ) & ( phasem(i - 1) < phasein + 1 ))
      phaseout(2) = phasem(i - 1);
      Kd(2) = kd_old; Kp(2) = kp;
   end
end
```

Appendix 2: designpi.m - PI designing software

```
function [Kp,Ki,phaseout] = designpi(num,den,phasein,K)
%DESIGNPI - PI controller design
%   [KP,KI,PHASEOUT]=DESIGNPI(NUM,DEN,PHASEIN) calculates the coefficients
%   Kp and Ki, and output phase margin of:     Gc(s) = Kp + Ki/s,
%   for a PI controller in series with a plant with unity feedback,
%   yielding phase margin specified by PHASEIN. If the desired phase
%   margin specified by PHASEIN is not attainable, or if PHASEIN is
%   specified as 180 degrees, the maximum phase margin attainable is
%   returned. An optional steady-state error constant K can be specified by
%   [KP,KI,PHASEOUT] = DESIGNPI(NUM,DEN,PHASEIN,K). Vectors NUM and
%   DEN must contain the numerator and denominator coefficients of the open
%   loop plant transfer function in descending powers of s. Phasein must be
%   a scalar value expressed in degrees. K is an optional scalar value, and
%   it is assumed K is of type two greater than DEN (since PI adds a pole at
%   the origin). See manual for details and examples.
%
%   Warning: Results may be inaccurate for transfer functions with orders
%            greater than 5. This controller design function should be
%            used only as a guide.
%
%   See also: DESIGNPD and PID

%   Brad Mathis 4/23/94

%-------------------------------------
%  Input/output parameter check
%-------------------------------------

%  Check number of input arguments
error(nargchk(3,4,nargin));

%  Check number of output arguments
if ( nargout ~= 3 )
   error('Not enough output arguments')
   return;
end

%  Check validity of input arguments
[k,l] = size(num); [m,n] = size(den);     % Check size of num, den
if (( k ~= 1 ) | ( m ~= 1 ))
   error('Numerator and denominator inputs must be vectors in descending powers of s')
   return;
end
if ( l >= n )                          % Check order of num to den
   error('Order of numerator must be less than order of denominator')
   return;
end
```

```
[m,n] = size(phasein);                % Check if phasein is a scalar
if (( m ~= 1 ) | ( n ~= 1 ))
   error('Phase margin input must be a scalar in degrees')
   return;
end
if ( nargin == 4 )
   [m,n] = size(K);                   % Check if K is a scalar
   if (( m ~= 1 ) | ( n ~= 1 ))
      error('Error constant input must be a scalar')
      return;
   end
end

%   Maximum phasein allowed = 180 degrees
if ( phasein > 180 )
   phasein = 180;
end

%------------------------------
%   Design PI controller
%------------------------------

disp('Designing PI controller...')

%-------------------------------------------------
%   Compute Kp, Ki if error constant K is not given
%-------------------------------------------------

if ( nargin == 3 )

   %   Find system type
   tfroots = sort(abs(roots(den)));
   for i = 1:1:length(tfroots)

      %  Find system type
      systype = i - 1;

      if (tfroots(i) ~= 0)
         break;
      end
   end

   %   Compute frequency of desired phase margin
   [mag,phase,w]=bode(num,den);
   phase = phase - (360 * floor( (systype+2)/4 ) );   % Correct phase
   for i=1:1:length(phase)
      if ( (phase(i) + 180) < (phasein + 10) )  % Add 10 deg. for compen.
         wc = w(i);  mag1 = mag(i);
         break;
      end
   end
```

```
   %  Compute frequency of optimum phase margin if desired does not exist
   if ( wc == [] )
      [Y,I]=max(phase);
      wc = w(I);
   end

   %------------------
   %  Compute Kp
   %------------------

   Kp = 10 ^ (-abs(log10(mag1)));          % wc becomes new crossover

   %------------------
   %  Compute Ki
   %------------------

   Ki = Kp * (wc / 10);                    % Locate Ki 1 decade away

   %-------------------------
   %  Compute phaseout
   %-------------------------

   num=conv(num,[Kp Ki]);                  % Find new num with PI
   den=conv(den,[1 0]);                    % Find new den with PI
   [mag,phase,w]=bode(num,den);
   phase = phase - (360 * floor( (systype+3)/4 ) );   % Correct phase
   [gm,phaseout,wcg,wcp]=margin(mag,phase,w);

%-------------------------------------------------
%  Compute Kp, Ki if error constant K is given
%-------------------------------------------------
else

