THESIS

MICROCOMPUTER BASED LINEAR SYSTEM DESIGN TOOL

by
Roy L. Wood, Jr.

September 1986

Thesis Advisor: George J. Thaler

Approved for Public Release; Distribution Unlimited
DISCLAIMER NOTICE

THIS DOCUMENT IS BEST QUALITY PRACTICABLE. THE COPY FURNISHED TO DTIC CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.
MICROCOMPUTER BASED LINEAR SYSTEM DESIGN TOOL

The theory, algorithms, and operation of a continuous-time, linear control system design computer program is presented. The program, LCS-CAD, was developed to demonstrate automated transfer function block manipulation in conjunction with such classical control design techniques as Bode, Nyquist single and two-parameter root locus, and time domain response. Both numeric data and high-resolution graphs are available to the user. The software, which is completely interactive and menu driven, is written in structured Pascal to be run on the IBM-PC microcomputer.
Microcomputer Based Linear System Design Tool

by

Roy L. Wood, Jr.
Lieutenant, United States Navy
B.S.C.S., Texas A & M University, 1980

Submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the
NAVAL POSTGRADUATE SCHOOL
September 1986

Author: Roy L. Wood, Jr.

Approved by: George J. Thaler, Thesis Advisor
Harriett Rigas, Second Reader

Harriett Rigas, Chairman, Department of Electrical and Computer Engineering

John N. Dyer
Dean of Science and Engineering
ABSTRACT

The theory, algorithms, and operation of a continuous-time, linear control system design computer program is presented. The program, LCS-CAD, was developed to demonstrate automated transfer function block manipulation in conjunction with such classical control design techniques as Bode, Nyquist, single and two-parameter root locus, and time domain response. Both numeric data and high-resolution graphs are available to the user. The software, which is completely interactive and menu driven, is written in structured Pascal to be run on the IBM-PC microcomputer.
# TABLE OF CONTENTS

I. LINEAR CONTROL SYSTEM COMPUTER AIDED DESIGN ........ 6
   A. INTRODUCTION TO LCS-CAD .......................... 6
   B. LINEAR CONTROL SYSTEMS AND BLOCK DIAGRAMS ..... 6
   C. PROGRAM FEATURES ................................ 8

II. COMPUTER-AIDED SYSTEM DESIGN STEPS .................. 10
   A. DEVELOPING A MODEL ................................. 10
   B. ANALYZING THE MODEL ............................... 13
   C. APPLYING THE SPECIFICATIONS ..................... 25
   D. SUMMARY ........................................... 31

III. DETAILED PROCEDURE MODULE DESCRIPTIONS ............. 32
   A. PROGRAM OVERVIEW .................................. 32
   B. INPUT/CHANGE MENU ROUTINES ....................... 34
      1. Input Menu Hierarchy ............................. 34
      2. Input Utility Functions ........................... 35
      3. Blocks Record .................................. 39
      4. RootFinder Module ................................ 41
      5. Polynomial Expansion Routine ................... 44
      6. Building an Equivalent Loop Block ............... 48
      7. Input Procedure ................................ 50
      8. Change Procedure ................................ 52
      9. Add and Delete Blocks ............................. 53
     10. Save Blocks to Disk ............................... 54
     11. Retrieve Problem from Disk File ................. 55
12. Using Input Routines to Build Complex Functions

C. LOCATION OF CHARACTERISTIC EQUATION ROOTS

D. FREQUENCY ANALYSIS
   1. Nyquist Plot
   2. Bode Plot

E. ROOT LOCUS

F. TWO-PARAMETER ROOT LOCUS

G. TIME RESPONSE

H. UTILITIES

I. HELP SCREENS

IV. SUMMARY AND RECOMMENDATIONS

APPENDIX A INPUT UTILITIES

APPENDIX B PROGRAM LISTING

LIST OF REFERENCES

BIBLIOGRAPHY

INITIAL DISTRIBUTION LIST
A. INTRODUCTION TO LCS-CAD

LCS-CAD is a software tool to aid in the analysis and design of continuous-time, linear control systems. It was designed to allow a user who is familiar with the "classical" design tools, such as Bode, Nyquist, and root locus, to apply these methods while eliminating much of the tedium.

