STARS

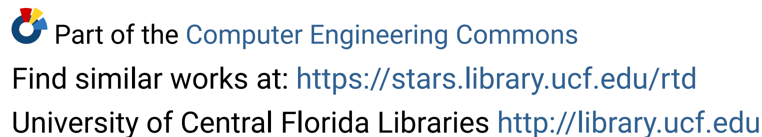
University of Central Florida
STARS

Retrospective Theses and Dissertations

1986

Root Locus Plotter for a Dual Tank System Under Feedback Control

John M. Decatrel
University of Central Florida



Part of the Computer Engineering Commons
Find similar works at: https://stars.library.ucf.edu/rtd
University of Central Florida Libraries http://library.ucf.edu

This Masters Thesis (Open Access) is brought to you for free and open access by STARS. It has been accepted for inclusion in Retrospective Theses and Dissertations by an authorized administrator of STARS. For more information, please contact STARS@ucf.edu.

STARS Citation

Decatrel, John M., "Root Locus Plotter for a Dual Tank System Under Feedback Control" (1986). *Retrospective Theses and Dissertations*. 4897.
https://stars.library.ucf.edu/rtd/4897

University of Central Florida
STARS
Showcase of Text, Archives, Research & Scholarship

ROOT LOCUS PLOTTER FOR A DUAL TANK SYSTEM UNDER FEEDBACK CONTROL

BY

JOHN MARK DECATREL
B.S.E., University of Central Florida, 1984

RESEARCH REPORT

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Engineering
in the Graduate Studies Program
of the College of Engineering
University of Central Florida
Orlando, Florida

Summer Term
1986

ABSTRACT

A root locus graphics routine was written in Turbo Pascal for the analysis and design of a linearized dual tank control system. The routine is a subprogram to be incorporated with an editor written by L. Fadden. This editor allows for the saving and changing of parameters to the system.

The dual tank system is a good example for classical feedback control analysis. A brief description of the process and system is presented. The system may be described by linearized differential and algebraic equations. From these, a characteristic equation is derived, which gives rise to the root locus. The root locus is a plot of the poles of the closed loop system. Poles or roots of the characteristic equation are found using the Lin-Bairstow algorithm. This method may be used to solve for the zeros of an nth degree polynomial.

The root locus plotter was exercised by attempting to optimally tune the system's controller. Corroboration of the results was provided by step response plots from the TUTSIM simulation program.

Minor modifications allow the root locus plotter to run without the editor. Graphics subroutines are provided by

the Turbo Graphix Toolbox. When run under the editor, the plotter is one interactive design module of the dual tank system analysis and design program. The subprogram was designed principally for user ease, error checking, and effective graphics.

ACKNOWLEDGEMENTS

Thanks to:

Dr. Klee

Dr. Matthews

Dr. Linton

Dr. Bauer

Regina

TABLE OF CONTENTS

LIST OF FIGURES vi

INTRODUCTION 1

Chapter
1. A DUAL TANK SYSTEM 3

2. THE ROOT LOCUS 10

3. THE LIN-BAIRSTOW ROOT SOLVING ALGORITHM 16
   Advantages of the Method
   The Algorithm
   Calculations from Polynomial Coefficients

4. PROGRAM DESCRIPTION 22

5. DISCUSSION OF THE RESULTS 29

6. SUMMARY AND CONCLUSION 44

Appendices
I. ADDITIONAL FIGURES 46

II. PROGRAM SOURCE CODE LISTING 53

III. "INCLUDE" FILES - NOTES 78

LIST OF REFERENCES 81

LIST OF FIGURES

1. Schematic Diagram of Dual Tank System 4

2. Block Diagram of Dual Tank System 6

3. Structure Blocks of Dual Tank System For Tutsim Simulation 6

4. Flow Chart of Root Solving Method 18

5. Flow Chart for Root Locus Program Numerical Routines 23

6. Flow Chart for Root Locus Program Graphics Routines 24

7. Typical Input Screen - Controller Parameters Select 27

8. Reaction Curve Method for Tuning the Controller . . 31

9. Optimally Tuned Controller Using Proportional Control Only 32

10. Step Response for P-I Controller and Proportional Controller Tuned Optimally by Reaction Curve Method 33

11. Optimally Tuned Controller Using Proportional-Integral Control 36

12. Optimally Tuned Controller Using Proportional-Integral-Derivative Control 38

13. Step Response for P-I-D Controller Tuned Optimally By Reaction Curve Method 39

14. Step Response for P-I Controller Gain Set Higher Than Recommended 40

15. Controller Set for System at Marginal Stability . . 41

16. Step Response for Marginally Stable System 43

17. System Under Proportional Control for Very High Gain 47

18. System with Gain Slightly Above Breakaway 48

19. Step Response for System at Breakaway. Controller Using Proportional Gain Only 49

20. System Under P-D Control. Plot Using Fixed Step Size 50

21. System Under P-D Control. Plot Using Variable Step Size 51

22. Sample Matlab Input and Output 52

INTRODUCTION

This computer program was developed as a module to be run with a simulation program by Leon Fadden (1986). It is a design routine which draws from the linearized model of a dual tank fluid system. With minor modification the module can run alone. Both programs are in support of a manuscript on system analysis being written by Dr. Harold Klee (University of Central Florida).

The root locus design tool presented herein allows for prediction of system stability, response characteristics, and aids in optimum tuning of a proportional-integral-derivative (P-I-D) controller. The root solving method used is based upon the Lin-Bairstow algorithm, which is good for any order polynomial. However, the two tank system under P-I-D control gives rise to a maximum third order characteristic polynomial equation. This algorithm was chosen because it is well known in numerical methods, converges rapidly, and is readily understood. If the system is modified for greater accuracy, or a more complex controller is incorporated, the characteristic equation may become higher order. This program would, then, still be useful. The reader can apply the root solving part of the program for other problems which contain higher order polynomials.

The program was written in Turbo Pascal because of the language's power, readability, and current popularity. High resolution monochrome graphics were realized with the aid of the Turbo Graphix Toolbox. Minor modifications were made to the Toolbox in order to obtain enhanced results for this particular application. Some necessary system time response graphs were obtained using the TUTSIM simulation program (Applied i 1985).

CHAPTER 1 A DUAL TANK SYSTEM

The system under investigation consists of a process which has two fluid holding tanks interconnected by a pipe (Klee 1986). Such a process might be part of a chemical batch production unit, or a flow regulating unit for the coolant of a power plant. The principal components to be analyzed in a simplified model of the system (Figure 1) are described as follows.

A constant displacement motor driven pump impels the inlet fluid into the first tank. Both tanks are unpressurized, i.e., open to the atmosphere. The inter-tank flow is a function of the pres-sure head of both tanks. It is assumed that the bottoms of the tanks as well as the inter-tank pipe are at ground reference level. This pipe has an adjustable valve which may be considered a load variable (θ_1) or disturbance input.

For design and analysis purposes we may let the two tank areas vary between simulation runs. At the outlet of the second tank is a discharge pipe with another hand actuated valve (θ_2) at some height above the reference. The discharge flow is a function of the tank 2 fluid level as well as the outlet valve's opening position. We also include some direct disturbance flow (F_L) to tank 2. This

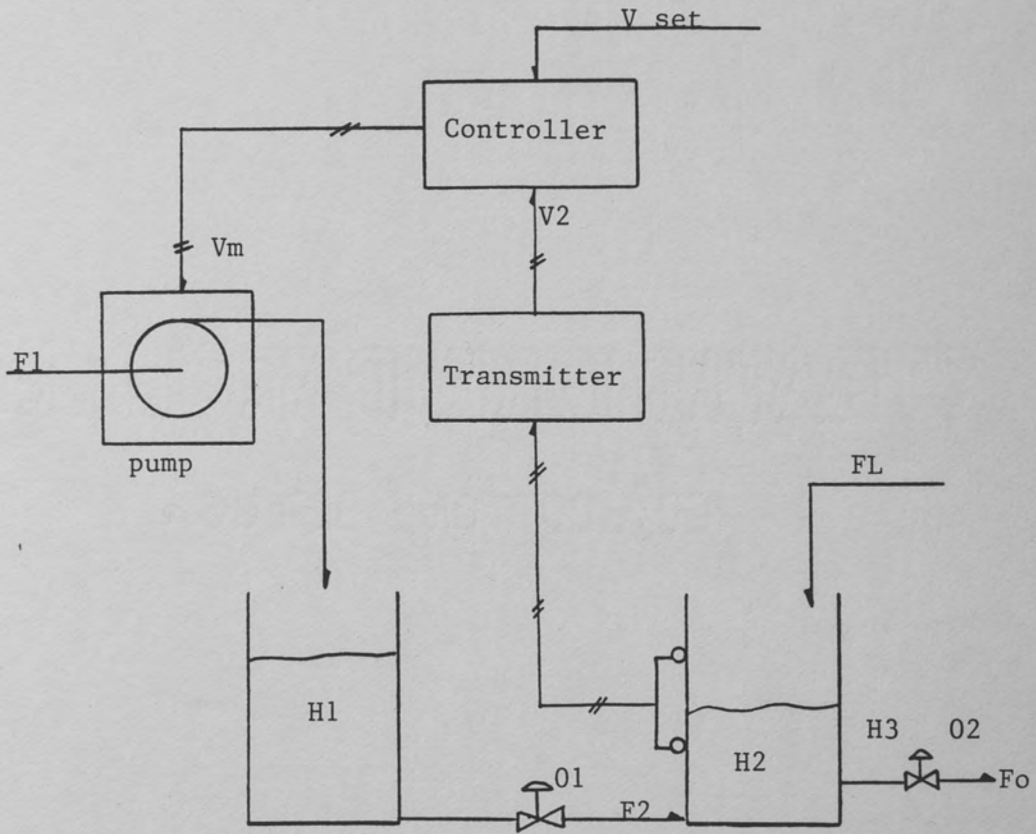


Figure 1 . Schematic Diagram of Dual Tank System

will have some impact upon the system equations, however, the three load variables Θ_1, Θ_2, and F_L will not influence the root locus of the system.

The components described so far represent an uncontrolled process, which is rarely useful in engineering operations. We can include certain other components which allow an operator to control some aspect of the process, e.g., the height of tank 2. A sensing device or transmitter is required to detect the height of the tank. The transmitter converts the height of fluid in the tank into an electrical signal (voltage), and provides an amplified signal to a controller unit (figures 2 and 3).

The controller considered herein is of the proportional-integral-derivative (P-I-D) type such as is commonly found in practice. It operates upon an error signal, i.e., the difference between some reference height and the height returned by the transmitter. Heights are first converted to analog electrical signals which can be recognized by the controller. Depending upon parameters set to adjust controller action, a voltage signal will be sent to the pump's motor. The inlet flow through the pump is considered to be a manipulated or controlled variable. Common configurations of controller parameters and their characteristics are as follows. For proportional control only, the response for this particular system is second

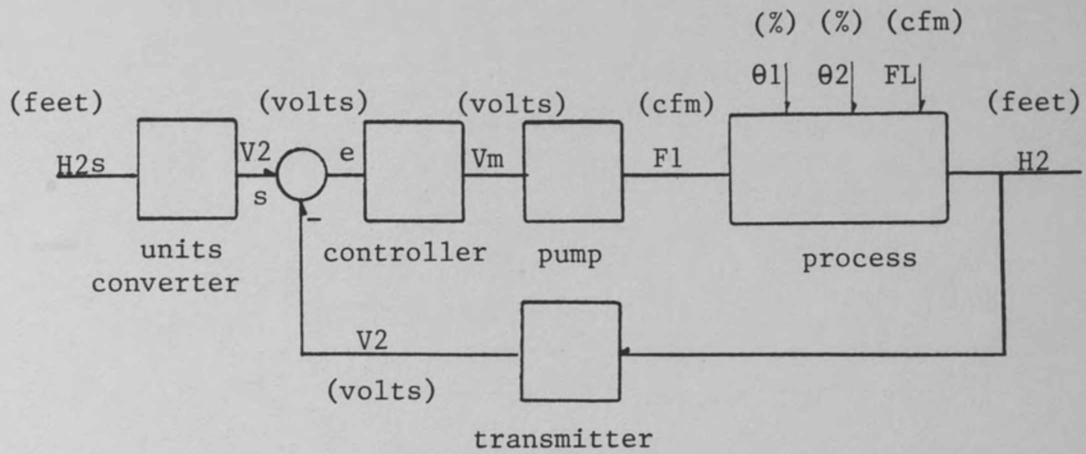


Figure 2. Block Diagram of Dual Tank System

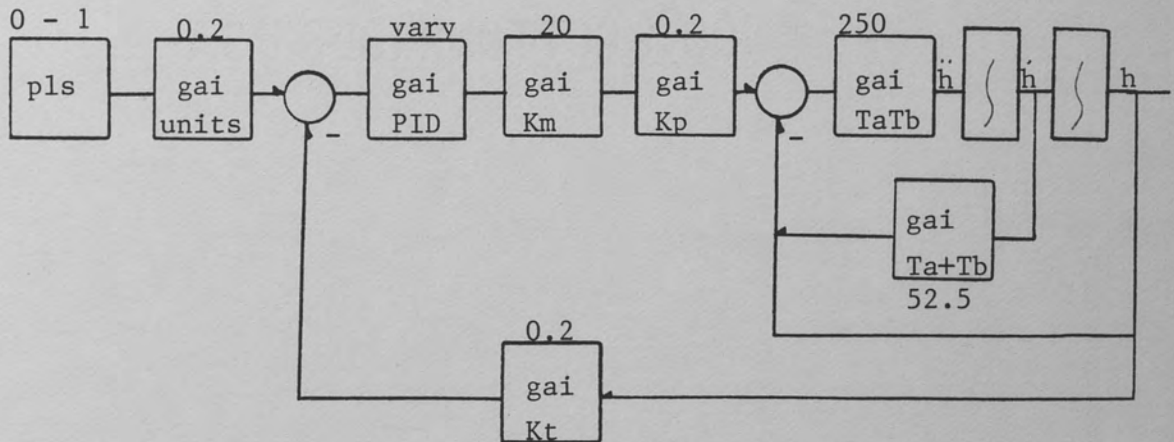


Figure 3. Structure Blocks of Dual Tank System for TUTSIM Simulation

order. That is, the Laplace transformed transfer function has a second order characteristic polynomial in the denominator. This implies that the system may be overdamped or underdamped. An increase in gain speeds up the response and pushes the system closer towards oscillation. The major drawback to proportional control is that the steady state step response is offset from the changed set point (servo action). Using the final value theorem upon the tranfer function, the step response is

$$r = \frac{K_c \cdot c}{1 + K_c} \quad [1]$$

where c is the magnitude of the step change in input and K_c is the controller gain. Offset is obvious from Eq. [1] above.

The addition of integral action eliminates steady state offset, however response speed is reduced (Weber 1973). The open loop transfer function acquires an additional pole and the system becomes third order. Setting controller parameters such that the proportional gain is low and integral action low (long reset time), the system behaves like a second order system. As integral action increases the system becomes more sluggish. Increasing the gain adds oscillations, and the system tends towards instability.

If derivative action is added the system remains third order, however response speed is improved. The system

becomes more stable allowing for a higher maximum gain and shorter reset time.

Changes to controller modes, e.g., proportional to proportional-integral, are not additive. This makes tuning the controller for optimum response a complex task. Certain methods have been developed for achieving this, including the Ziegler-Nichols reaction curve method and the continuous cycling method (Weber 1973). Each method has its drawbacks. The most common method of tuning a controller in practice is by operator trial and error. The root locus gives some insight as to what the response will be for various controller settings. This alleviates the problem of experimenting on a real process, which may be time consuming, expensive, and potentially disastrous.

The system model used with this design tool must be linear. The real system undoubtedly incorporates many non-linearities. By "linear" it is meant that the response to the sum of two signals is the same as the sum of the responses to each signal input. A linearized model is usually valid for relatively small changes about a set of design conditions. A model type (linear or non-linear) may be identified by the nature of its describing differential and algebraic equations.

Design conditions are found by setting all external inputs to desired values (Klee 1986). Then the internal,

dependent variables may be found by taking the system differential equations and rewriting them for steady state behavior. The resulting values determine the quiescent operating point about which new equations for the linearized model may be developed.

CHAPTER 2 THE ROOT LOCUS

The root locus is a graphical path drawn on the complex s-plane. Each point along the path indicates a pole of the closed loop (Laplace domain) transfer function for a system with fixed controller parameters. As a controller parameter is perturbed (typically the controller gain) the poles of the system transfer function change. A pole is a value of s on the complex plane which causes the denominator of the transfer function to go to zero, hence it causes the transfer function to go to infinity.

When set equal to zero, the denominator is called the characteristic equation of the system. The following discussion shows how the characteristic equation arises from time domain, linearized model equations. Whether or not load disturbance variables are introduced the characteristic equation does not change. For simplicity, load variables are assumed to be fixed at system design conditions, hence they do not appear in the following equations which use deviation variables.