   %------------------
   %  Compute Ki
   %------------------

   %  Multiply plant function by 1/s  (PI =  (Ki + Kp*s) / s)
   den = conv(den,[1 0]);

   %  Find lowest order non-zero term in denominator
   for ( i=length(den):-1:1 )
      if ( den(i) ~= 0 )
         break;
      end
   end

   %  Find Ki
   ki = K * (den( i )) / (num(length(num)));

   %------------------
   %  Compute Kp
   %------------------
```

```
% Find minimum nonzero open loop transfer function root
tfroots = sort(abs(roots(den)));
for i = 1:1:length(tfroots)

   % Find system type
   systype = i - 1;

   % Find minimum nonzero root
   if (tfroots(i) ~= 0)
      minroot = tfroots(i);
      break;
   end
end

% Initialize loop parameters
phasem = zeros(1,500);
magnitude = log10(ki/(minroot*10));        % set magnitude using min.
phasem(1) = -360;                          %    root value and ki
phasem(2) = -359;
i = 2;

% Loop to calculate Kp for optimum or desired phase margin
while (( phasem(i) < phasein ) & ( phasem(i) > phasem(i - 1) ) & ( fix(10*phasem(i)) ~=
fix(10*phasem(i - 1) ) ) )
   kp_old = kp_current;                    % Save old kp
   magnitude = magnitude + 0.03;           % Increase magnitude
   kp_current = 10 ^ magnitude;            % Set kp value
   num2 = conv([kp_current ki],num);       % Multiply plant by PI*s
   w = freqint(num2,den,20);               % Find freq. interval
   w = logspace(log10(w(1)), log10(max(w)*100), 20);
   [mag,pha,w] = bode(num2,den,w);         % Find freq. response
   pha = pha - (360 * floor( (systype+2)/4 ) );  % Correct phase
   [Gm,phasem(i + 1),wcg,wcp] = margin(mag,pha,w);  % Check phase margin
   i = i + 1;
end

% Save Ki, Kp value if optimum phase margin is met
if ( phasem(i) < phasem(i - 1) )
   phaseout(1) = phasem(i - 1);
   Kp(1) = kp_old; Ki(1) = ki;

% Save Ki, Kp value if desired phase margin is met
else
   phaseout(1) = phasem(i);
   Kp(1) = kp_current; Ki(1) = ki;
```

```
      %   Loop to find second Kp for desired phase margin, if possible
      while (( phasem(i) > phasein ) & (fix(10*phasem(i)) ~= fix(10*phasem(i - 1))))
         kp_old = kp_current;              % Save old kp
         magnitude = magnitude + 0.03 ;    % Increase magnitude
         kp_current = 10 ^ magnitude;      % Set kp value
         num2 = conv([kp_current ki],num); % Multiply plant by PI*s
         w = freqint(num2,den,20);         % Find freq. interval
         w = logspace(log10(w(1)), log10(max(w)*100), 20);
         [mag,pha,w] = bode(num2,den,w);   % Find freq. response
         pha = pha - (360 * floor( (systype+2)/4 ) );  % Correct phase
         [Gm,phasem(i + 1),wcg,wcp] = margin(mag,pha,w); % Check phase mar.
         i = i + 1;
      end
      if (( phasem(i - 1) > phasein - 1 ) & ( phasem(i - 1) < phasein + 1 ))
         phaseout(2) = phasem(i - 1);
         Kp(2) = kp_old; Ki(2) = ki;
      end
   end
end
```

Appendix 3: pid.m - PID designing software

```
function [Kp,Ki,Kd,phaseout] = pid(num,den,phasein,K)
%PID - PID controller design
%   [KP,KI,KD,PHASEOUT]=PID(NUM,DEN,PHASEIN) calculates the coefficients
%   Kp, Ki, and Kd, and output phase margin of:   Gc(s) = Kp + Kd*s + Ki/s,
%   for a PID controller in series with a plant with unity feedback,
%   yielding phase margin specified by PHASEIN. If the desired phase margin
%   is not attainable, or if PHASEIN is specified as 180 degrees, the
%   maximum phase margin attainable is returned. If a PD controller meets
%   the input requirements, the user is prompted to continue with the PI.
%   An optional steady-state error constant K can also be specified by:
%   [KP,KI,KD,PHASEOUT]=PID(NUM,DEN,PHASEIN,K). If the function is called
%   without output parameters, no values are returned and a step response
%   plot is generated. Vectors NUM and DEN must contain the numerator and
%   denominator coefficients of the open loop plant transfer function in
%   descending powers of s. Phasein must be a scalar value expressed in
%   degrees. K is an optional scalar value, and it is assumed K is of type
%   two greater than DEN (since PID adds a pole at the origin). For more
%   control over design process, use DESIGNPD and DESIGNPI to design PD and
%   PI independently. See manual for details and examples.
%
%   Warning: Results may be inaccurate for transfer functions with orders
%            greater than 5. This controller design function should be
%            used only as a guide.
%
%   See also: DESIGNPI and DESIGNPD