The software system is completely menu-driven and attempts to be user-friendly in a number of ways. First, the hierarchical menu structure is only two levels deep at any point so the user will not become "lost" in the program. Second, user inputs are systematically verified and validated. Third, the user has access to a powerful "change" facility that allows erroneous system information to be changed easily. Finally, the program makes use of the concept of "transfer function blocks" for data entry and manipulation.

B. LINEAR CONTROL SYSTEMS AND BLOCK DIAGRAMS

The behavior of many physical systems can be described by linear differential equations, or at least approximated this way. Of these, many may be described as single-input, single-output systems. For example, simple electrical
circuits are often modelled as linear systems with single inputs, say \( V_{in} \), and a single output, \( V_{out} \). It is possible to think of these systems as "black boxes" being acted on by an input and producing an output. Inside these boxes, then, would be a function to translate the input "signal" to the output. This so called "transfer function" is very convenient when dealing with system models.

After the differential equations for a linear, time-invariant system with zero initial conditions are determined, the Laplace transform can be applied yielding algebraic equations in the complex variable "s". After some manipulation, the equations can be written as a ratio of output to input. This is the standard transfer function form.

Unfortunately, very large systems can have extremely complex transfer functions which are difficult to work with. Often, however, such problems can be divided into smaller sub-problems, and these sub-problems can be modelled as independent transfer function blocks. The blocks can then be analyzed separately, if necessary, and later recombined for overall system analysis. Recombination, or reduction, of transfer function blocks is done in accordance with the rules of "block diagram
algebra", or alternatively, by applying "Mason's rule".\footnote{These block reduction techniques are described in detail in virtually all elementary control theory textbooks.}

LCS-CAD is a loop-based system. That is, the program operates on a single path or closed loop at a time. This simplifies the computations required to automatically reduce the loop's blocks to single equivalent block transfer function. Since the reduction is done automatically, the user is free to concentrate on analysis and design and not the tedious process of manipulating block transfer functions.

C. PROGRAM FEATURES

LCS-CAD is a large program and requires at least 256K of memory to run. It will run on any IBM-PC or compatible "MS-DOS" computer and requires a standard IBM Color Graphics Adapter (CGA). It will run on either monochrome or color monitor, but the menus are color coded and are easier to work with if a color monitor is available. The graphics are in high-resolution (640x200) mode and only appear in white-on-black. The graphics can be dumped to an IBM-Graphics, Epson, or compatible printer.

The program is written in Turbo Pascal (tm) and, due to its limitation to 64K code and data segment sizes, LCS-CAD was compiled as a single main executable program and six "chain" files. The chain files are essentially programs compiled without a run-time support module. They,
therefore, are not themselves executable. These program modules include:

(1) CAD.COM {the executable main module}
(2) INPUT.CHN {the input and change routines}
(3) FREQ.CHN {Bode and Nyquist modules}
(4) TIMERESP.CHN {the time response simulation module}
(5) TWOPARAM.CHN {the two parameter root locus module}
(6) UTILMENU.CHN {the utilities menu and routines}
(7) HELPMENU.CHN {the help menu and all help screens}

In addition to these files, there are two additional "system" files needed to run the program, 4x6.FON, and ERROR.MSG. The .FON file is a graphics lettering font used by the graphics routines to print letters and numbers on-screen. The ERROR.MSG file is a Turbo Pascal file containing various error messages.

The next chapter presents a detailed example using the LCS-CAD program to analyze and design a controller for a simple motor. From that discussion, many of the features of the program can be examined. The third chapter details each individual program module and outlines the theory and algorithms used to construct the LCS-CAD program. Chapter Four summarizes the work done already in LCS-CAD and proposes additions and further work which would make the program more useful and versatile.
II. COMPUTER-AIDED SYSTEM DESIGN STEPS

A. DEVELOPING A MODEL

The first step in the successful design of a linear system is to formulate a mathematical model which describes the system. Once modelled, the system can be analyzed and a suitable control system can be designed.

Consider, for example the simple armature controlled dc motor driving a load as shown in Figure 2-1. The left side of the figure shows the armature circuit including its resistive and inductive components. The mathematical model for this part of the motor can be derived from Kirchhoff’s circuit law. The mechanical side of the arrangement, shown on the right, includes the motor generated torque, the load inertia, and the viscous damping effect. Newton’s laws will provide the basis for modelling the mechanical portion of the motor/load combination.