Starting with the process equations (Klee 1986), by conservation of mass for tank 1 and tank 2:

$$A_1\dot{H}_1(t) + F_2(t) = F_1(t)$$

$$A_2\dot{H}_2(t) + F_0(t) = F_2(t)$$

Symbol definitions are found in Table I. Succeeding variables are functions of time unless otherwise noted. From Bernoulli's equation the inter-tank flow is

$$F_2 = c_1 (\bar{H}_1 - \bar{H}_2)^{1/2} \quad [2]$$

and the discharge flow is

$$F_0 = c_2 (\bar{H}_2 - \bar{H}_3)^{1/2}. \quad [3]$$

Using deviation variables ($\triangle X$ is a relatively small deviation from design point $\bar{X}$) it follows from eqs.[2] and [3] that

$$A_1 \triangle\dot{H}_1 + \triangle F_2 = \triangle F_1 \quad [4]$$

and

$$A_2 \triangle\dot{H}_2 + \triangle F_0 = \triangle F_2. \quad [5]$$

Since F_2 is a function of H_1 and H_2,

$$\triangle F_2 = \frac{\partial F_2}{\partial H_1}\triangle H_1 + \frac{\partial F_2}{\partial H_2}\triangle H_2 .$$

It is necessary to substitute functions of H_1 and H_2 for F_0 and F_2 into eqs. [4] and [5]. Taking the partials of F_2 and evaluating at design conditions yields

$$\frac{\partial F_2}{\partial H_1} = \frac{\bar{F}_2}{2(\bar{H}_1 - \bar{H}_2)} ,$$

a linear approximation which we define as 1/R1 where R_1 is called the fluid resistance. Similarly

$$\frac{\partial F_2}{\partial H2} = \frac{-\bar{F}_2}{2(\bar{H}1 - \bar{H}2)} = \frac{-1}{\bar{R}1} .$$

TABLE I

PROCESS AND SYSTEM SYMBOLS

CONSTANT	DESCRIPTION
A_1	area of tank 1
A_2	area of tank 2
R_1, R_2	linearized fluid resistances
c_1, c_2	valve constants
K_c	controller gain
K_p	process gain
K_t	transmitter gain
T_i	controller integral (reset) time
T_d	controller derivative time
T_A, T_B	process time constants

TIME VARYING	DESCRIPTION
H_1	tank 1 fluid level
H_2	tank 2 fluid level
F_1	input flow to tank 1
F_2	inter-tank flow
F_0	discharge flow from tank 2

Eq. [5] can be rewritten as

$$\Delta F_2 = \frac{\Delta H_1 - \Delta H_2}{R_1} . \qquad [6]$$

By a parallel argument, for F_0 a function of H_2 and H_3 (H_3 constant) it follows that

$$\Delta F_0 = \frac{\partial F_0}{\partial H_2} \Delta H_2$$

$$\frac{\partial F_0}{\partial H_2} = \frac{\bar{F}_0}{2(\bar{H}_2 - \bar{H}_3)} = \frac{1}{R_2}$$

$$\Delta F_0 = \frac{\Delta H_2}{R_2} . \qquad [7]$$

Substituting Eq. [6] into [4]

$$A_1 \Delta \dot{H}_1 + \frac{\Delta H_1 - \Delta H_2}{R_1} = \Delta F_1 . \qquad [8]$$

Substituting eqs.[6] and [7] into [5]

$$A_2 \Delta \dot{H}_2 + \frac{\Delta H_2}{R_2} = \frac{\Delta H_1 - \Delta H_2}{R_1} . \qquad [9]$$

Rearranging eqs.[8] and [9] into standard form where the dependent variables are on the left-hand side

$$A_1 R_1 \Delta \dot{H}_1 + \Delta H_1 - \Delta H_2 = R_1 \Delta F_1 \qquad [10]$$

and

$$A_2 \frac{R_1 R_2}{R_1 + R_2} \Delta \dot{H}_2 + \Delta H_2 - \frac{R_2}{R_1 + R_2} \Delta H_1 = 0 \qquad [11]$$

which are a pair of coupled first order linear differential equations (Eq. [11] also is homogeneous).

If the tanks are initially at design conditions (zero initial conditions), the Laplace transforms of eqs.[10] and [11] are

$$(A_1R_1s + 1)\Delta H_1(s) - \Delta H_2(s) = R_1\Delta F_1(s)$$

and
$$-R_2\Delta H_1(s) + [A_2R_1R_2s + (R_1 + R_2)]\Delta H_2(s) = 0.$$

Note that we have moved from the time domain to the complex s domain.

Solving for $\Delta H_2(s)$, the controlled variable by Cramer's rule is

$$\Delta H_2(s) = \frac{\begin{vmatrix} A_1R_1s + 1 & R_1\Delta F_1 \\ -R_2 & 0 \end{vmatrix}}{\begin{vmatrix} A_1R_1s + 1 & -1 \\ -R_2 & A_2R_1R_2s + R_1 + R_2 \end{vmatrix}}$$

$$= \frac{R_2\Delta F_1(s)}{A_1A_2R_1R_2s_2 + [A_1(R_1 + R_2) + A_2R_2]s + 1} .$$

Dividing both sides by $\Delta F1(s)$ gives the transfer function of the process, $G_p(s)$. The denominator of the above equation is the characteristic polynomial of the process. Since it is second order it can be rewritten for convenience as

$$(T_As + 1)(T_Bs + 1) = A_1A_2R_1R_2s^2 + [A_1(R_1 + R_2) + A_2R_2]s + 1 .$$

If we let $K_p = R_2$ (the process gain), then

$$G_p(s) = \frac{K_p}{(T_As + 1)(T_Bs + 1)} .$$

Looking at the open loop system transfer function G(s), we have

$$G(s) = K_t G_c(s) K_m G_p(s). \quad [12]$$

$G_c(s)$ and $G_p(s)$ are, respectively, the controller and process transfer functions. Other terms are defined in Table I.

The closed loop system transfer function is

$$\frac{\Delta H_2(s)}{\Delta H_{2s}(s)} = \frac{G(s)}{1 + G(s)} \quad [13]$$

assuming no change of load variables from design conditions. The transfer function for a P-I-D controller is

$$G_c(s) = K_c \left[1 + \frac{1}{T_i s} + T_d s \right] . \quad [14]$$

Therefore substituting eqs.[12] and [14] into Eq. [13]

$$\frac{\Delta H_2(s)}{\Delta H_{2s}(s)} = \frac{K_0(T_d s + 1)T_i s + 1}{(T_A s + 1)(T_B s + 1)T_i s + K_0 \left[(T_d s + 1)T_i s + 1 \right]} \quad [15]$$

where $K_0 = K_t K_c K_m K_p$ is the loop gain.

The characteristic equation of the system, which comes from the denominator of Eq. [15] can be rewritten as

$$s^3 + \frac{(T_A + T_B + K_0 T_d)s^2}{T_A T_B} + \frac{(K_0 + 1)s}{T_A T_B} + \frac{K_0}{T_A T_B T_i} = 0.$$

The roots of this equation yield points which may be plotted on the complex s plane. As K_c varies the roots change and a locus of points may be drawn.

CHAPTER 3 THE LIN-BAIRSTOW ROOT SOLVING ALGORITHM

Advantages of the Method

The roots of any order polynomial may be solved for by using this algorithm (McCalla 1967) even though the highest order characteristic equation generated by the model is third order. However, a more complex controller could be incorporated. The model might be expanded. Each of these changes would probably induce a higher order characteristic equation. This root solving method would, then, still be useful. Furthermore, most linear systems could employ the algorithm.

In addition to its general utility, the instructional value of Lin-Bairstow's method contributed to its selection. It is very efficient since it converges to each root quadratically. A root locus usually involves the calculation of complex roots. The Lin-Bairstow method has an advantage of not requiring any complex arithmetic. It requires only real arithmetic to calculate a complex zero and, simultaneously, its conjugate.

The Algorithm

A polynomial of any order greater than two can be factored into products of quadratic factors and perhaps one linear factor, all of which involve only real

coefficients (McCalla 1967). The roots of the quadratic may, of course, be complex. The main idea of the algorithm is to provide an efficient way of extracting the quadratic factors iteratively, and if necessary, the linear factor. Flow chart Figure 4 provides an overview of the method.

Suppose that some polynomial $P_n(x)$ of order n is divided by an arbitrary quadratic factor $x^2 + rx + s$. We obtain a polynomial $P_{n-2}(x)$ two orders lower and a remainder term $Rx + S$. If the remainder term were zero then our trial factor would be an exact factor of $P_n(x)$. Therefore, operative constraints are that

$$R(r, s) = 0 \qquad [16]$$

and

$$S(r, s) = 0. \qquad [17]$$

The remainder coefficients are written as functions of r and s since variations in these produce different remainders. Equations [16] and [17] are two non linear equations in two unknowns. Newton's method may now be applied. For sufficiently close initial estimates the method will converge (Dorn 1972).

Using a first order Taylor series approximation about an initial r and s, and in terms of differentials

$$dR = R(r + dr, s + ds) - R(r, s)$$
$$= R_r dr + R_s ds + \ldots \qquad [18]$$

and

$$dS = S(r + dr, s + ds) - S(r, s)$$
$$= S_r dr + S_s ds + \ldots \qquad [19]$$

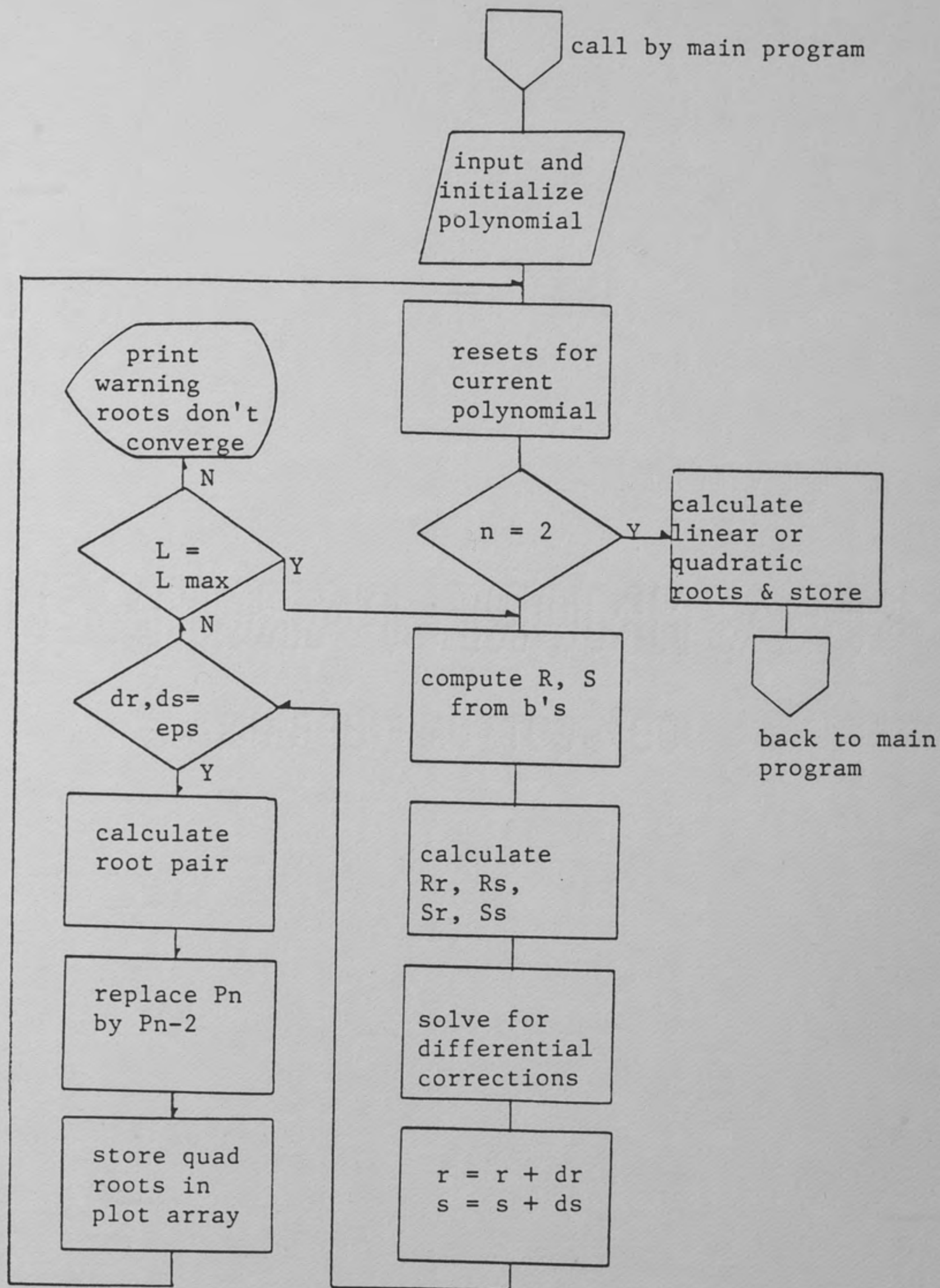


Figure 4. Flow Chart of Root Solving Method

where " . . . " indicates higher order terms which may be dropped. Now if r_0 and s_0 are estimates for the factor $x^2 + r_0x + s_0$ such that

$$R(r_0, s_0) \neq 0$$

and

$$S(r_0, s_0) \neq 0,$$

dr and ds must be found such that these constraints are true:

$$R(r_0, s_0) + dR = 0$$

and

$$S(r_0, s_0) + dS = 0,$$

or

$$dR = - R(r_0, s_0) \quad [20]$$

and

$$dS = - S(r_0, s_0). \quad [21]$$

Therefore from Eqs. [18], [19], [20], and [21]

$$R_r dr + R_s ds = - R(r_0, s_0) \quad [22]$$

and

$$S_r dr + S_s ds = - S(r_0, s_0). \quad [23]$$

The last two equations are called "differential-correction" equations.

By solving for dr and ds we can satisfy our original constraints

$$R(r_0 + dr, s_0 + ds) = 0 \quad [24]$$

and

$$S(r_0 + dr, s_0 + ds) = 0. \quad [25]$$

Since $r_0 + dr = r_1$ is a first order approximation to the zero of R, we can refine our solution using the above technique iteratively, starting from r_1. A similar procedure is needed for finding the zero of S. If the initial guess r_0 and s_0 is sufficiently close we can

converge towards the roots of eqs. [22] and [23] to within some arbitrarily small number epsilon.

Calculations from Polynomial Coefficients

In order to solve for dr and ds in eqs. [24] and [25] it is necessary to obtain six numbers from the original polynomial. Suppose it is given that

$$P_n(x) = x^n + a_1x_{n-1} + a_2x_{n-2} + \dots + a_{n-1}x + a_n$$
$$= (x^2 + rx + s)(x_{n-2} + b_1x_{n-3} + b_2x_{n-4} + \dots$$
$$+ b_{n-3}x + b_{n-2}) + Rx + S.$$

The a's are coefficients of the original polynomial, and b's are coefficients of the reduced polynomial. It is shown that (McCalla 1967) after quadratic factor division:

$$R = b_{n-1} = a_{n-1} - rb_{n-2} - sb_{n-3} \qquad [26]$$
$$b_n = a_n - rb_{n-1} - sb_{n-2}$$
$$S = b_n + rb_{n-1}. \qquad [27]$$

Furthermore, using the notation

$$p_k = \frac{\partial b_k}{\partial r}$$

and

$$q_k = \frac{\partial b_k}{\partial s}$$

it is also shown that (McCalla 1967):

$$p_k = -b_{k-1} - rp_{k-1} -sp_{k-1} -sp_{k-2}$$
$$q_k = -b_{k-2} - rq_{k-1} -sq_{k-2}$$
$$R_r = \frac{\partial b_{n-1}}{\partial r} = p_{n-1} \qquad [28]$$

$$R_s = \frac{\partial b_{n-1}}{\partial s} = q_{n-1} \quad [29]$$

$$S_r = p_n + rp_{n-1} + b_{n-1} \quad [30]$$

$$S_s = q_n + rq_{n-1}. \quad [31]$$

Eqs. [26], [27], and [28] through [31] are the six numbers obtained from recursion formulas for solving the differential correction equations.

CHAPTER 4 PROGRAM DESCRIPTION

The root locus plotter can be divided into two major sections. The first is concerned with obtaining variable controller parameters from the editor, calculating a system characteristic polynomial, and iteratively finding the roots of the characteristic as the controller gain is varied. The roots are then stored in real number arrays for plotting.

The second section makes use of the Turbo Graphix Toolbox for IBM monochrome high resolution graphing of the root locus. Procedures needed from the utility package are accessed via "include files," which are compiled integrally with the main program. The procedures used feature windowing; axis drawing; drawing of points, lines, and numbers; an automatic world coordinate system; and a virtual memory screen. A few of the Turbo Graphix routines were modified slightly for improved results.

Figures 5 and 6 are macroscopic flow charts which provide an overview of program control flow. For a closer look at how the program is structured, and for details regarding subroutines the reader may refer to the commented program source code in Appendix II.