%   Brad Mathis 4/23/94

%-------------------------------------
%  Input/output parameter check
%-------------------------------------

%  Check number of input arguments
error(nargchk(3,4,nargin));

%  Check number of output arguments
if ( nargout ~= 4 & nargout ~= 0 )
   error('Incorrect number of output arguments')
   return;
end

%  Check validity of input arguments
[k,l] = size(num); [m,n] = size(den);
if (( k ~= 1 ) | ( m ~= 1 ))          % Check size of num, den
   error('Numerator and denominator inputs must be vectors in descending powers of s')
   return;
end
```

```
if ( l >= n )                          % Check order of num to den
   error('Order of numerator must be less than order of denominator')
   return;
end
   [m,n] = size(phasein);              % Check if phasein is scalar
   if (( m ~= 1 ) | ( n ~= 1 ))
      error('Phase input must be a scalar in degrees')
      return;
   end
if ( nargin == 5 )
   [m,n] = size(K);                    % Check if K is scalar
   if (( m ~= 1 ) | ( n ~= 1 ))
      error('Error constant input must be a scalar')
      return;
   end
end

%  Maximum phasein allowed = 180 degrees
if ( phasein > 180 )
   phasein = 180;
end

%-----------------------------------
%  Design PD controller
%-----------------------------------

%  Design PD
[Kp1,Kd1,phaseout] = designpd(num,den,phasein);

%  If 2 values of Kp, Kd, phaseout, choose second
if ( length( Kd1 ) == 2 )
   temp = Kp1(2);
   Kp1 = temp;
   temp = Kd1(2);
   Kd1 = temp;
   temp = phaseout(2);
   phaseout = temp;
end

%  Incorporate PD controller into plant transfer function
numpd = conv(num,[Kd1 Kp1]);
denpd = den;

%  Check to see if PD controller meets phase requirements given no K.
if ( (phaseout > phasein) & (nargin == 3) )
   disp('PD controller meets input specifications (given no error constant).')
   proceed = input('Do you wish to continue with PI design (Y/N)? [Y] ','s');
   PD = 1;
%  Else conditions are not met; proceed = 'y'
else
   PD = 0;
   proceed = 'y';
end
```

```
%  Proceed with PI if proceed = 'y'
if ( isempty( proceed ) | proceed == 'y' | proceed == 'Y' )

   %----------------------------
   %  Design PI controller
   %----------------------------

   %  Design PI with no K specification given
   if ( nargin == 3 )
      [Kp2,Ki2,phaseout] = designpi(numpd,den,phasein);

   %  Design PI if K is given
   else
      [Kp2,Ki2,phaseout] = designpi(numpd,den,phasein,K);
   end

   %  If 2 values of Kp, Ki, choose second
   if ( length( Ki2 ) == 2 )
      Ki2 = Ki2(2);
      temp = Kp2(2);
      Kp2 = temp;
      temp = phaseout(2);
      phaseout = temp;
   end

%  Else do not design PI; Ki = 0; Kp = 1
else
   Ki2 = 0;
   Kp2 = 1;
   PD = 0;          %  PD controller does not exist (for plotting purp.)
end

%-------------------------------------
%  Find Kp, Ki, Kd
%-------------------------------------

%  Determine Kp, Ki, Kd
Kp = Kp2 + ( Kd1 * Ki2 );
Ki = Ki2;
Kd = Kd1 * Kp2;

%-------------------------------------
%  Plot option
%-------------------------------------

%  Plot step responses if number of output arguments = 0
if ( nargout == 0 )

   %  Determine stability of original function
   tfroots = sort(abs(roots(den)));
   for i = 1:1:length(tfroots)
```

```
   % find system type
   systype = i-1;

   if (tfroots(i) ~= 0)
      break;
   end
end
[mag,phase,w] = bode(num,den);
phase = phase - (360 * floor( (systype+2)/4 ) );   %phase correction
[gm,pm,wg,wp] = margin(mag,phase,w);