The differential equation for the armature circuit is:

\[ L_a \frac{di}{dt} + R_a i + e_b = e_a \]  \hspace{1cm} (2.1)

Similarly, the description of the mechanical system is a differential equation in shaft angle, but this time of second order.

\[ J \frac{d^2 \theta}{dt^2} + f \frac{d \theta}{dt} = T \]  \hspace{1cm} (2.2)
Knowing that for an armature controlled dc motor the field current, $i_f$, is constant and, therefore, the magnetic flux is constant, the motor torque is directly proportional to the armature current

$$ T = k_i a $$  \hspace{1cm} (2.3) \hspace{1cm} .$$

Also, under constant flux conditions, the back electromotive force, $e_b$, will be directly proportional to the motor shaft's angular velocity

$$ e_b = k_b \frac{d\theta}{dt} $$  \hspace{1cm} (2.4) \hspace{1cm} .$$

These equations are sufficient to mathematically represent the dynamical behavior of the dc motor. If zero
initial conditions are assumed, then the differential equations can be written in their Laplace transform representation:

\[(J_s^2 + f_s) \theta(s) = T(s) = K I_a(s) \quad (2.5).\]

\[(L_a s^2 + R_a) I_a(s) + E_b(s) = E_a(s) \quad (2.6).\]

\[K_b s \theta(s) = E_b(s) \quad (2.7).\]

With the armature voltage as input and the motor shaft angle as output, the transfer function relation of the system can be formulated as shown below:

\[
\theta(s) = \frac{K}{E_a(s)} \quad (2.8).
\]

Since the typical armature inductance of this type of dc motor is very small, it will be neglected. This greatly simplifies the transfer function without inducing much error due to the approximation as can be seen in Equation 2.9 below.

\[
\theta(s) = \frac{K_m}{E_a(s)} \quad (2.9)
\]

where \(K_m = \frac{K}{(R_a + K K_b)}\) Motor gain constant

\(T_m = \frac{R_a J}{(R_a + K K_b)}\) Motor time constant

Using the numerical values for the motor given in Figure 2-1 the transfer function becomes

\[
\theta(s) = \frac{100}{E_a(s)} \quad (2.10)
\]

\[s (0.1 s + 1)\]
B. ANALYZING THE MODEL

Now that the system has been modelled, the next step is to examine the model's behavior before a controller is designed and installed. This step will tell the designer whether the system will require stabilization or simply "fine-tuning" to meet the design specifications. The computer can be used to assist in the analysis process.

The first step, of course, is to enter the model into the computer program. Working from the system's block diagram, shown in Figure 2-2, we begin the computer program. From the opening menu of LCS-CAD, shown here in Figure 2-3, we choose the "Input/Change Transfer Function(s)" option to bring us to the secondary menu shown in Figure 2-4.

![Figure 2-2. DC Motor Equivalent Block Diagram](image)
**MAIN MENU**

- <I> Input/Change Transfer Function(s)
- <L> Location of Char. Eq. Roots
- <F> Frequency Analysis
- <R> Root Locus Analysis
- <P> Two-Parameter Root Locus
- <T> Time Response
- <U> Utilities
- <H> Help
- <Q> Exit Program

Press Your Selection

Figure 2-3. Main Menu

**INPUT/CHANGE MENU**

- <I> Input Block Transfer Function(s)
- <C> Change Block in Current Loop
- <A> Add a Block to Current Loop
- <D> Delete a Block from Current Loop
- <S> Save Current Loop to Disk File
- <R> Retrieve Problem from Disk File
- <H> Help
- <Q> Exit to Main Menu

Press Your Selection

Figure 2-4. Input/Change Menu
To initially enter the transfer function of the motor, select the <I> Input Block Transfer Function(s) option. The next screen will prompt for number of blocks in the current loop (only one in the case of the motor), and whether the block will be input from the keyboard or from a disk file. Since this is the initial input for this problem, it must be from the keyboard. Next, the block input page, as shown below in Figure 2-5, will appear.