Certain problems unique to this application were encountered. When a fixed step size is specified for the

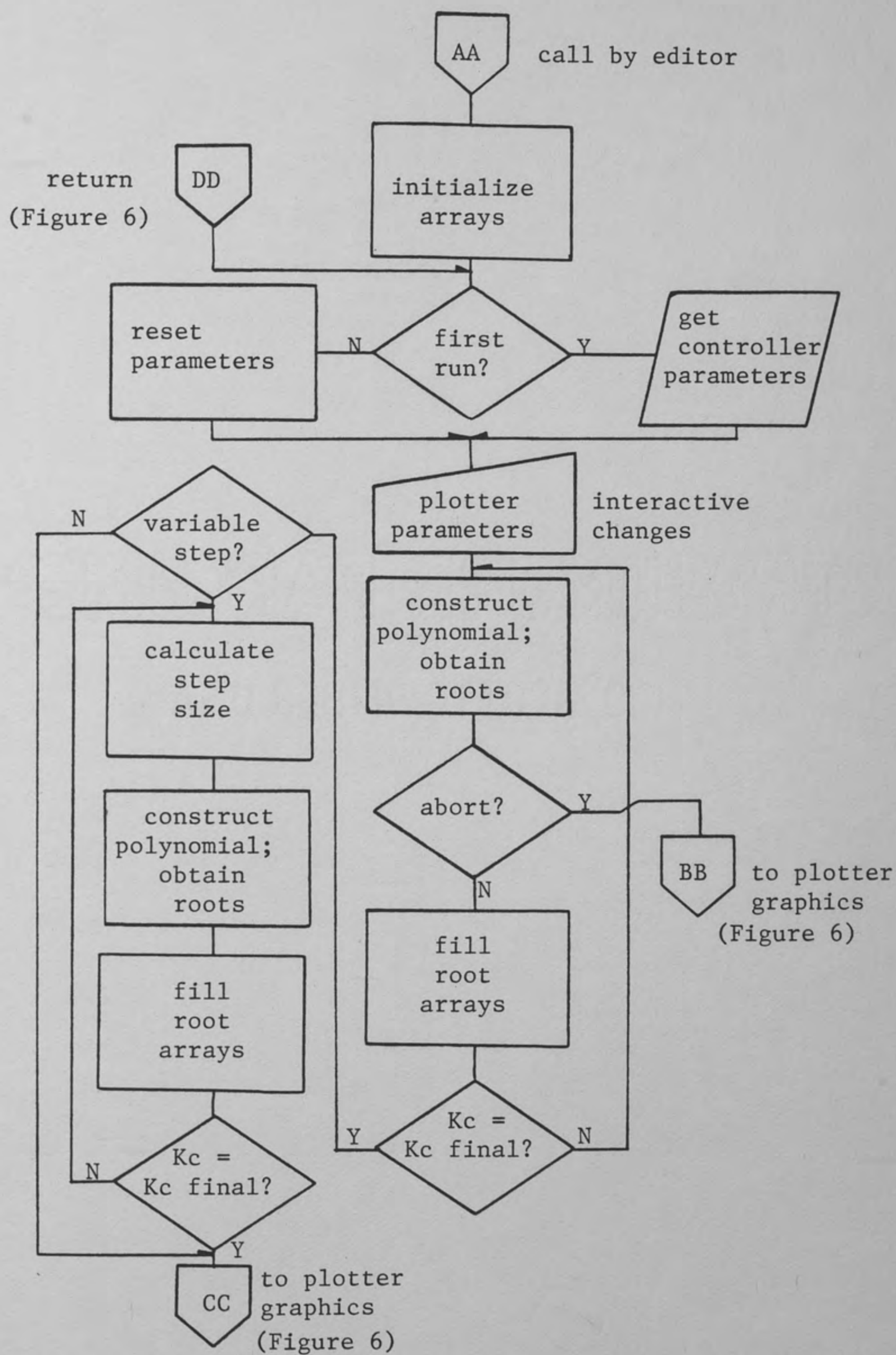


Figure 5. Flow Chart for Root Locus Program Numerical Routines

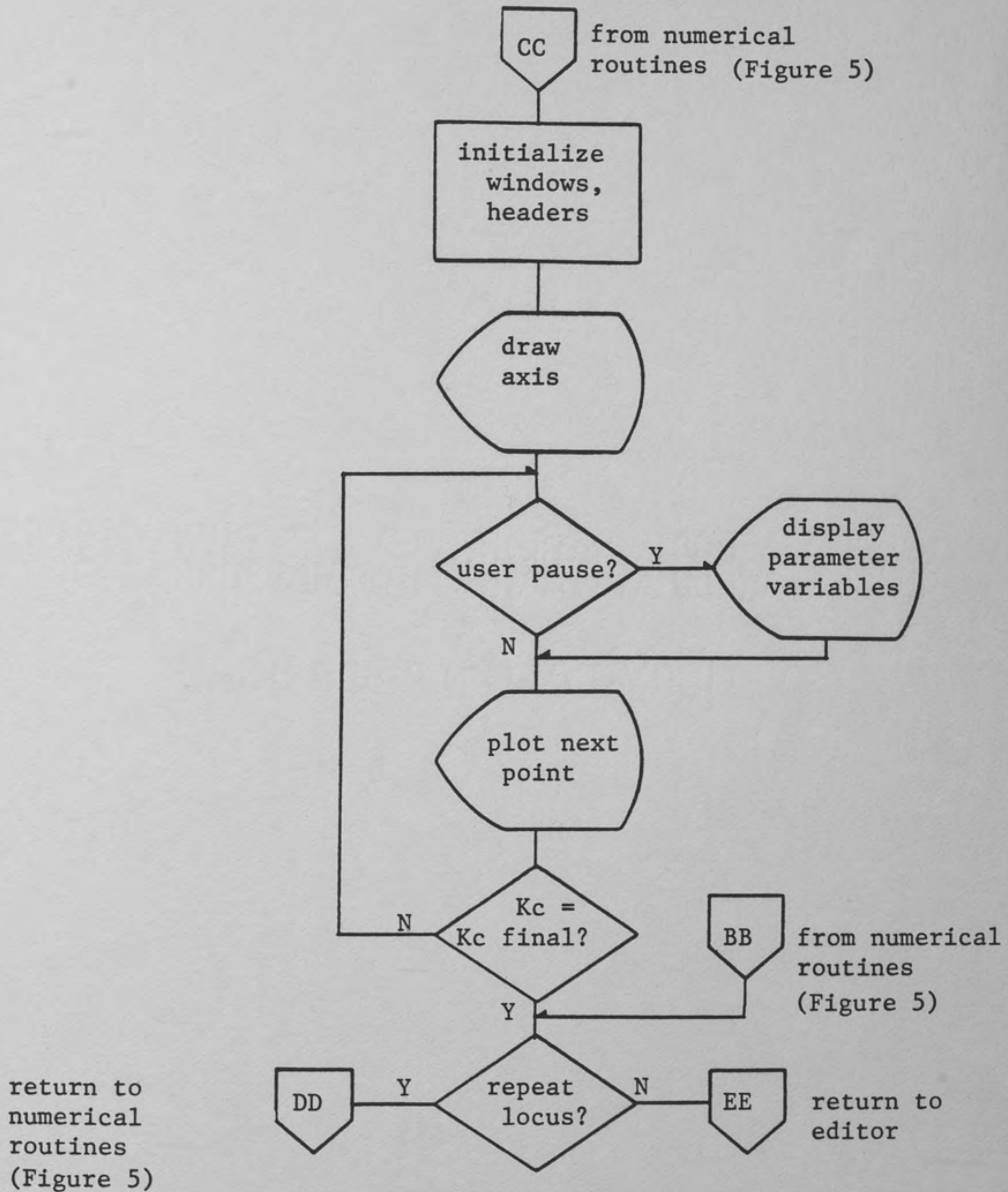


Figure 6 . Flow Chart for Root Locus Program Graphics Routines

controller gain K_c, the spacing of points on the root locus varies. A moderate step size should be selected to produce a quick but smooth curve. However when the roots break away from or reenter the real axis, point spacing suddenly widens. Therefore it was necessary to incorporate a variable step size option.

This requires two passes to find the roots. The first pass uses a fixed step size and stores the calculated points (roots) in a two dimensional plot array. (The imaginary part of the root corresponds to the root locus' vertical axis.) A second pass examines the spacing between any two points found in the first pass. A linear interpolation is made to find the desired step size:

$$\frac{\text{fixed step size}}{\text{desired step size}} = \frac{\text{first pass point spacing}}{\text{desired point spacing}}$$

The procedure which handles this calculation is "getstepsize." The effect of this linear interpolation is to moderate the step size as needed for most points, but near the break away, spacing change is too large for this method alone. Therefore the step size is additionally decreased by a factor of one-fifth. A counter ensures that this reduced step size is used five times. A similar procedure is used for re-entry. Since either fixed or variable step size options may be selected by the user, they may be compared for best results.

Another problem encountered was that the Graphix Toolbox was not able to produce round, even numbers for axi numbering in spite of a variable axis density scheme provided. It was necessary to incorporate a "world" finding routine into the plotter program. This routine, "findXlimYlim," determines the order of magnitude and size limits of the roots to be plotted for a simulation. The values obtained are rounded off. Thus the Toolbox routine "Findworld" was bypassed. Furthermore, the axis drawing routine was modified in order to yield nice, even numbers on the axes for most cases.

The user may elect to temporarily pause plotting. A memory based virtual screen is used to save the display, and auxiliary windows pop up which contain certain variables and parameters.

Occasionally due to certain odd controller parameters entered by the user, the root solving algorithm does not converge to within user specifications. A diagnostic warning is displayed, and the user may elect to abort further calculations, or continue with unpredictable plotter results. Error checking is incorporated for all user input to ensure that entries are reasonable and within range. Figure 7 shows two typical input screens.

The plotter program, including Turbo Graphix "include" files, yields an instruction code segment of more than forty

CONTROLLER

Change a parameter. Select one.

1. Kc
2. Td
3. Ti

_/ no change

?

OPTIONS

Vary step size?

Yes No

Figure 7. Typical Input Screen - Controller Parameters Select

kilobytes. When coupled with the calling editor and other subprograms the code is larger than the Turbo Pascal compiler can handle easily. A Turbo Extender shell program (TurboPower 1986) was used to assist in compilation of the entire program. Note again that the plotter subprogram can run alone with minor modifications, and will compile using only Turbo Pascal.

Heap and stack memory management was necessary, but this was easily implemented with standard Turbo Pascal functions. Since the compiler window procedure does not work with the Toolbox window routines the Toolbox was, again, modified.

CHAPTER 5 DISCUSSION OF THE RESULTS

The following illustrates how the root locus plotter may be used to analyze and design for the dual tank system. Sample graphs of the plotter are provided. The TUTSIM simulation program is also used to validate or extend plotter results.

In order to find some unique combination of controller parameters it is necessary to specify some goal to be achieved with respect to the system. One such goal is to tune the controller optimally by the quarter decay ratio method, as first published by Ziegler and Nichols (Weber 1973).

By quarter decay it is meant that the step response should exhibit damped oscillations (underdamped response) such that the second peak is one-quarter of the height of the first peak (overshoot). While there is no unique combination of P-I-D controller parameters which yield such behavior, a logical set of values may be found by first examining pure proportional control.

Suppose the system under feedback control is brought to design conditions. The controller is switched to manual mode, which breaks open the feedback loop. A unit step voltage directly to the pump motor will produce a typical

second order response at the transmitter output since the process under consideration is second order. An optional controller chart recorder can graph the output (Figure 8). Extend a tangent line of maximum slope down to the time axis. The time intercepted is, in effect, a delay. The open loop system can be approximated by a first order transfer function in series with a pure delay element. The slope line and delay time found as detailed above characterize the approximate model. Such a simplified model can mathematically be shown to have the quarter decay response characteristic desired (Smith 1985) when the controller is tuned as follows (thus closing the loop):

$$Kc = \frac{1}{SL}$$

where S is the slope of the tangent line and L is the delay time described above. For our system, values of S = 0.013/min, L = 3.57 min, and Kc = 21.55 were found.

Having found some particular value to set the proportional control, the root locus plotter may be employed for further analysis. Figure 9 shows the root locus stopped at this value. The locus appears to be heading straight up as the gain increases. This observation is confirmed by Figure 17. Using a simulation with Kc = 21.55, a closed loop step response was graphed (Figure 10). A decay ratio smaller than 0.25 is evident, therefore the recommended setting is too conservative. It turns out that