%  Determine stability of PD function
[mag,phase,w] = bode(numpd,den);
phase = phase - (360 * floor( (systype+2)/4 ) );   %phase correction
[gm,pmpd,wg,wp] = margin(mag,phase,w);

%  Determine transfer function with PID controller
if ( Ki ~= 0 )            %  New transfer function with PID
   numcon = conv(num,[Kd Kp Ki]);
   dencon = conv(den,[1 0]);
   ctype = 'PID';
   pid = 1;
else                      %  New transfer function with PD
   numcon = numpd;
   dencon = den;
   ctype = 'PD';
   pid = 0;
end

%  Close loop transfer functions
[numc,denc] = cloop(num,den);
[numpd,denpd] = cloop(numpd,denpd);
[numcon,dencon] = cloop(numcon,dencon);

%  Find plot window
if ( pm > 0 )
   [y,x,t] = step(numc, denc);
   [y,x,t2] = step(numcon,dencon);
   tmax = max(t);
   t2max = max(t2);
   if ( (t2max > tmax) & (t2max < (32 * tmax)) )
      x = tmax + (t2max - tmax)/2;
      t = 0:x/100:x;      %  X axis coordinates; 100 point plot
   elseif ( (t2max <= tmax) & (tmax < (32 * t2max)) )
      x = t2max + (tmax - t2max)/2;
      t = 0:x/100:x;      %  X axis coordinates; 100 point plot
   elseif ( t2max >= (32 * tmax) )
      t = 0:tmax/100:tmax;
   else
      t = 0:t2max/100:t2max;
   end
```

```
elseif ( (pm < 0) & (pmpd > 0) & (PD == 1) )
   [y,x,t] = step(numpd,denpd);
   [y,x,t2] = step(numcon,dencon);
   tmax = max(t);
   t2max = max(t2);
   if ( (t2max > tmax) & (t2max < (32 * tmax)) )
      x = tmax + (t2max - tmax)/2;
      t = 0:x/100:x;      %  X axis coordinates; 100 point plot
   elseif ( (t2max <= tmax) & (tmax < (32 * t2max)) )
      x = t2max + (tmax - t2max)/2;
      t = 0:x/100:x;      %  X axis coordinates; 100 point plot
   elseif ( t2max >= (32 * tmax) )
      t = 0:tmax/100:tmax;
   else
      t = 0:t2max/100:t2max;
   end
else
   [y,x,t] = step(numcon,dencon);
   tmax = max(t);
   t = 0:tmax/100:tmax;    %  X axis coordinates; 100 point plot
end

%--------------------------
%   Plot step responses
%--------------------------
clg
axis('normal')

%   If uncontrolled system is stable, plot
if ( pm > 0 )
   [yunc,x,t] = step(numc,denc,t);
else
   yunc = [];
end

%   If PD controlled system exists and is stable, plot
if ( (pmpd > 0) & (PD == 1) )
   [ypd,x,t] = step(numpd,denpd,t);
else
   ypd = [];
end

%   Plot PID
[ypid,x,t] = step(numcon,dencon,t);