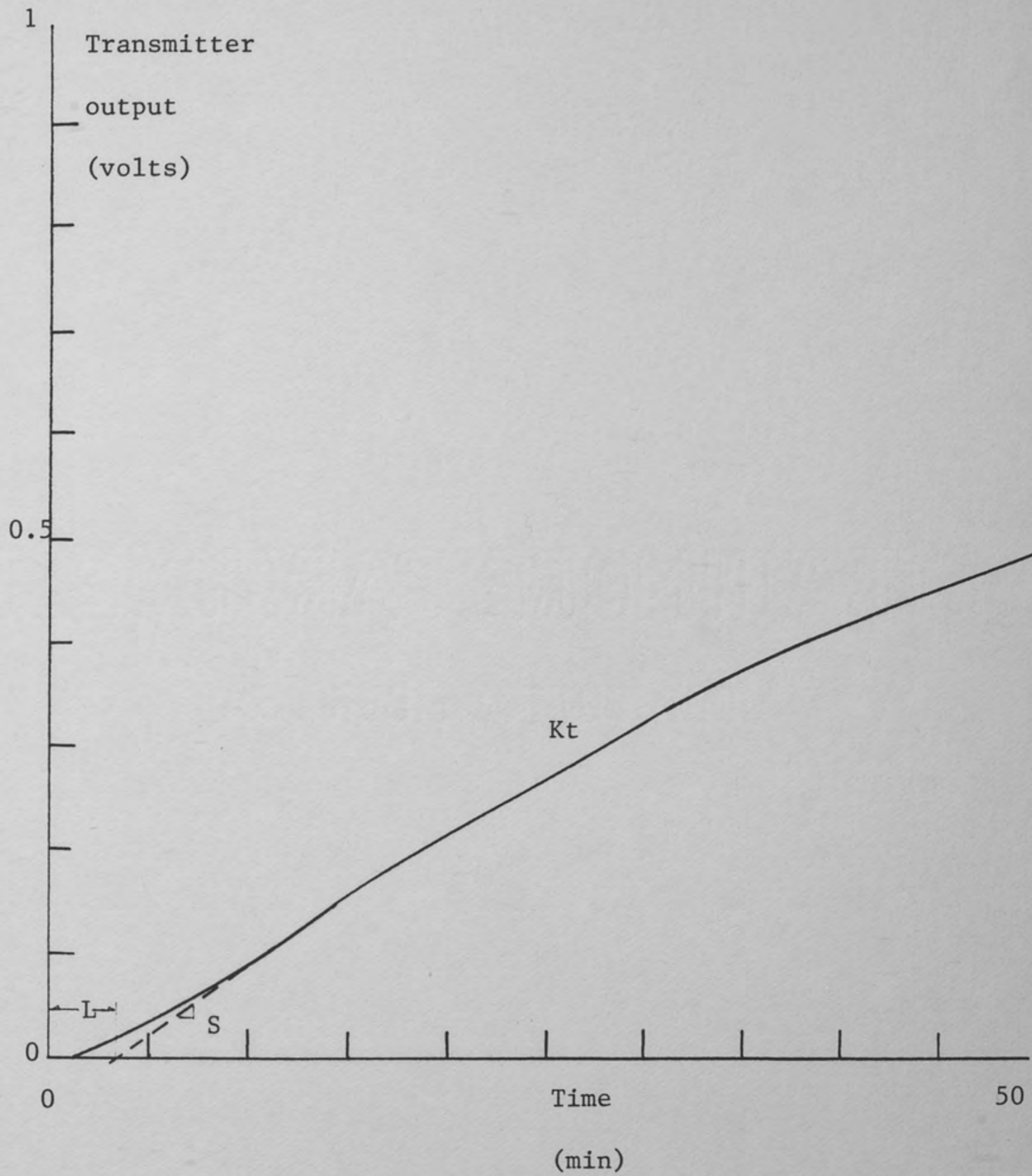


Figure 8. Reaction Curve Method for Tuning the Controller

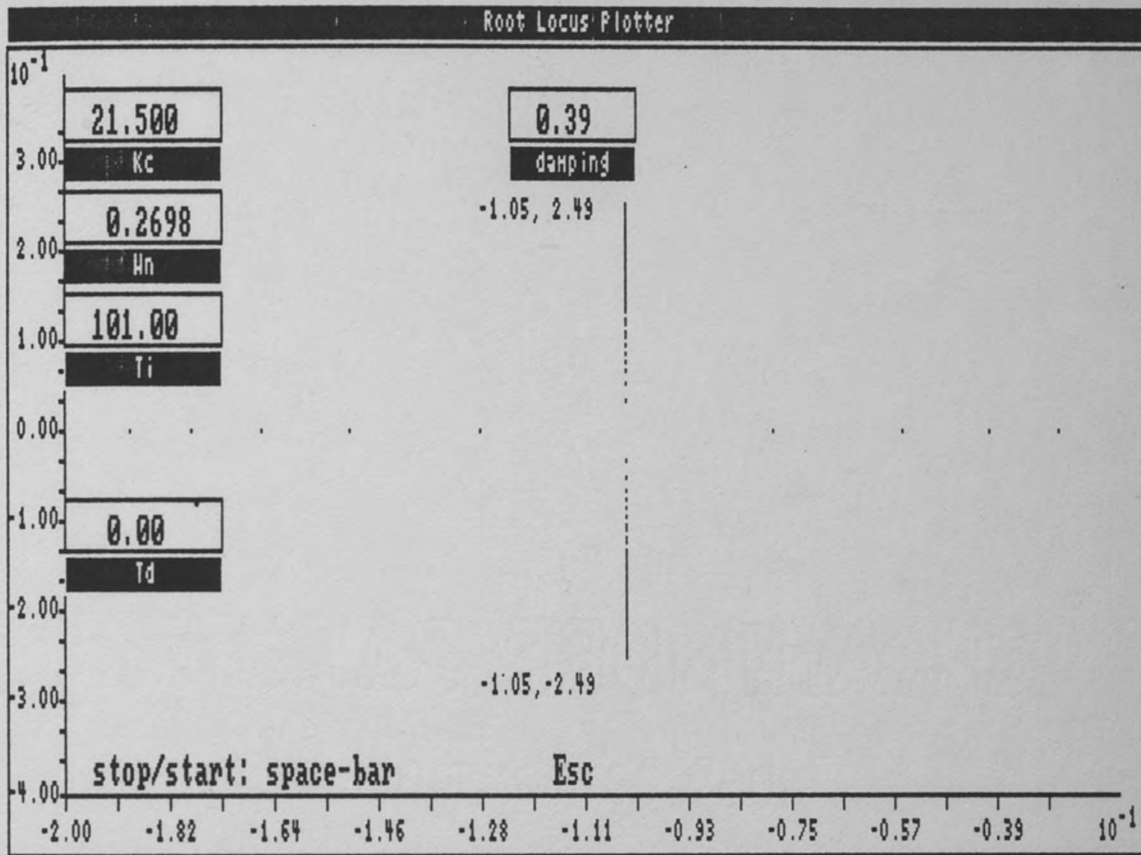


Figure 9 . Optimally Tuned Controller Using Proportional Control Only

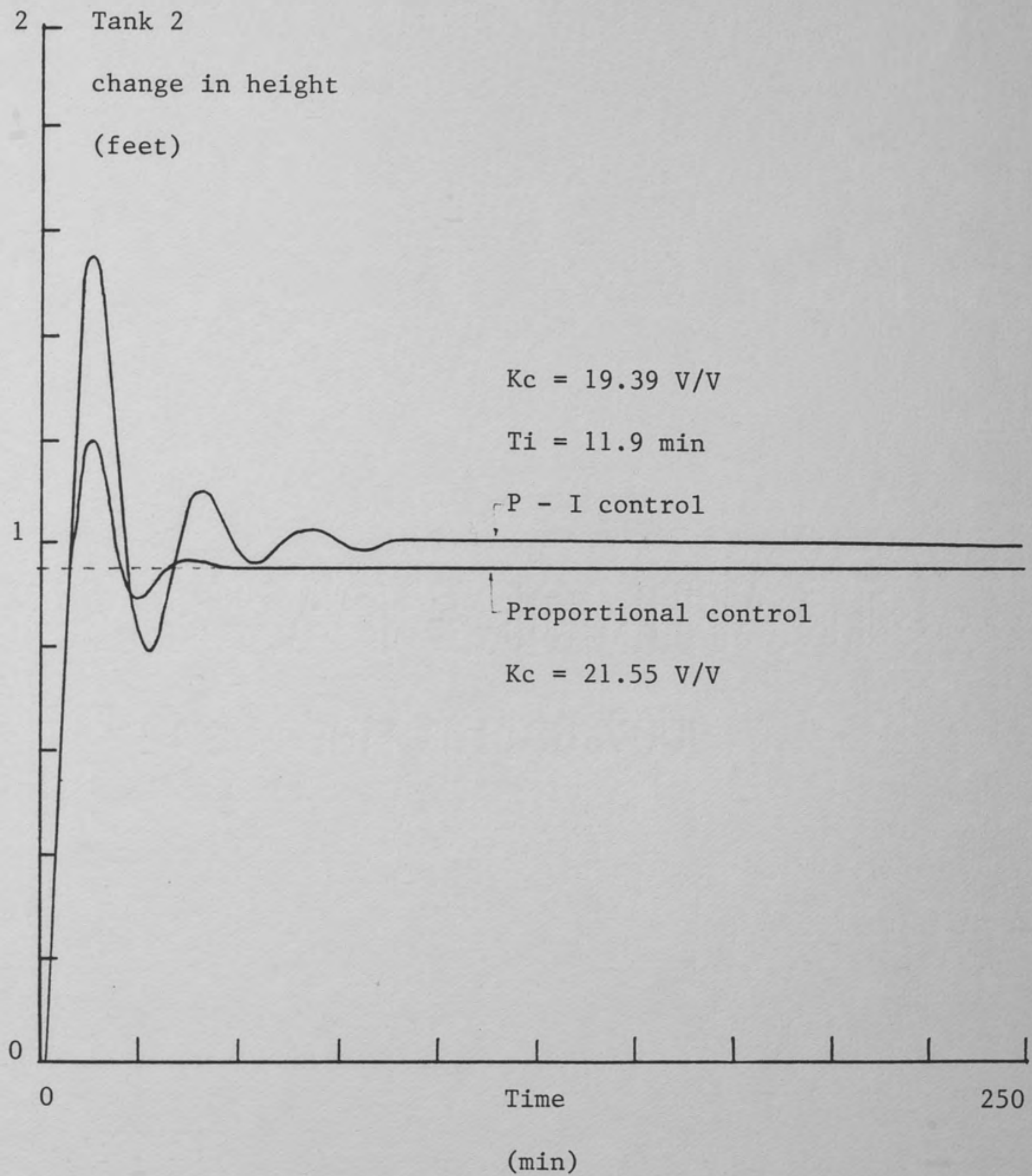


Figure 10. Step Response for P - I Controller and Proportional Controller Tuned Optimally by Reaction Curve Method

a small increase in gain yields the desired response. It should be noted that the goal of quarter decay ratio is more useful in controlling load changes than set point changes. For set point changes this method produces too much overshoot, but it prevents load changes from deviating too far from design without being too oscillatory (Smith 1985).

Another method which has the same quarter decay goal and may employ the root locus principally is the continuous cycling or ultimate gain method. The method dictates that the closed loop system be placed under proportional control only. Increase the controller gain until the system step response oscillates continuously. This point corresponds to the vertical axis crossing of the root locus. At that point read the ultimate gain K_{cu} and the natural frequency w_{nu}. Controller parameters for optimum control may be calculated as follows (Weber 1971).

P $$K_c = 0.5\ K_{cu}$$

P-I $$K_c = 0.45\ K_{cu}$$

$$T_i = \frac{2\ \Pi}{1.2\ w_{nu}}$$

P-I-D $$K_c = 0.6\ K_{cu}$$

$$T_i = \frac{2\ \Pi}{2.0\ w_{nu}}$$

$$T_d = \frac{2\ \Pi}{8.0\ w_{nu}}\ .$$

Unfortunately, for the dual tank system this method does not work. As already mentioned, increasing the gain for the

system under proportional control causes a vertical locus. The system never becomes unstable, which is characteristic of a second order process under proportional control. Most real world processes are of higher order, and some value of gain will cause instability.

One inportant item yielded by the root locus plotter is the damping ratio. Standard second order response curves are available for specified damping factors, which allow the designer to predict what the response will be, i.e., how fast the oscillations will die out. Many higher order systems are characterized by two poles which dominate the response, and can be approximated by a second order system.

Another item is the natural frequency, w_n, which indirectly gives the period of oscillation. Figure 10 also indicates the step response for a P-I controller tuned at the Ziegler-Nichols optimum. It is evident from the corresponding root locus (Figure 11) that the system is far from unstable, but that increasing the gain will eventually cause more oscillations since the path is turning back towards the Y axis. A natural frequency of $w_n = 0.2343$ predicts that the period of oscillation is

$$T = \frac{2\,\Pi}{(1- z^2)\, w_n}$$

$$= \frac{6.28}{0.968\ (0.234)} = 27.7 \text{ min}$$

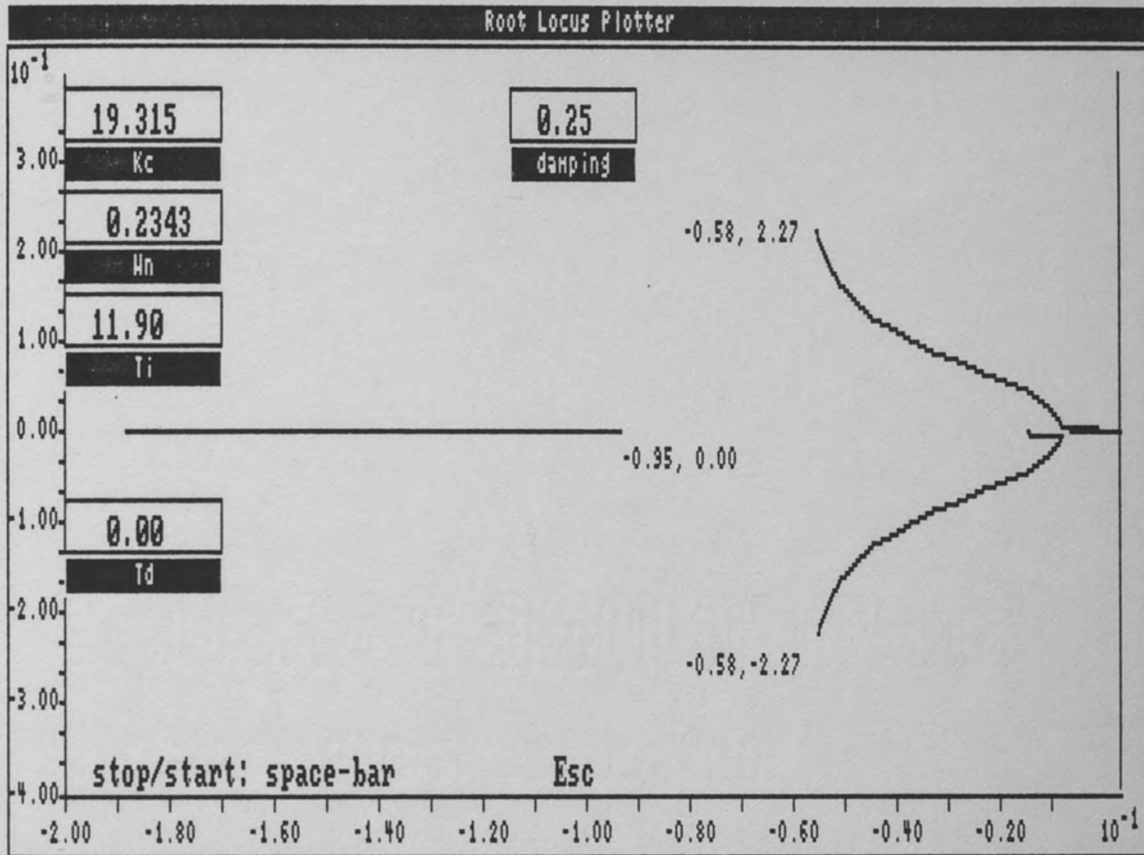


Figure 11. Optimally Tuned Controller Using Proportional - Integral Control

which agrees well with the time response (Figure 10). (z is the damping ratio.) Three peaks are found over a 75-minute interval. Notice that the addition of integral control adds a third pole which moves horizontally along the horizontal axis. The contribution to the response is a negative exponential which dies out more quickly as the gain increases.

The addition of derivative action to P-I control reveals that the system has 3 poles and remains third order (Figure 12). The locus reveals a greater degree of stability as the path continues to move away from the jw axis. Compared to P-I control an increased value of gain is permitted for optimum tuning (Kc = 25.85). Figure 13 shows the corresponding step response which has less initial overshoot, and a faster settling out to steady state. Note that the step input has been arbitrarily delayed for 10 minutes for better graphics. The faster settling time could be predicted from the root locus plot, by noting a higher damping factor (0.32). If we increase the gain of the optimally tuned P-I controller to that of the P-I-D controller, i.e., from Kc = 19.3 to Kc = 25.8, the response becomes too oscillatory, as shown by Figure 14.

The system model was further exercised and certain unique conditions were observed. A controller setting was quickly determined by using the root locus to find a point of marginal stability (Figure 15). The corresponding step

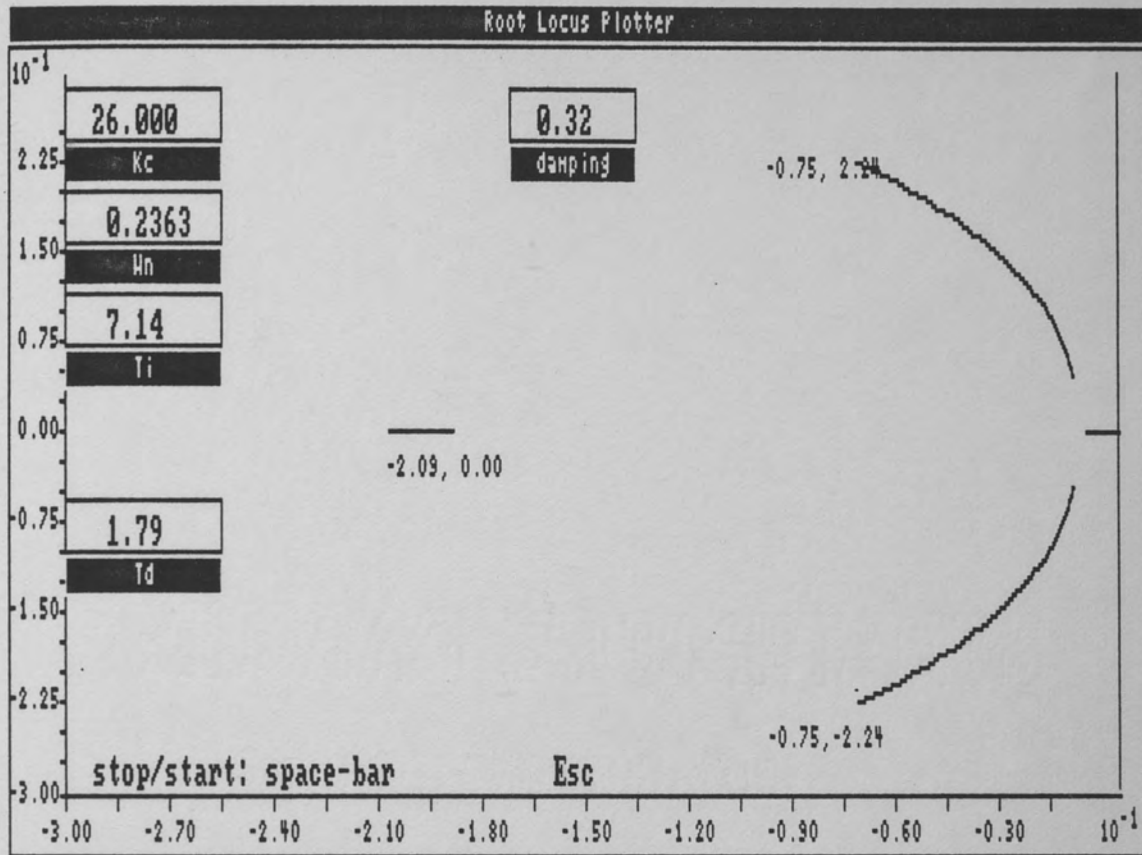


Figure 12. Optimally Tuned Controller Using Proportional - Integral - Derivative Control

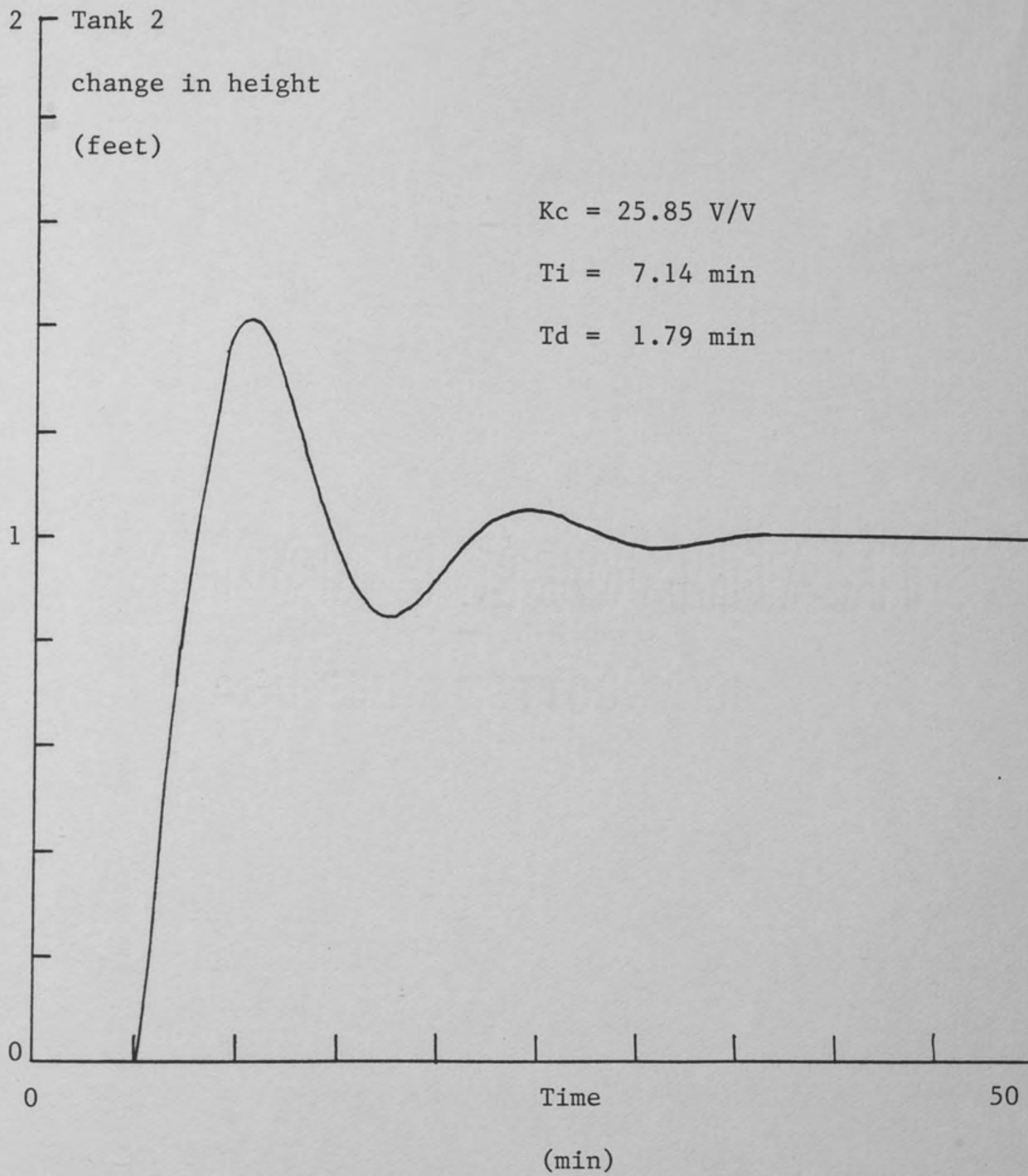


Figure 13. Step Response for P - I - D Controller Tuned Optimally by Reaction Curve Method