%   Plot functions
y = [ypid'; yunc'; ypd'];
plot(t,y,'w');
```

```
    %  Title and key
    title('Step Response');
    xlabel('time (sec.)');
    ylabel('Amplitude');
    if ( pid == 0 )
       str = ['Kp=',num2str(Kp),', Kd=',num2str(Kd),', P.M.='];
       str = [str,num2str(phaseout)];
    else
       str = ['Kp=',num2str(Kp),', Ki=',num2str(Ki),', Kd=',num2str(Kd)];
       str = [str,', P.M.=',num2str(phaseout)];
    end
    text(0.15,0.15,str,'sc');
    text(0.25,0.2,['Solid line:   system with ',ctype],'sc');
    loc = 0.25;
    if ( (pm > 0) & (PD == 1) )
       text(0.25,loc,'Dotted line:  system with PD','sc');
       loc = loc + 0.05;
    end
    if ( (pm < 0) & (PD == 1) )
       text(0.25,loc,'Dashed line:  system with PD','sc');
    end
    if ( pm > 0 )
       text(0.25,loc,'Dashed line:  uncompensated system','sc');
    end
end
```

PD, PI, PID Control Design

for use with MATLAB

Brad Mathis

PD, PI, PID Control Design

for use with MATLAB

User's Guide

May 1, 1994

By Brad Mathis

College of Engineering and Engineering Technology
Northern Illinois University
DeKalb, IL 60115

Table of Contents

designpd

Purpose

Design a PD controller for a unity feedback SISO system.

Synopsis

[kp,kd,phaseout] = designpd(num,den,phasein)

Description

designpd designs a proportional-derivative (PD) controller for a SISO LTI system with unity feedback, where the controller is in series with the plant, or controlled process. The PD controller is used in an effort to improve the relative stability of a system, increase system damping, reduce overshoot and rise time, by increasing system bandwidth and adding phase.

[kp,kd,phaseout] = designpd(num,den,phasein) evaluates the coefficients **kp** and **kd** of the controller transfer function

$$Gc(s) = Kp + Kd\ s$$

and computes the resulting phase margin **phaseout** when Gc(s) is in series with the plant transfer function. The plant transfer function Gp(s) is described by **num** and **den**, where **num** and **den** are vectors of polynomial coefficients in descending powers of s. The scalar **phasein** is the desired phase margin of the controlled process in degrees. If the desired phase margin is not attainable, or if **phasein** is specified as 180 degrees, the controller will be designed for the maximum phase margin attainable.

It is assumed any error constant is incorporated into the plant transfer function gain constant, and therefore **designpd** designs a PD controller with Kp = 1. The actual controller function being designed becomes

$$Gc(s) = 1 + (Kd/Kp)\ s$$

and the values resultant from the function call must then be multiplied by Kp, which is found from the error constant information.

designpd

Since the PD controller is being designed for a desired phase margin, and if the desired phase margin is attainable, the possibility exists that two values of Kp and Kd may be found. Both values are returned, so **kp**, **kd** and **phaseout** become 2 element matrices, and the corresponding values have identical indices.

Example:

Given the open loop plant transfer function

$$Gp(s) = \frac{100}{s(1 + 0.1s)(1 + 0.2s)},$$

design a PD controller to stabilize the system.

Since no error constant is given, it is assumed any error constant is incorporated into Gp(s). To design for maximum stability, specify **phasein** as 90 degrees.

```
phasein = 90;
num = [100];
den = conv([1 0],conv([0.1 1],[0.2 1]));
[kp,kd,phaseout] = designpd(num,den,phasein);
```

This yields **kp** = 1, **kd** = 0.2094, and **phaseout** = 18.6693. To plot the step response of the controlled system:

```
num = conv(num,[kd kp]);
[num,den] = cloop(num,den);
step(num,den);
```

Algorithm

designpd is an M-file, and uses an iterative process incorporating frequency domain analysis.

Limitations

Results may be inaccurate for transfer functions with orders greater than 5.

This controller design function should be used only as a guide.

Diagnostics

If there is a system pole on the $j\omega$ axis the warning message

Matrix is singular to working precision.

may occur. This message may also occasionally arise during the calculation of the frequency interval.

See Also

designpi, pid

References

Kuo, Benjamin C. Automatic Control Systems, sixth ed. Englewood Cliffs, NJ: Prentice Hall, 1991.

designpi

Purpose

Design a PI controller for a unity feedback SISO system.

Synopsis

[kp,ki,phaseout] = designpi(num,den,phasein)
[kp,ki,phaseout] = designpi(num,den,phasein,K)

Description

designpi designs a proportional-integral (PI) controller for a SISO LTI system with unity feedback, where the controller is in series with the plant, or controlled process. The PI controller is used in an effort to improve the relative stability of a system, increase system damping, and reduce overshoot by reducing system gain beyond a critical frequency.

[kp,kd,phaseout] = designpi(num,den,phasein) evaluates the coefficients **kp** and **ki** of the controller transfer function

$$Gc(s) = Kp + Ki / s$$

and computes the resulting phase margin **phaseout** when Gc(s) is in series with the plant transfer function. The plant transfer function Gp(s) is described by **num** and **den**, where **num** and **den** are vectors of polynomial coefficients in descending powers of s. The scalar **phasein** is the desired phase margin of the controlled process in degrees. If the desired phase margin is not attainable, or if **phasein** is specified as 180 degrees, the controller will be designed for the maximum phase margin attainable.

[kp,kd,phaseout] = designpd(num,den,phasein,K) designs the PI controller to meet a steady state error requirement specified by the error constant scalar **K**. It is assumed **K** is of two types greater than **den** (since the PI controller adds a pole at the origin). For example, if **den** is type 1 (one pole at origin), **K** is a parabolic-error constant. If **K** is not specified, it is assumed the controller may be designed without regard to any error constant.

designpi

Since the PI controller is being designed for a desired phase margin, and if the desired phase margin is attainable, the possibility exists that two values of Kp and Ki may be found. Both values are returned, so **kp**, **ki** and **phaseout** become 2 element matrices, and the corresponding values have identical indices.

Example:

Given the open loop plant transfer function

$$Gp(s) = \frac{100}{s\,(1 + 0.1s)(1 + 0.2s)}\ ,$$

design a PI controller to provide 45 degrees of phase margin.

Since no error constant is given, it is assumed any error constant is incorporated into Gp(s). To design for a phase margin of 45 degrees, specify **phasein** as 45 degrees.