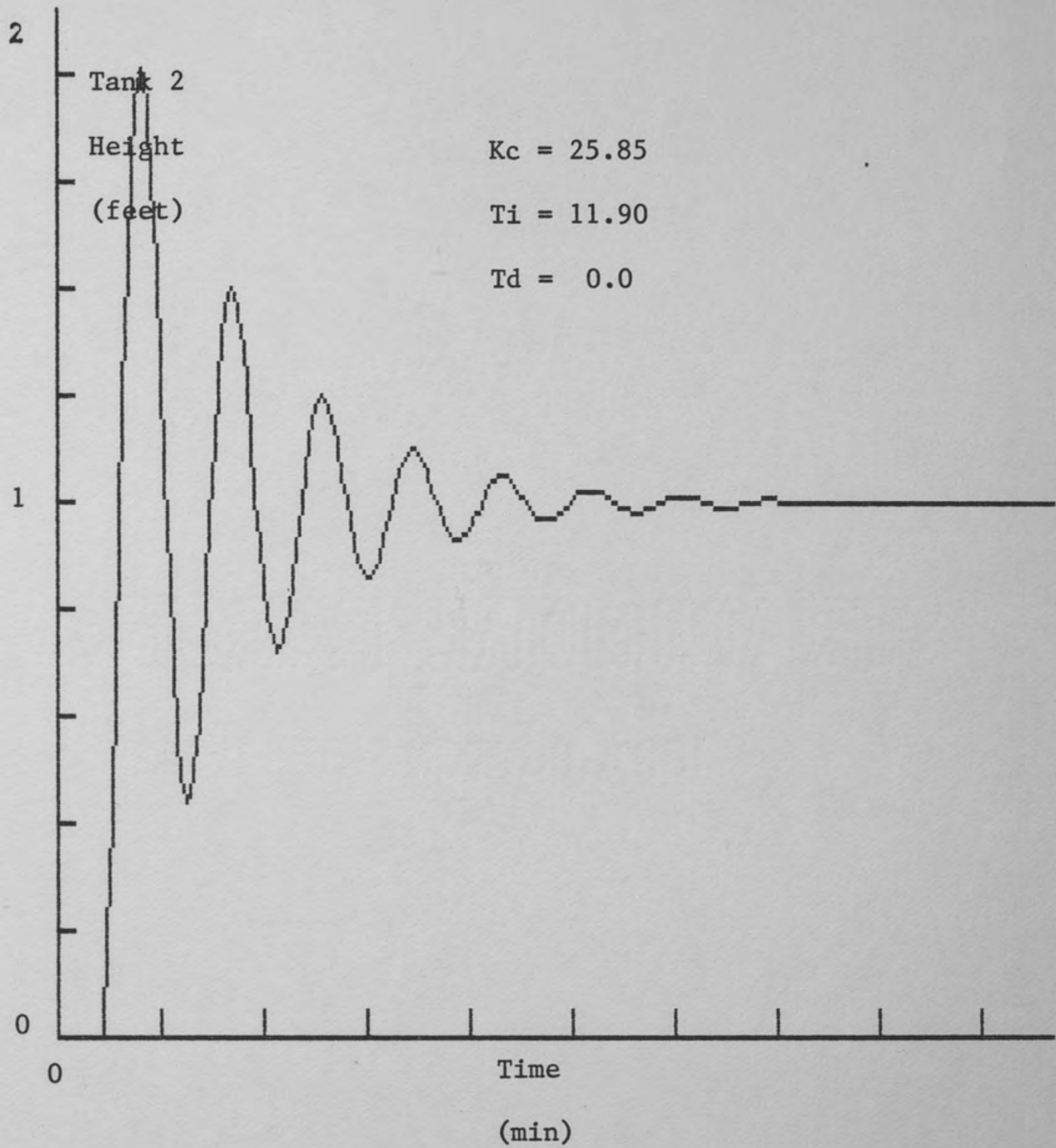


Figure 14. Step Response for P - I Controller Gain Set Higher Than Recommended

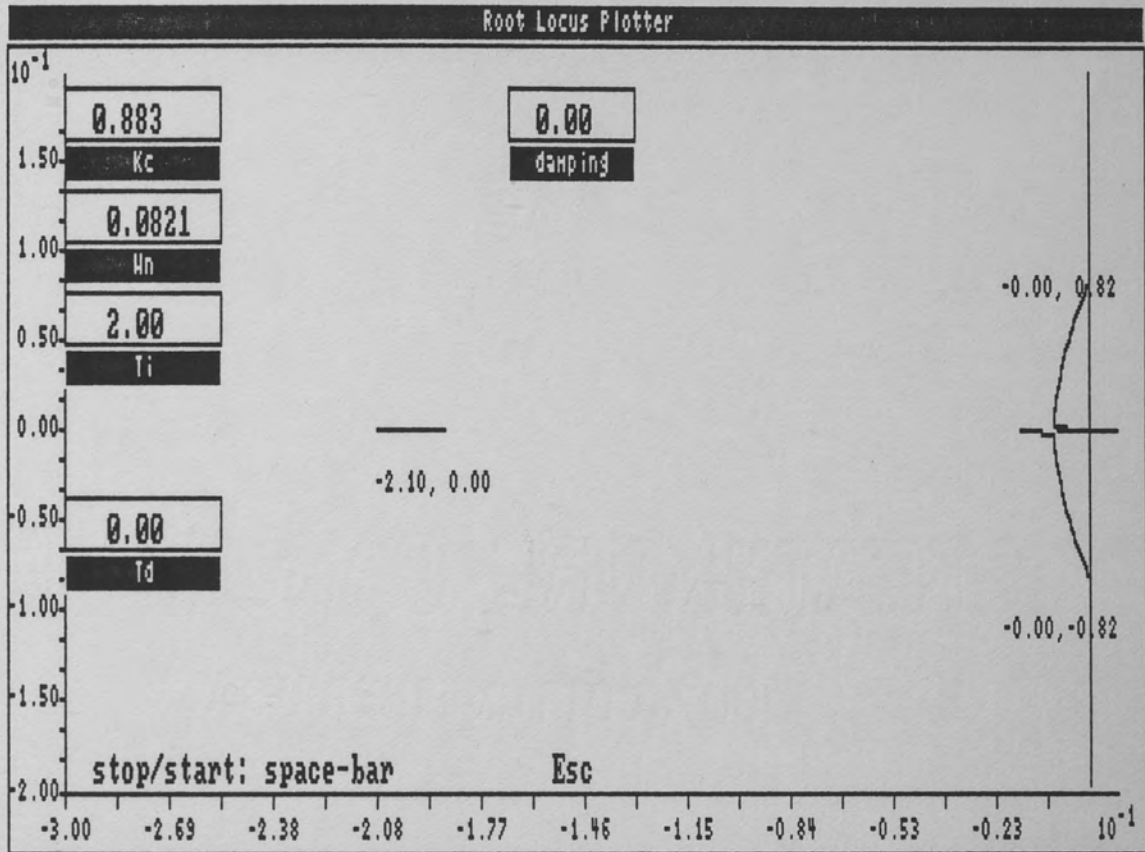


Figure 15. Controller Set for System at Marginal Stability

response agreed nicely, producing relatively constant oscillations (Figure 16). At the point of breakaway the response is critically damped. Using the root locus, this condition was easily found (Figures 18 and 19). For certain systems any overshoot might be unacceptable. The root locus could then be used to find controller settings for the fastest response with no overshoot, as done above.

Another item of interest concerns the plotter's variable step size feature. Although proportional-derivative control is seldom used in practice because of the resulting offset, the root locus plot is interesting as it forms a complete loop (Figure 20). Note how much improved the plotter draws Figure 21 which incorporates a variable step size. At breakaway and reentry, point spacing suddenly increases. A fixed K_c is especially undesirable for this plot. Standard root locus plotters, such as found in the Matlab Control System Toolbox (Figure 22), do not incorporate automatic variable spacing (Moler 1985). Several of the plots made with the root locus were run on Matlab for validation, and there were no discrepancies found between the two plotter routines.

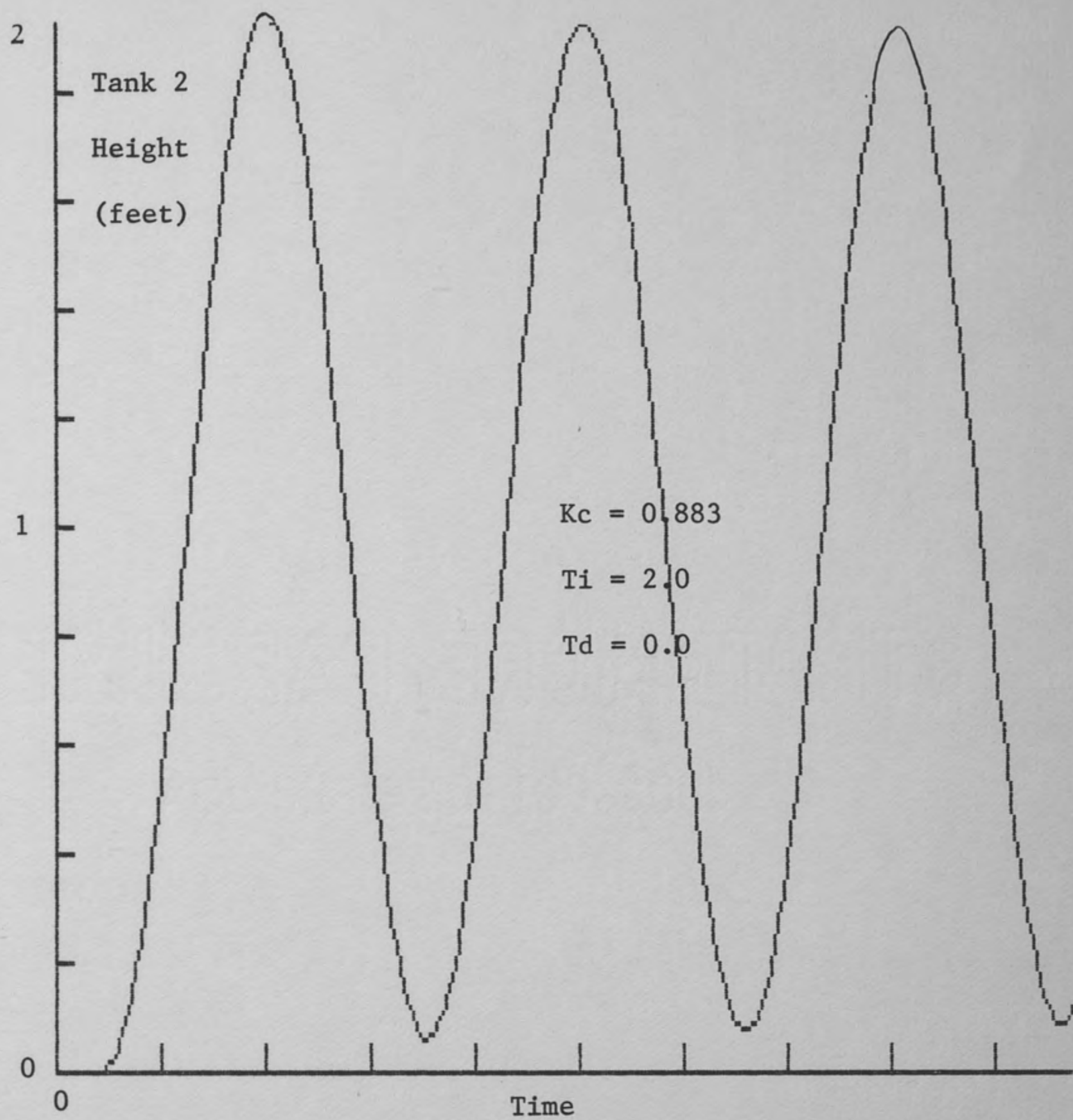


Figure 16. Step Response for Marginally Stable System

CHAPTER 6 SUMMARY AND CONCLUSION

A subprogram was written for the IBM PC/AT to be used as one design tool which is called by an editor program. The editor allows for making, changing, and saving various configurations of the dual tank system. This design tool is the first in a series of several subprograms to be developed by other students. The entire software is written in support of a manuscript on system analysis to be later published as a textbook.

In writing the subprogram my intent was to make the numerical routine modular in the event that the reader wishes to use a Turbo Pascal root finder for some other application. The graphics routines which use the Turbo Graphix Toolbox are set apart, and not essential to the first part of the subprogram. The reader may wish to incorporate his own graphics routines to plot points residing in arrays, since the Toolbox carries a large overhead in subroutines (about 2500 total source lines of code) not used for this application. With minor changes the subprogram can run alone without a calling editor.

Although the subprogram source code (excluding Graphix Toolbox subroutines) is about 1200 lines, its real-time execution is very fast. It is even necessary to incorporate delays in order to allow the user to pause the graphing. The goal in writing the program was to allow for user ease, error checking, and informative graphics. Code minimization was of minor concern.

Rudimentary differential equations for the system led to a derivation of the characteristic equation. Although the system under consideration gave rise to a maximum of third degree polynomials, it was instructive to obtain a higher capability root solving method and make it available in Turbo Pascal. Since the roots of the characteristic equation were almost always small numbers the initial guess of $(r_0, s_0 = 0, 0)$ guaranteed convergence.

The root locus plotter was exercised by attempting to optimally tune the system's controller according to the quarter decay ratio method. It was demonstrated how the root locus could be used to obtain quick information about the time response of a system. Principal items obtained included the relative stability, damping ratio, and the frequency of oscillations of the system.

APPENDIX I ADDITIONAL FIGURES

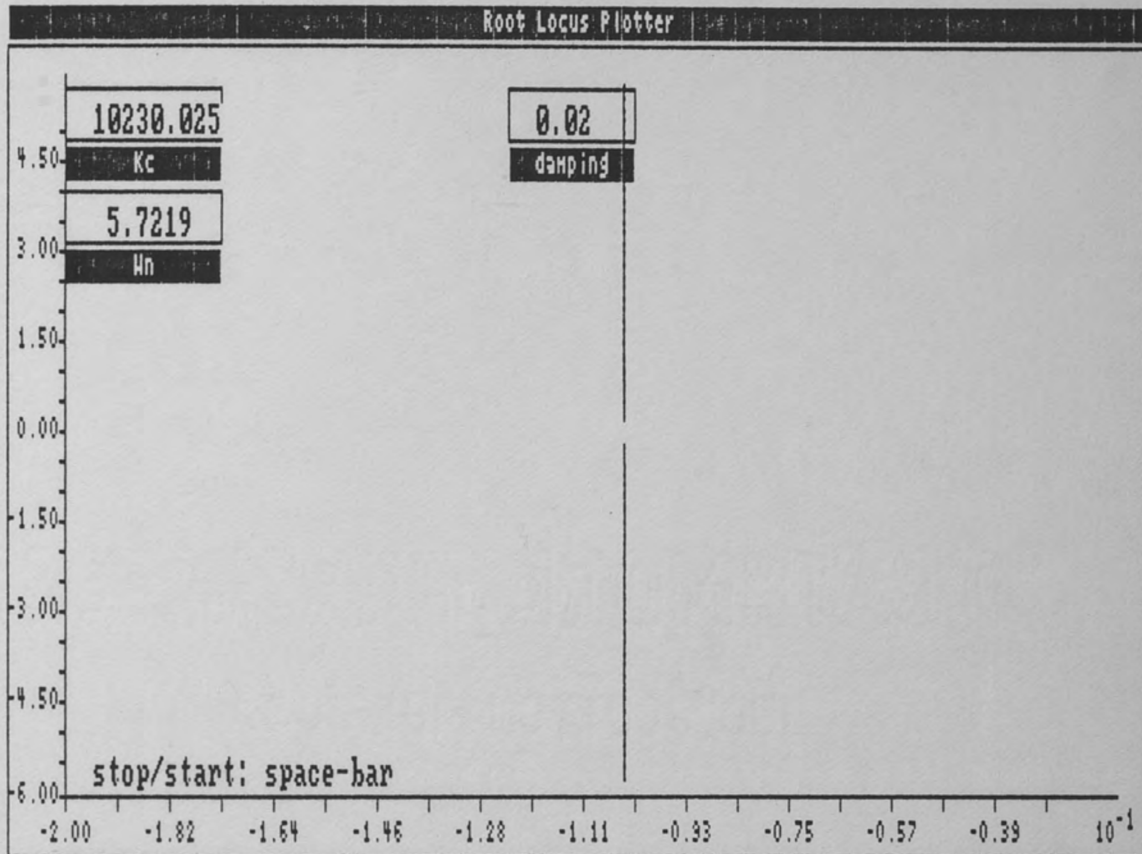


Figure 17. System Under Proportional Control for Very High Gain

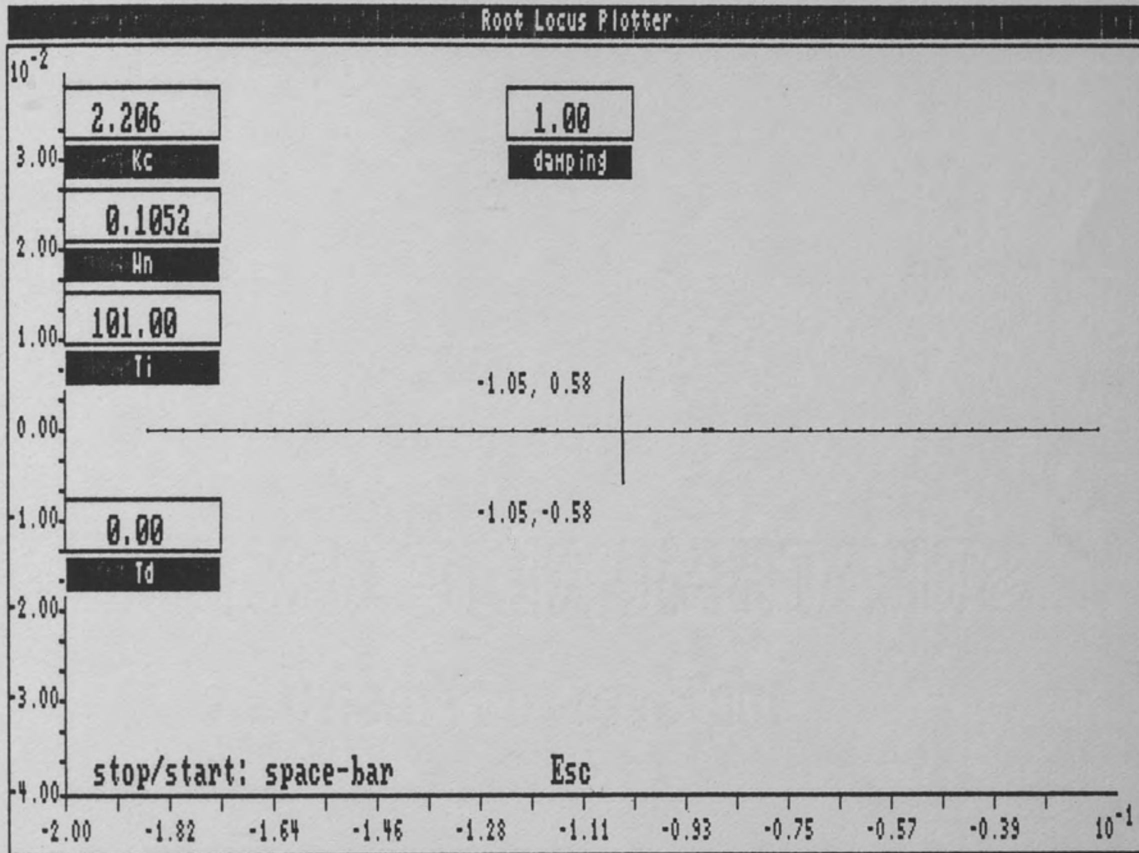


Figure 18. System with Gain Slightly Above Breakaway

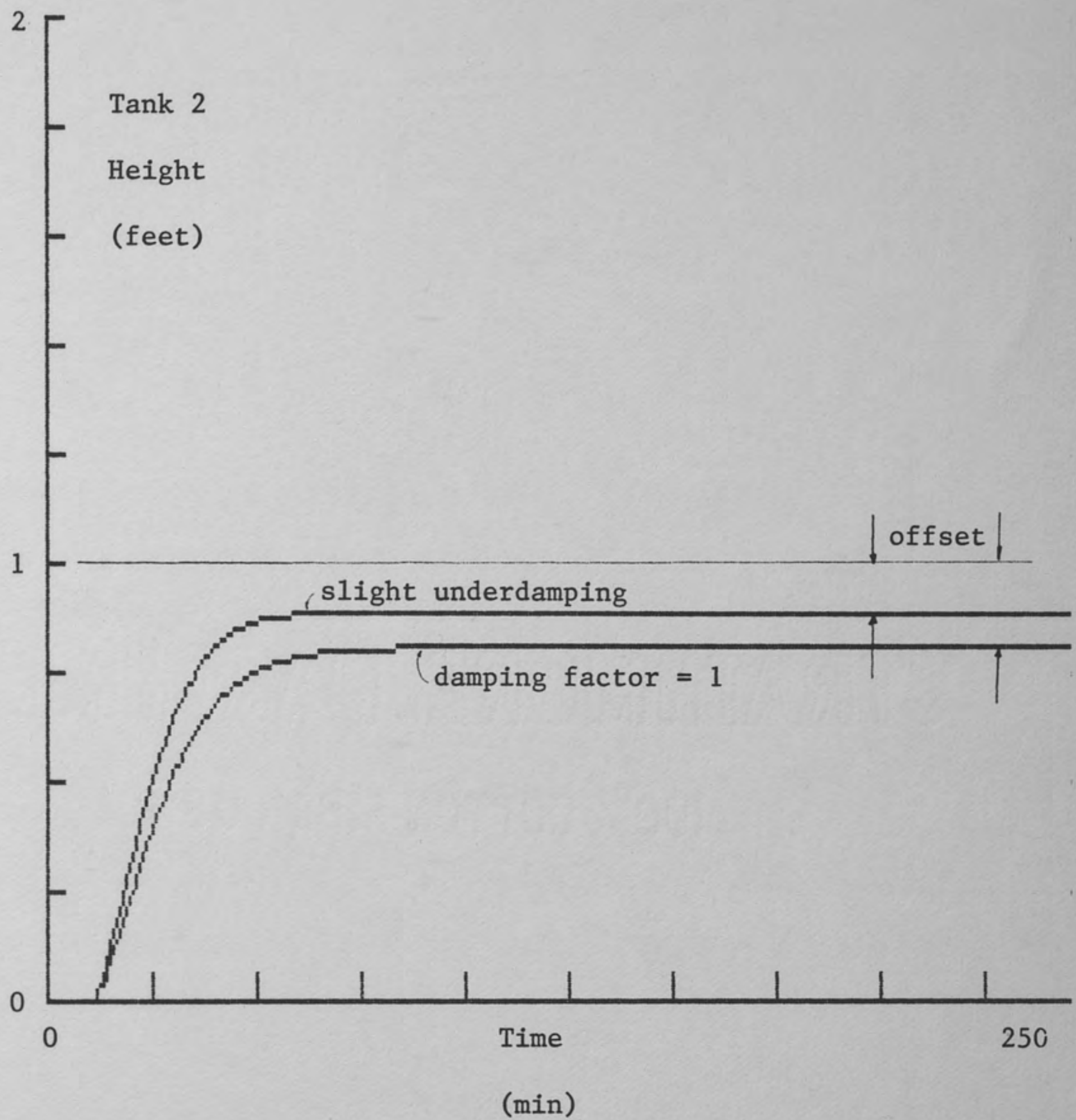


Figure 19. Step Response for System at Breakaway. Controller Using Proportional Gain Only

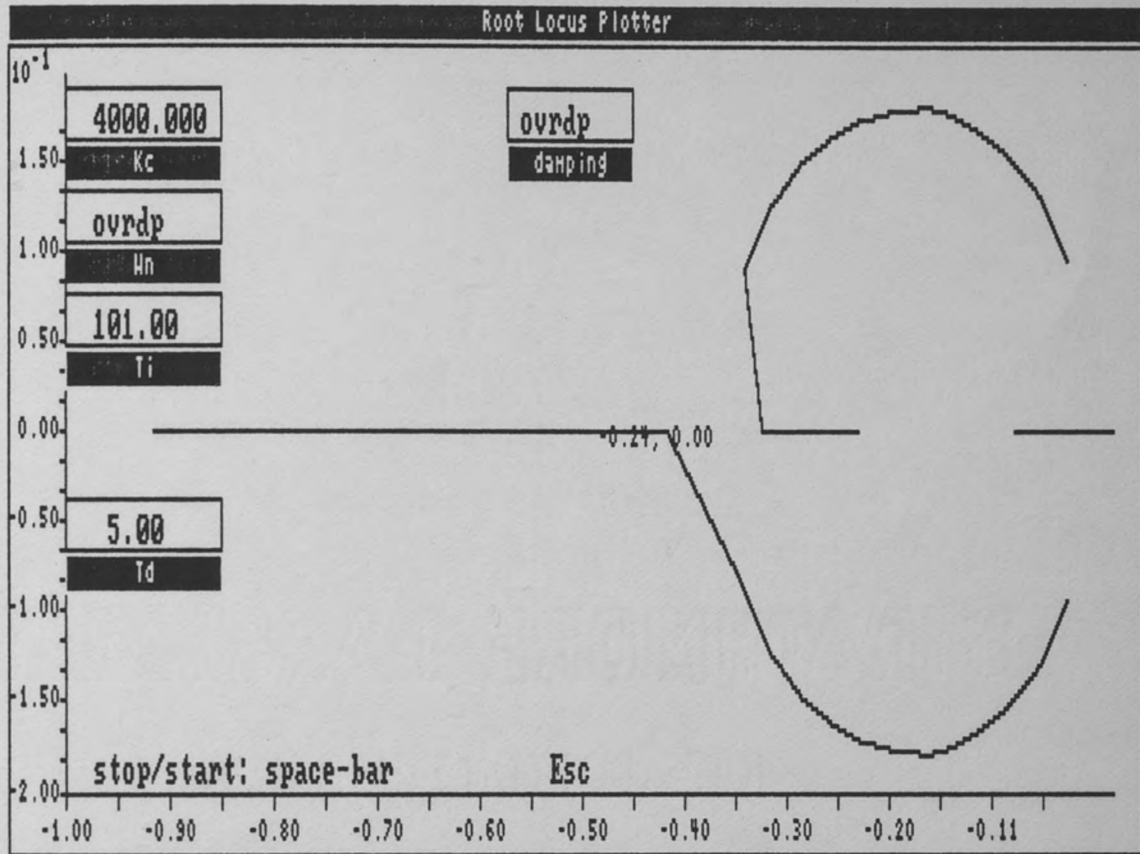


Figure 20. System Under P - D Control. Plot Using Fixed Step Size

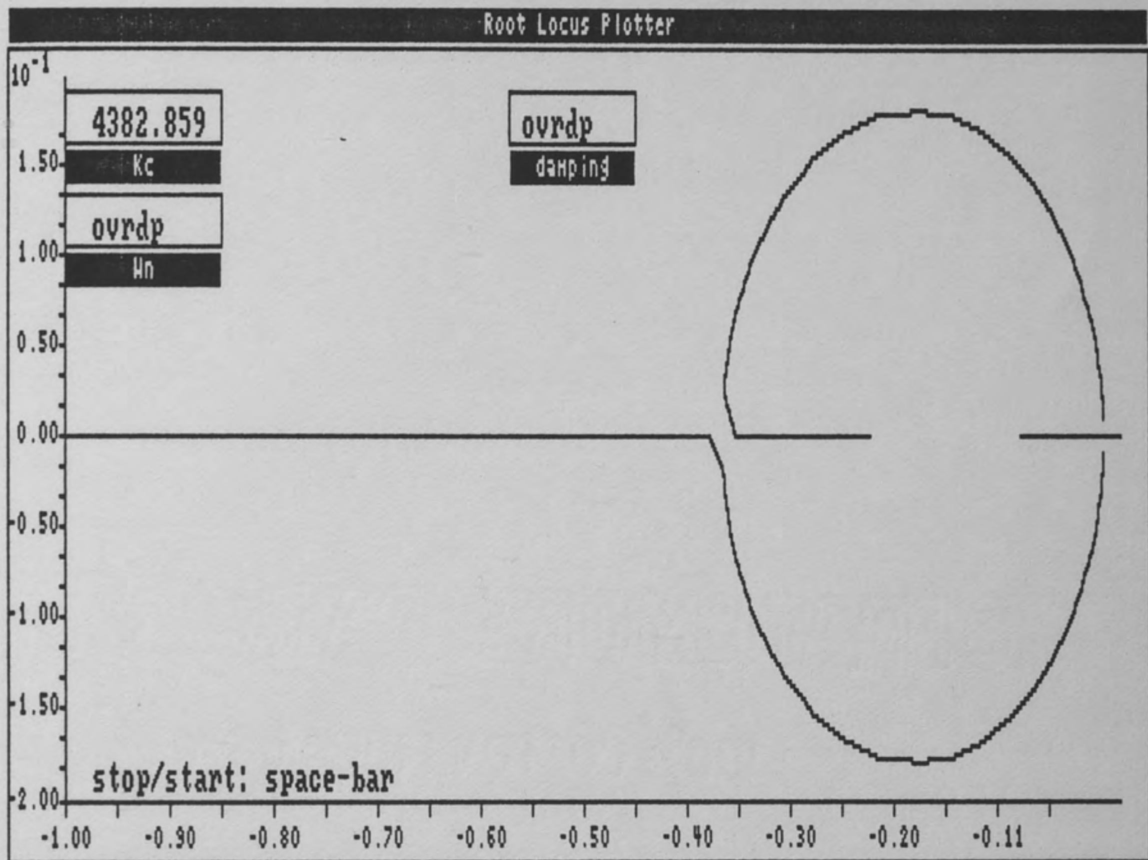
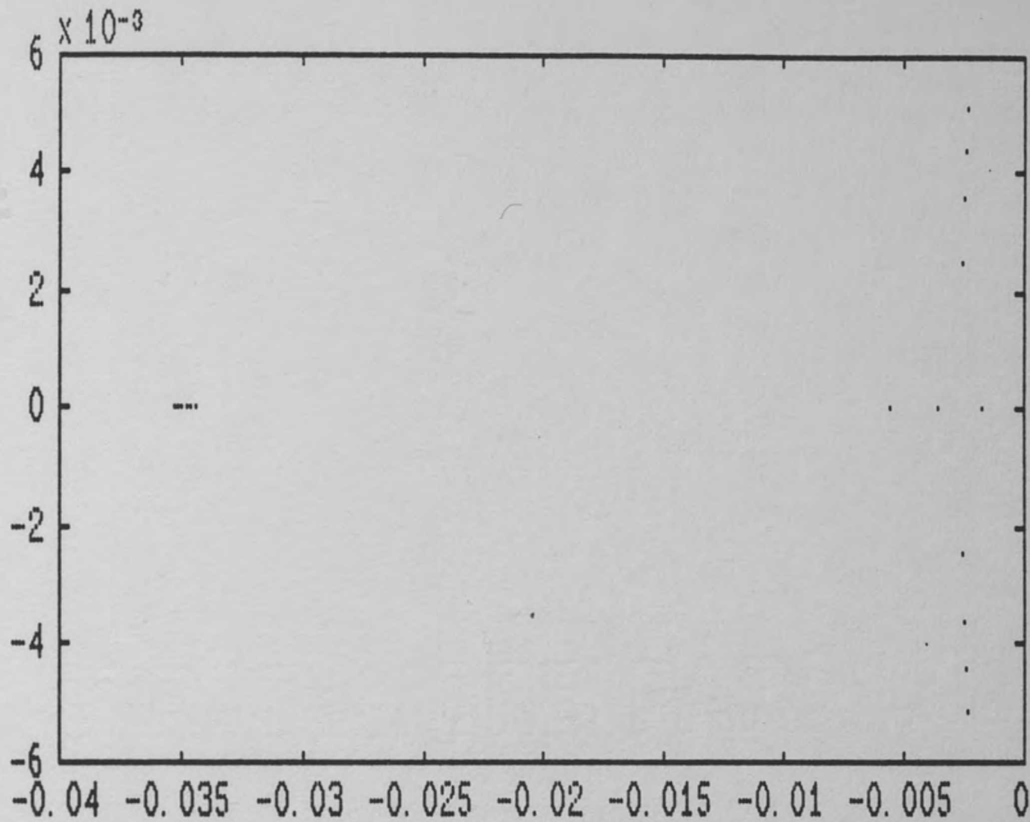