```
phasein = 45;
num = [100];
den = conv([1 0],conv([0.1 1],[0.2 1]));
[kp,ki,phaseout] = designpi(num,den,phasein);
```

This yields kp = 0.0261, ki = 0.0060, and phaseout = 46.2966. To plot the step response of the controlled system:

```
num = conv(num,[ki kp]);
den = conv(den,[1 0]);
[num,den] = cloop(num,den);
step(num,den);
```

Algorithm

designpi is an M-file, and uses an iterative process incorporating frequency domain analysis.

Limitations

Results may be inaccurate for transfer functions with orders greater than 5.

This controller design function should be used only as a guide.

Diagnostics

If there is a system pole on the $j\omega$ axis the warning message

Matrix is singular to working precision.

may occur. This message may also occasionally arise during the calculation of the frequency interval.

See Also

designpd, pid

References

Kuo, Benjamin C. Automatic Control Systems, sixth ed. Englewood Cliffs, NJ: Prentice Hall, 1991.

pid

Purpose

Design a PID controller for a unity feedback SISO system.

Synopsis

[kp,ki,kd,phaseout] = pid(num,den,phasein)
[kp,ki,kd,phaseout] = pid(num,den,phasein,K)

Description

pid designs a proportional-integral-derivative (PID) controller for a SISO LTI system with unity feedback, where the controller is in series with the plant, or controlled process. The PID controller is used in an effort to improve the relative stability of a system, by combining the advantages of PD and PI controllers.

[kp,ki,kd,phaseout] = pid(num,den,phasein) evaluates the coefficients **kp**, **ki** and **kd** of the controller transfer function

$$Gc(s) = Kp + Ki / s + Kd\, s$$

and computes the resulting phase margin **phaseout** when Gc(s) is in series with the plant transfer function. The plant transfer function Gp(s) is described by **num** and **den**, where **num** and **den** are vectors of polynomial coefficients in descending powers of s. The scalar **phasein** is the desired phase margin of the controlled process in degrees. If the desired phase margin is not attainable, or if **phasein** is specified as 180 degrees, the controller will be designed for the maximum phase margin attainable.

If, when designing the PID controller, the PD portion of the controller satisfies the phase requirements that must be met, a user prompt will be generated as to whether or not the function should continue to design the PI portion. If the user answers "no" to the prompt, Ki = 0, and the coefficients Kp and Kd are given for a PD controller. If the user answers "yes", the function continues to design the PI portion, and the coefficients Kp, Ki, and Kd will be generated.

pid

[kp,ki,kd,phaseout] = pid(num,den,phasein,K) designs the PID controller to meet a steady state error requirement specified by the error constant scalar **K**. It is assumed **K** is of two types greater than **den** (since the PID controller adds a pole at the origin). For example, if **den** is type 1 (one pole at origin), **K** is a parabolic-error constant. If **K** is not specified, it is assumed the controller may be designed without regard to any error constant.

When invoked without left-hand arguments, the function pid generates a step-response plot of the controlled system, as well as a plot of the uncontrolled system (if stable), and the PD controlled system (if it meets the phase margin requirements), along with the coefficients Kp, Ki, Kd and the controlled system phase margin.

For more user control over the PID design process, it is suggested the user design the PID controller using the separate functions **designpd** and **designpi**.

Example:

Given the open loop plant transfer function

$$Gp(s) = \frac{100}{s\,(1 + 0.1s)(1 + 0.2s)}\ ,$$

design a PID controller to provide 45 degrees of phase margin.

Since no error constant is given, it is assumed the controller may be designed without regard to any error constant. To design for 45 degrees of phase margin, specify **phasein** as 45 degrees.