Figure 21. System Under P - D Control. Plot Using Variable Step Size




```
num =

    1.4600    1.4600    0.2920

/ den

den =

        26000          1040             5             0

/ r = rlocus(num,den,k);
/ r

r =

  -0.0344             -0.0056              0.0000
  -0.0346             -0.0036             -0.0018
  -0.0348             -0.0026 + 0.0025i   -0.0026 - 0.0025i
  -0.0350             -0.0025 + 0.0036i   -0.0025 - 0.0036i
  -0.0351             -0.0024 + 0.0044i   -0.0024 - 0.0044i
  -0.0353             -0.0024 + 0.0051i   -0.0024 - 0.0051i

/ plot(r,'.')
/
```

Figure 22. Sample Matlab Input and Output

APPENDIX II PROGRAM SOURCE CODE LISTING

```
{ a comment generally refers to the procedure
  directly following that comment}

procedure RootLoci (Kc, Ti, Td, areal, area2: real);


type
     stringy = string[ 78];
     Onedim = Array[ 1..100] of real;
     dummypointer = ^dummy;
     dummy = record { used to release heap pointer}
             end;

var heapaddress: dummypointer;
    apoints, bpoints, cpoints: PlotArray;
    II, JJ, Imax : integer;
    xlim, ylim, Xlo, Ylo: real;
    eps, r0, s0, R1, R2, Kp, Km, Kt,
    Kcsaved, Kcfinalsaved, stepsizesaved, Tisaved,
    Tdsaved: real;
    alpha, alphasaved: real;
    Kstore, Kstore2: Onedim;
    linerequested, abort, firstRun, modify, exit,
    thirdorder: boolean;
    message: stringy;


{ check that the response to an option message
  displayed is yes or no only}

procedure check_response( message: stringy;
                          requestnum: integer);
  var  inp: char;
       ValidChar: boolean;

  begin
    GotoXY( 35, 2);
    Textcolor( Lightred);
    write( '  OPTIONS  ');
    Textcolor( Yellow);
    Window( 1, 1, 80, 25);
    GotoXY( 34,13);
    writeln( message);
    GotoXY( 37,23);
         Textcolor( Lightgreen);
         write( 'Y');
         Textcolor( Cyan);
         write( 'es');
         Textcolor( Lightgreen);
         write( '  N');
```

```
        Textcolor( Cyan);
        write( 'o');
      repeat
        read (kbd, inp);
        inp := upcase( inp);
        case inp of
        'N':  exit := true;
        'Y':  case requestnum of
              1:  modify := true;
              2:  linerequested := true;
              3:  abort := true;
              4: ;
              end {case}
        else
          begin
            Sound (880);
            Delay (200);
            Nosound
          end {else}
        end; {case}
      validchar := inp in ['N','Y']
      until validchar;
      Textcolor( Yellow);
      Window(4, 23, 78, 24);
      ClrScr;
      Window( 1, 1, 80, 25)
 end; {check response}


{ display a warning or error, such as invalid
  input}

procedure display_error( message: stringy);

  begin
    Sound (880);
    Delay (200);
    Nosound;
    Window(3, 4, 78, 21);
    ClrScr;
    Window(1, 1, 80, 25);
    GotoXY( 3,5);
    TextBackground( red);
    TextColor( white);
    write( message);
    TextBackground( black);
    TextColor( Yellow);
    Delay( 3000);
    Window( 3, 4, 78, 21);
    ClrScr;
```

```
    Window( 1, 1, 80, 25)
  end;  {display error}


{ large supervisory routine, which does numerical
  calculations, and stores required root locus
  points in plot arrays.  All succeeding procedures
   are directly or indirectly controlled by this
   routine until graphics procedures are reached}

procedure getpoints;
  type TwoDimArray = array[ 1..1200, 1..2] of real;
       OneDimArray = array [1..3] of real;

  const TwoDimMax = 1200;
        OneDimMax = 3;

  var infile: text;
      delta_r, delta_s, r, s,
      capR, capS, Rr, Rs, Sr, Ss: real;
      Kcfinal, stepsize, rholim: real;
      n, l, capN, JJMax: integer;
      a, b,imagroot, realroot: OneDimArray;
      fwpoints: TwoDimArray;
      ap, bp, cp: PlotArray;
      in_range, breakaway: boolean;


  { The next five routines are within getpoints.
   All are involved with initialization of arrays
   which if not performed can yield undesirable side
   effects}

  procedure zeroOut( var thisarray: OneDimArray);
    var I: integer;

    begin
      for I := 1 to OneDimMax do
        thisArray[I] := 0.0
    end;


  procedure initialize1DArrays;

    begin
      zeroOut (a);
      zeroOut (b);
      zeroOut (imagroot);
      zeroOut (realroot)
    end;
```

```
procedure zero( var thisarray: TwoDimArray);
  var I,J: integer;

  begin
    for J := 1 to 2 do
    for I := 1 to TwoDimMax do
      thisArray[I,J] := 0.0
  end;


procedure NullOut( var thisarray: plotArray);
  var I,J: integer;

  begin
    for J := 1 to 2 do
    for I := 1 to MaxPlotGlb do
      thisArray[I,J] := 0.0
  end;


procedure initializePlotArrays;

  begin
    NullOut (apoints);
    NullOut (bpoints);
    NullOut (cpoints);
    NullOut (ap);
    NullOut (bp);
    NullOut (cp);
  end;


{ get auxiliary system parameters from a disk file
  if root locus procedure is called from editor}

procedure get_parameters;

  begin
    assign (infile,'RLINPUT.DTA');
    reset (infile);
    read (infile, eps, r0, s0);
    read (infile, areal, area2, R1, R2, Kp, Km, Kt);
    read (infile, Kc, Kcfinal, stepsize, Td, Ti);
    {initialize " saved" variables}
    Kcsaved := Kc;
    Kcfinalsaved := Kcfinal;
    stepsizesaved := stepsize;
    Tdsaved := Td;
```

```
    Tisaved := Ti
  end; {get_parameters}


{ if repeating the root locus procedure, then
  reset the controller parameters to what they were
  when first called from editor}

procedure reset_parameters;

  begin
    Kc := Kcsaved;
    Kcfinal := Kcfinalsaved;
    stepsize := stepsizesaved;
    Ti := Tisaved;
    Td := Tdsaved
  end; {reset parameters}


{ display the menu which allows user to change
 various controller parameters}

procedure submenu;
  var selecting, validnum: boolean;
    selection: char;


  {within submenu}

  procedure display_old( parameter: real);

  begin
    GoToXY( 20, 13);
    TextBackground( blue);
    TextColor( lightgray);
    write( parameter:9:3);
    TextBackground( black);
    TextColor( yellow);
  end;


  procedure display_new( parameter: real);

  begin
    GotoXY( 20, 13);
    TextBackground( blue);
    TextColor( yellow);
    write( parameter:9:3);
    TextBackground( black);
  end;
```

```
{ error checking to ensure that all inputs for
 controller parameters are within a valid range}

procedure check_real( var number: real);

  begin
      Textcolor( lightgreen);
      GotoXY( 38, 23);
      write( '? ');
      Textcolor( yellow);
      {$I-}
      readln( number);
      {$I+}
      if( (IOresult <> 0) or
         (number < 0 ) or (number > 1E5) or
          (Ti = 0.0)) then
            display_error
        ( '                          ' +
         'Invalid number.  Please re-enter.')
          else
            validnum := true;
      GotoXY( 38, 23);
      write( '                          ')
  end; {check real}


{gets user parameter changes from the keyboard}

procedure obtain( var param, paramsaved: real;
                  message: stringy; Y: integer);
     var tempparam: real;

     begin
       validnum := false;
       tempparam := param;

       repeat
         GoToXY( 25, Y);
         display_old( tempparam);
         write( message);
         check_real( param)
       until validnum;
       paramsaved := param;
       display_new( paramsaved);
       Delay( 1000);
     end; {obtain}
```

```
{display prompt for user}

procedure writehere( message: stringy; X, Y: integer);
  begin
    GoToXY( X, Y);
    write( message);
  end;


begin  {submenu}
selecting := true;
while selecting do begin
  Window (3, 4, 78, 21);
  ClrScr;
  Window( 1, 1, 80, 25);
  GotoXY( 30, 2);
  Textcolor( lightred);
  writehere( 'CONTROLLER', 35, 2 );
  Textcolor( yellow);
  writehere
  ( 'Change a parameter. Select one.', 25, 5);
  writehere( '1. Kc', 38, 12);
  writehere( '2. Td', 38, 13);
  writehere( '3. Ti', 38, 14);
  writehere( '_| no change', 34, 21);
  Textcolor( lightgreen);
  writehere( '? ', 38, 23);
  Textcolor( yellow);
  read (kbd, selection);
  Window (3, 4, 78, 21);
  ClrScr;
  Window( 1, 1, 80, 25);
  writehere( '        ', 38, 23);

  case selection of
  '1': begin
         obtain( Kc, Kcsaved,
          ' initial Kc  (Return: no change)', 13);
         obtain( Kcfinal, Kcfinalsaved,
          '          final Kc?            ', 14);
         obtain( stepsize, stepsizesaved,
          '          step size?          ', 15);
       end;
  '2':   obtain( Td, Tdsaved, '             Td   ', 13);
  '3':   obtain( Ti, Tisaved, '             Ti   ', 13);
  #13: selecting := false;
    else
       sound (660);
       delay (250);
       nosound
```

```
    end {case}
  end; {while}
  writehere( '                ', 35, 2)
  end; {submenu}


{from the various controller parameters and other
 system parameters construct the characteristic
 polynomial for the closed loop system}

procedure construct_polynomial;
  const Tlarge:integer = 100;
  var tauAtauB, tauAplustauB, K0: real;

  begin
    tauAtauB := areal * area2 * Rl * R2;
    tauAplustauB := areal * (Rl + R2) + area2 * R2;
    K0 := Kp * Km * Kc * Kt;
    a[1] := (tauAplustauB + (K0 * Td)) / tauAtauB;
    a[2] := (K0 + 1) / tauAtauB;
    a[3] := K0 / (tauAtauB * Ti);

    if (Ti >= Tlarge) then
      begin
        capN := 2;
        thirdorder := false
      end
    else
      begin
        capN := 3;
        thirdorder := true
      end;{else}
  end; {construct_polynomial}


{ this routine and its subroutines implement the Lin-
  Bairstow algorithm, as described in the text Ch. 3}

procedure roots_driver;

  const lmax: integer = 20;
  var k, m, j: integer;
    stop, continue: boolean;
    st, steps: string[ 6];


  procedure init_next_factor;

    begin
      n:= capN - (2 * m);
```

```
      l := 0;
      r := r0;
      s := s0;
  end; {init next factor}


  function testdisc (capR, capS: real): integer;

    var disc: real;

    begin
      disc := capR * capR - 4.0 * capS;
      if disc < 0 then
        testdisc := -1
      else
        if disc = 0 then
          testdisc := 0
        else
          testdisc := 1
    end; {testdisc}


  procedure quadroots (R, S:real);

    var rad: real;

    begin
      case testdisc(R, S) of
      -1: begin
            rad := sqrt (4.0 * S - R * R);
            realroot [J] := -R / 2.0;
            realroot [J+1] := -R / 2.0;
            imagroot [J] := rad /2.0;
            imagroot [J+1] := -rad / 2.0
          end;
       0: begin
            realroot [J] := -R /2.0;
            realroot [J+1] := -R / 2.0;
            imagroot [J] := 0.0;
            imagroot [J+1] := 0.0
          end;
       1: begin
            rad := sqrt (R * R - 4.0 * S);
            realroot [J] := (-R + rad) / 2.0;
            realroot [J+1] := (-R - rad) / 2.0;
            imagroot [J] := 0;
            imagroot [J+1] := 0
          end
      end {case}
    end; {quadroots}
```

```
procedure reduce_polynomial;

  begin
    b[1] := a[1] - r;
    b[2] := a[2] - (r * b[1]) - s;
    for k := 3 to n do
      b[k] := a[k] - (r * b[k-1]) - s * b[k-2];
    capR := b[n-1];
    capS := b[n] + (r * b[n-1]);
  end; {reduce polynomial}


procedure partials;

  var p, q: array [1..6] of real;

  begin
    p[1] := -1.0;
    p[2] := r - b[1];
    for k := 3 to n do
      p[k] := -b[k-1] - r * p[k-1] - s * p[k-2];
    Rr := p[n-1];
    Sr := p[n]+ (r * p[n-1]) + b[n-1];
    q[1] := 0.0;
    q[2] := -1.0;
    for k := 3 to n do
      q[k] := -b[k-2] - r * q[k-1] - s * q[k-2];
    Rs := q[n-1];
    Ss := q[n] + r * q[n-1]
  end; {partials}


procedure differential_corrections;

  var denom: real;

  begin
    denom := Rr * Ss - Rs * Sr;
    delta_r := (-capR * Ss + capS * Rs)/ denom;
    delta_s := (-Rr * capS + Sr * capR)/ denom;
    r:= r + delta_r;
    s := s+ delta_s;
  end; {diff'l corrections}


procedure replace_polynomial;

  var newN: integer;
```

```
  begin
    m := m+1;
    j := j+2;
    newN := capN - (2* m);
    for k:= 1 to newN do
    a[k] := b[k];
  end; {replace polynomial}


begin  {roots driver}
  abort := false;
  k := 0; m := 0; j := 1; stop := false;
  continue := false;

  while not stop do
    begin
      init_next_factor;
      if n < 2 then
        begin
          realroot [J] := -a[1];
          stop := true
        end
      else
       if n = 2 then
         begin
           quadroots(a[1],a[2]);
           stop := true
         end
       else
         begin
           repeat
             reduce_polynomial;
             partials;
             differential_corrections;
             l := l+1;
             stop := (l > lmax);
             continue := not stop and
               ((abs(delta_r) > eps)
               or (abs(delta_s) > eps));
           until not continue;
           if not stop then
             begin
               quadroots(r, s);
               replace_polynomial;
             end {if not stop}
             else  if (l > lmax) then      {stop true }
               begin
                 str( l:3,st);
                 str(eps:6,steps);
```

```
                    message :=  'roots have not'+
                    ' converged after ' + st  +
                    ' iterations to within '+ steps;
                    display_error( message);
                    check_response( '    Abort? ', 3);
                    quadroots( r, s);
                    replace_polynomial
                   end {else}
              end {else begin repeat}
          end {while}
        end; {roots_driver}


      { determine a world coordinate system.  Bypass the Turbo
        Graphix procedure  in order to condition the limits
        obtained.  Scan the array of points for max & min
        values}

      procedure FindXlimYlim( A:TwoDimArray; NPoints:integer);
        var j:integer;

        begin
          NPoints := abs( NPoints);
          if NPoints >= 2 then
            begin
              Xlim := A[ 1, 1];
              Ylim := A[ 1, 2];
              Xlo := Xlim;
              Ylo := Ylim;
              for j := 2 to NPoints do
                begin
                if A [ j, 1] > Xlim then
                  Xlim := A[ j, 1]
                else
                  if A[ j, 1] < Xlo then
                    Xlo := A[ j, 1];
                  if A[ j, 2] > Ylim then
                    Ylim := A[ j, 2]
                  else
                    if A[ j, 2] < Ylo then
                      Ylo := A[ j, 2]
                end; {for}
            end {if NPoints}
          else error (7,4);
        end; {FindXlimYlim}


      {after calculating the roots of a polynomial for
       some Kc, store that point in an array}
```

```
procedure fillarrays
         ( var Aarray, Barray, Carray: plotarray;
           var Karray: Onedim);

  begin
    II := II + 1;
    Aarray [II,1] := realroot [1];
    Aarray [II,2] := imagroot [1];
    JJ := JJ + 1;
    fwpoints[JJ,1] := Aarray[II,1];
    fwpoints[JJ,2] := Aarray[II,2];

    Barray [II,1] := realroot [2];
    Barray [II,2] := imagroot [2];
    JJ := JJ + 1;
    fwpoints[JJ,1] := Barray [II,1];
    fwpoints[JJ,2] := Barray [II,2];
    if thirdorder then
      begin
        Carray [II,1] := realroot [3];
        Carray [II,2] := imagroot [3];
        JJ := JJ + 1;
        fwpoints [JJ,1] := Carray [II,1];
        fwpoints [JJ,2] := Carray [II,2];
      end; {if}
    Karray[ II] := Kc
  end; {fillarrays}


  {ensure that user requirements are not too large
   or too small for the arrays}

  procedure test_range;
    var st: string[8];

    begin
      if  ( II > MaxPlotGlb) or  ( II < 1) then
        begin
          II := MaxPlotGlb + 1;
          str( II:8, st);
          in_range := false;
          message := ' # of points to be plotted, '+ st
          + #13#10 + 'is too large.'+
          ' Please change requirements';
          display_error( message);
          Delay(3000)
        end {if}
     end; {testrange}
```

```
{if a fixed increment is used for Kc, then pass 1
 obtains the required array of points.  Otherwise
 these points are necessary for comparison when
 calculating a variable step size}

procedure pass1;

  begin
    Window( 3, 4, 78, 21);
    ClrScr;
    Window(1, 1, 80, 25);
    GotoXY( 35, 2);
    Textcolor( lightred);
    write('CALCULATING');
    GoToXY(29, 12);
    TextColor( yellow);
    TextBackground( black);
    write( '# of points to be plotted:' );
    TextColor( Brown);
    repeat
      construct_polynomial;
      roots_driver;
      if (not abort) then
      begin
        fillarrays( ap, bp, cp, Kstore);
        GoToXY( 39,13);
        write( II:3);
        test_range;
        Kc := Kc + stepsize
      end {if not abort}
    until ( Kc > Kcfinal) or abort or ( not in_range);
    TexTColor( Yellow);
  end; {pass1}


{if a variable step size is required, a scan is made
 of the distance between points obtained in pass 1.
 for each pair of adjacent points a linear inter-
 polation is made to obtain the desired step size.
 New points are calculated and stored in separate
 arrays}

procedure pass2;
  var count, count2, imin: integer;
      maskbreakout, maskbreakin: boolean;


  {within pass2.  Does linear interpolation as per
   text Ch. 4.  Upon break away or reentry at x axis,
   the step size is additionally reduced for a few
```

```
 steps}

procedure getstepsize;
  const small = 1E-6;
  var aNum, deltaA1, deltaA2, deltaA: real;
      i: integer;
      stopcondition, breakout, breakin: boolean;

  begin
    i := imin;

    repeat
      if (i < Imax) then
        stopcondition := ( Kstore[ i-1] <= Kc ) and
                         ( Kc < Kstore[ i])
      else
        begin
        stopcondition := ( Kstore[ i-1] <= Kc);
        end;
      i := i + 1;
    until stopcondition;

    i := i - 1;
    imin := i;
    deltaA1 :=  ap[ i, 1] - ap[ i-1, 1];
    deltaA2 :=  ap[ i, 2] - ap[ i-1, 2];
    deltaA := sqrt( sqr( deltaA1) + sqr( deltaA2) );
    stepsize := stepsizesaved *
             ( rholim /  deltaA) * ( 1.0/60.0 );


    breakout :=   ( abs( ap[ i, 2]) > small) and
                  ( abs( ap[ i-1, 2]) < small);
    breakin  :=   ( abs( ap[ i, 2]) < small) and
                  ( abs( ap[ i-1, 2]) > small);
    if not maskbreakout then
      if breakout then
        count := 5;
    if count > 0 then
      begin
        count := count - 1;
        maskbreakout := true;
        stepsize := stepsize/ 5
      end;

    if not maskbreakin then
      if breakin then
        count2 := 10;
    if count2 > 0 then
      begin
```

```
          count2 := count2 - 1;
          maskbreakin := true;
          stepsize := stepsize/ 3
        end;
      end; { get step size}

    begin {pass 2}
      maskbreakout := false;
      maskbreakin := false;
      count := 0;
      count2 := 0;
      II := 1;
      imin := 2;
      Kc := Kcsaved;
      rholim := sqrt( sqr( Xlim-Xlo) + sqr( Ylim-Ylo) );
      TextColor( Brown);

      repeat
        getstepsize;
        Kc := Kc + stepsize;
        construct_polynomial;
        roots_driver;
        if not abort then
          begin
            fillarrays( apoints, bpoints,
                        cpoints, Kstore2);
            GoToXY( 39,13);
            write( II: 3);
            test_range
          end {if not abort}
      until ( Kc > Kcfinal)  or abort or ( not in_range);

      TextColor( Yellow)
    end; {pass2}


begin  {getpoints}
  initialize1DArrays;
  initializePlotArrays;
  zero( fwpoints);
  repeat
    in_range := true;
    if firstRun then
      begin
        get_parameters;
        firstRun := false
      end
    else
      reset_parameters;
    submenu;
```

```
    modify := false;
    Window (3, 4, 78, 21);
    ClrScr;
    Window (1, 1, 80, 25);
    check_response( 'Vary step size? ', 1);
    linerequested := false;
    check_response( 'Lines instead of points? ', 2);
    II := 0; {index associated with plotarrays}
    JJ := 0; {index associated with fwpoints array only}
    pass1;
    if (not abort) and ( in_range) then
        begin
          Imax := II;
          JJmax := JJ;    {was set in fillarrays,
                           now used to find Xlim, Ylim}
          findXlimYlim( fwpoints, JJmax);
          if modify then
            pass2
          else
            begin
              apoints := ap;
              bpoints := bp;
              cpoints := cp;
              Kstore2 := Kstore
            end {else}
      end {if not abort}
  until in_range
end; {getpoints}


(* * *   g r a p h i c s   r o u t i n e s       * * *)


{initializes the root locus graph by using initial-
 ization procedures from the Turbo Graphix Toolbox.
 Incorporates a round off procedure to make the
 graph limits round numbers}

procedure initializeGraph;
  var
    XMinAdj, YMinAdj, XMaxAdj, YMaxAdj: integer;


  function round_off( number:real): real;
     var firstDigit, exponent, stringnum: string[8];
         num, code, digit: integer;
         temp: real;

     begin
```

```
      str( number:8, firstDigit);
      str( number:8, exponent);
      if number >= 0 then begin
     delete( firstDigit, 2, 7);
      val( firstDigit, digit, code);
      num := 1 + digit end
      else begin
      delete( firstDigit, 3, 6);
      val( firstDigit, digit, code);
      num := -1 + digit end;
      delete( exponent, 1, 4);
      str( num, stringnum);
      val( stringnum + exponent, temp, code);
      round_off := temp;
    end;

begin
  DefineWindow (1, 0, 0, XMaxGlb, YMaxGlb);
  DefineHeader (1, 'Root Locus Plotter');
  XMinAdj :=  4;
  YMinAdj :=  16;
  XMaxAdj := XMaxGlb-2;
  YMaxAdj := YMaxGlb-14;

  DefineWindow (2, XMinAdj, YMinAdj, XMaxAdj, YmaxAdj);
  DefineWindow (3, XMinAdj, YMinAdj + 4,
                   XMinAdj + 10, YMinAdj + 26);
  DefineHeader (3, 'Kc');
  DefineWindow (4, trunc( XMaxAdj/2.0) - 3, YMinAdj +4,
                   trunc( XMaxAdj/2.0) + 5,
                   YMinAdj + 26 );
  DefineHeader (4, 'damping');
  DefineWindow (5, XMinAdj, YMinAdj + 28, XMinAdj + 10,
                   YMinAdj + 50);
  DefineHeader (5, 'Wn');
  DefineWindow (6, XMinAdj, YMinAdj + 52, XMinAdj + 10,
                   YMinAdj + 74);
  DefineHeader (6, 'Ti');
  DefineWindow (7, XMinAdj, YMinAdj +100, XMinAdj + 10,
                   YMinAdj + 122);
  DefineHeader (7, 'Td');
  SelectWindow (1);
  SetHeaderOn;
  DrawBorder;
  IMax := II;

  {final value II was set by fill arrays,
   used for repeat control below}

  if linerequested then
```

```
      begin
        II := 2;

         {reset plot array index to beginning,
         #2 required for DrawLine.  Now first
         two points equal to remove side effects.}

        apoints[ 1, 1] := apoints[ 2, 1];
        apoints[ 1, 2] := apoints[ 2, 2];
        bpoints[ 1, 1] := bpoints[ 2, 1];
        bpoints[ 1, 2] := bpoints[ 2, 2];
        cpoints[ 1, 1] := cpoints[ 2, 1];
        cpoints[ 1, 2] := cpoints[ 2, 2]
      end
    else
      II := 1;

    if ( abs( Ylo) < eps) and ( abs( Ylim) < eps) then
      begin
        Ylo :=  -1.0;
        Ylim := 1.0
      end;
    Xlo := round_off( Xlo);
    Ylo := round_off( Ylo);
    Ylim := round_off( Ylim);
    DefineWorld( 1, Xlo, Ylo, Xlim, Ylim);

    SelectWindow (3);
    SetHeaderToBottom;
    SetHeaderOn;
    DrawBorder;
    SelectWindow (4);
    SetHeaderToBottom;
    SetHeaderOn;
    DrawBorder;
    SelectWindow (5);
    SetHeaderToBottom;
    SetHeaderOn;
    DrawBorder;

  end; {initializeGraph}


{next two subroutines are from the Toolbox, and are
 used to draw point positions which pop up when the
 plotting is temporarily stopped by the user}

function StringNumber(X1:real;
                           MaxExponent:integer):wrkstring;
  var y:wrkstring;
```

```
  begin
    str(X1*exp(-MaxExponent*ln(10.0)):5:2,y);
    StringNumber:=y;
  end;


function GetExponent(X1:real):integer;

  begin
    GetExponent:=0;
    if X1<>0.0 then
      if abs(X1)>=1.0 then GetExponent:=
        trunc(ln(abs(X1))/ln(10.0))
      else GetExponent:=
      -trunc(abs(ln(abs(X1)))/ln(10.0)+1.0);
  end;


{when user pauses graphing, various numbers and
 windows display from a virtual memory screen.
 The user may resume graphing, whereupon these
 items vanish, or he may elect to terminate
 the graph prematurely (esc)}

procedure pop_up;
  var sta1, sta2: string[ 5];
      sta: string[ 13];
      cposY, aposX, bposX, bposY: real;
      maxexponentX, maxexponentY: integer;

  begin
    CopyScreen;
    SelectWindow( 6);
    SetHeaderOn;
    DrawBorder;
    GoToXY (7, 10);
    write (Ti:5:2);
    SelectWindow( 7);
    SetHeaderOn;
    DrawBorder;
    GoToXY (7, 16);
    write (Td:5:2);
    SelectWindow( 2);
    maxexponentX := GetExponent( Xlo);
    maxexponentY := GetExponent( Ylo);

    sta1 := StringNumber( apoints[ II, 1], maxexponentX);
    sta2 := StringNumber( apoints[ II, 2], maxexponentY);
    sta := sta1 + ',' + sta2;
```

```
    aposX :=    apoints[II, 1] - 0.25 *
                exp( maxexponentX * ln(10.0));
    drawtextW( aposX, -apoints[II, 2], 1, sta);

    sta1 := StringNumber( bpoints[ II, 1], maxexponentX);
    sta2 := StringNumber( bpoints[ II, 2], maxexponentY);
    sta := sta1 + ',' + sta2;
    bposX :=    bpoints[II, 1] - 0.25 *
                exp( maxexponentX * ln(10.0));
    bposY :=    bpoints[II, 2] - 0.25 *
                exp( maxexponentY * ln(10.0));
    drawtextW( bposX, -bposY, 1, sta);

    sta1 := StringNumber( cpoints[ II, 1], maxexponentX);
    sta2 := StringNumber( -cpoints[ II, 2], maxexponentY);
    sta := sta1 + ',' + sta2;
    {reverses maxexponent to base 10 number};
    cposY :=   -cpoints[II, 2] - 0.25 *
               exp( maxexponentY * ln(10.0));
    drawtextW( cpoints[ II, 1], -cposY, 1, sta);

    GotoXY( 39,23);
    write( 'Esc')
  end;


{actually plots the points which were stored in arrays.
 Also writes design information into auxiliary windows}

procedure graphit;
  var  theta, zeta, Wn, Temp: real;
       maxexponentX: integer;
       ch: char;


  {controlled by graphit.  Checks the keyboard
   at each plot iteration to see if the user
   wishes to pause and view auxiliary information,
   or abort}

  procedure check_keyboard_buffer;
    var stopped: boolean;

    begin
    if KeyPressed then
      begin
        read( kbd, ch);
        if ( ch = #32) then
          begin
            stopped := true;
```

```
          Sound (880);
          Delay (200);
          NoSound;
          pop_up;
          while  stopped  do
            begin
              read (kbd, ch);
              if (ch = #32) and (II < Imax) then
                stopped := false
              else
                if (ch = #27) then
                  begin
                    II := IMax + 1;
                    stopped := false
                  end
            end; {while}
          if ch = #32 then
            SwapScreen
        end {if ch}
    end {if KeyPressed}
  end; {check keyboard buffer}


{write root locus variables while plotting}

procedure write_to_subwindows;

  begin
    GoToXY ( 7, 4);
    Write (Kstore2 [II]:5:3);

    if ( apoints[ II, 1] < 1E-6) and
       ( apoints[ II, 2] < 1E-6) then
      zeta := 1.0
    else
      begin
        theta :=  ArcTan
        ( apoints[ II,2] / apoints[ II,1]);
        zeta := cos (theta)
      end;

    Wn := sqrt( sqr( apoints[ II,2]) +
            sqr( apoints[ II,1]));
    GoToXY ( 37,4);
    if zeta < 1 then
      begin
        write (zeta:5:2);
        GoToXY( 7,7);
        write( Wn:7:4)
      end
```

```
      else
        begin
          write( 'ovrdp');
          GotoXY( 7,7);
          write( 'ovrdp')
        end
    end; {write to sub windows}

begin {graphit}
  initializeGraph;
  with World[ 1] do     {invert world coord
                         to Cartesian type coord}
    begin
      Temp := Y1;
      Y1 := Y2;
      Y2 := Temp
    end; {with}
  SelectWorld (1);
  SelectWindow (1);
  DrawAxis( 8,7,0,0,0,0, -1, 0, false);
  GotoXY ( 7,23);
  writeln
  ('stop/start: space-bar ');
  SelectWindow( 2);
  maxexponentX := GetExponent( XLo);
  Temp :=     0.1 *
             exp( maxexponentX * ln(10.0));
  Drawline( Temp, 0, -Temp, 0); {cross hair, origin}

  repeat
    if linerequested then
      begin
        DrawLine (apoints[ II-1, 1], -apoints[ II-1, 2],
                  apoints[ II, 1], -apoints[ II, 2]);
        DrawLine (bpoints[ II-1, 1], -bpoints[ II-1, 2],
                  bpoints[ II, 1], -bpoints[ II, 2]);
        if thirdorder then
          DrawLine (cpoints[ II-1, 1],
                    -cpoints[ II-1, 2],
                     cpoints[ II, 1], -cpoints[ II, 2])
      end
    else
      begin
        DrawPoint (apoints[II,1], -apoints[II,2]);
        DrawPoint (bpoints[II,1], -bpoints[II,2]);
        if thirdorder then
            DrawPoint (cpoints[II,1], -cpoints[II,2])
      end; {else}
    write_to_subwindows;
    Delay (1000);
```

```
        check_keyboard_buffer;
        II := II + 1;
      until II > IMax;

      if ch <> #27 then
        repeat until KeyPressed;
    end; {graphit}


begin {root loci}
  firstRun := true;
  putframe;
  repeat
    mark( heapaddress);
    getpoints;
    if not abort then
      begin
        InitGraphic;
        ClearScreen;
        HiResColor(Yellow);
        graphit;
        LeaveGraphic
      end;
    exit := false;
    ClrScr;
    putframe;
    check_response( 'Continue root locus? ',4);
    release( heapaddress)
  until exit
end; {root loci}
```

APPENDIX III "INCLUDE" FILES - NOTES

I. Since nesting is not allowed, Include Files used in this subprogram were compiled as part of the editor program (Fadden 1986). Include Files required are:

<u>Turbo Graphix Toolbox files (Borland 1985)</u>

TYPEDEF.SYS

KERNEL.SYS

GRAPHIX.SYS

WINDOWS.SYS

AXIS.HGH

4X6.FON

8X8.FON

<u>Editor files (Fadden 1986)</u>

TEXTBOX.INC

PUTFRAME.INC

II. The following changes were made to Turbo Graphix Toolbox version 1.05:

A. file TYPEDEF.SYS

change MaxPlotGlb = 400.

B. files TYPEDEF.SYS, KERNEL.SYS, GRAPHIX.SYS, WINDOWS.SYS, AXIS.HGH

change Window to WWindow

C. file AXIS.HGH

1. change line 193 to

NPoints := Delta div 29

2. change line 199 to

   ```
   for i := 1 to NPoints - 1 do
   ```

3. change line 201 to

   ```
   xs := xs + 29 + Balance
   ```

LIST OF REFERENCES

Applied i. Tutsim User's Manual for IBM PC Computer. Palo Alto, Ca.: Applied i, 1985.

Borland International, Inc. Turbo Pascal Reference Manual. Scotts Valley, Ca.: Borland International, Inc., 1985.

Borland International, Inc. Turbo Graphix Toolbox Owner's Handbook. Scotts Valley, Ca.: Borland International, Inc., 1985.

Dorn, William S. and McCracken, Daniel D. Numerical Methods with Fortran IV Case Studies. New York: John Wiley and Sons, 1972.

Fadden, Leon. "Editor Design in the Context of Control System Simulation." M.S.E. thesis, University of Central Florida, 1986.

Klee, Harold I. University of Central Florida, Orlando, Fl. Unpublished textbook manuscript, 1986.

Moler, C., Little, J., Bangert, S., and Kleiman, S. PC-MATLAB for MS-DOS Personal Computers. Portola Valley, Ca.: The Mathworks, Inc., 1985.

McCalla, Thomas R. Introduction to Numerical Methods and Fortran Programming. New York: John Wiley and Sons, 1967.

Smith, Carlos A.; Corripio, Armando B. Principles and Practice of Automatic Process Control. New York: John Wiley and Sons, 1985.

TurboPower Software. Turbo Extender User's Manual and Reference Guide. Campbell, Ca.: TurboPower Software, 1986.

Weber, Thomas W. An Introduction to Process Dynamics And Control. New York: John Wiley and Sons, 1973.