<table>
<thead>
<tr>
<th>Block Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block 1:</td>
</tr>
<tr>
<td>Is block in Forward (F) or Feedback (B) Path?</td>
</tr>
<tr>
<td>What is the order of the Numerator? 0</td>
</tr>
<tr>
<td>What is the order of the Denominator? 2</td>
</tr>
<tr>
<td>What is the block gain constant? 100</td>
</tr>
<tr>
<td>Will you enter the block in Factored (F) or Coefficient (C) form?</td>
</tr>
</tbody>
</table>

Figure 2-5. Block Input Screen

The options here allow input of both forward path and feedback path blocks in any order. An input block can have numerator and/or denominator polynomial up to ninth order. There is provision for overall block gain input, and finally, the user is allowed to input the transfer function for the block in either coefficient or factored form.
The limitation of a block to order nine is artificial here, and systems up to thirtieth order can be input by breaking the block transfer functions into smaller order blocks. For example, if the transfer function were

\[
\frac{(s^2 + 4s + 10)(s + 2)^2}{s^5 (s^4 + 2s^3 + 6s^2 + 1)}
\]

then the function could be divided into two or more blocks as shown here

\[
\frac{(s^2 + 4s + 10)}{(s^4 + 2s^3 + 6s^2)} \times \frac{(s + 2)^2}{s^5(s^2 + 1)}
\]

with each block less than order nine and system order less than thirty.

The next screen shown will vary depending on whether the input is to be in factored or coefficient form. Figure 2-6 shows the factored input screen, and Figure 2-7 shows the coefficient form input screen.

***Block Transfer Function Input***

---

DENOMINATOR Transfer Function Input -- FACTORED Form

\[
s = \quad +j \quad \\
s = \quad +j \\
\]

Press <F1> to change previous entry

Figure 2-6. Factored-Form Input Screen
For the dc motor example, the transfer function lends itself more easily to coefficient form input, and Figure 2-7 shows what the screen should look like after the entry is made. Notice that on both screens, the bottom line directs the user to press the F1 function key to move back to the previous entry. This facility can greatly enhance the capability to edit previous entries which may have been input incorrectly. Another error correction facility in the LCS-CAD program is the Change option executed from the Input/Change Menu (see Figure 2-4). This option will allow the user to select which block to change, then brings back the block input screens in the sequence discussed above except that the previously entered numbers are shown and a second message-prompt appears at the bottom of the screen.
This message directs the user to press the F10 function key to edit any item on the page. In addition to changing only the numbers previously entered, the user can also change the structure of the block by changing the order of the numerator and denominator.

Now that the system transfer function has been entered into the program, the analysis can begin. First, a quick check of the system's stability can be made. From the Main Menu, the selection

< L> Location of Char. Eq. Roots

will provide a listing of the unity-feedback closed loop characteristic equation roots. If all these roots are in the left-half of the s-plane, i.e., if all roots have negative real parts, then the system will be stable. As can be seen from Figure 2-8, the dc motor and load are stable.

*** Block Transfer Function Closed-Loop Roots ***

| ZEROS: |
| POLES: |

Press any key to continue or [Shift] [PrtSc] for hardcopy

Figure 2-8. Root Locations
Frequency domain analysis can be accomplished with LCS-CAD by selecting the

<F> Frequency Analysis

choice from the Main Menu. This selection will enable the user to prepare either a Nyquist (polar) diagram, or Bode (logarithmic) plot with user-selected radian frequencies. Figure 2-9 shows the on-screen selections for the Nyquist plot, and Figure 2-10 is the Bode selection screen. Both ask for the range of frequencies to be plotted. The first frequency should be an even power of ten. The range of frequencies is calculated based on the number of decades of frequency the user requests. That is, if the user selects 0.01 as the starting frequency with 4 decades, the end frequency will be 100 radians/sec.

***Bode/Nyquist Plotting Routine***

<table>
<thead>
<tr>
<th>Bode (B) or Nyquist (N) Plot?</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td>See the 3IG (B) picture, or select your Window (W)?</td>
<td>W</td>
</tr>
<tr>
<td>Open (O) or Closed (C) Loop Plot?</td>
<td>O</td>
</tr>
</tbody>
</table>

What is the first frequency to be plotted?
(e.g. .01, .001, 1000, etc.)

.01

How many decades do you want plotted?

5

Figure 2-9. Nyquist Plot Parameter Selection
Table 2-10. Bode Plot Parameter Selection

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Selection</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bode (B) or Nyquist (N) Plot?</td>
<td>B</td>
</tr>
<tr>
<td>Open (O) or Closed (C) Loop Plot?</td>
<td>O</td>
</tr>
<tr>
<td>What is the first frequency to be plotted? (e.g. .01, .001, 1000, etc.)</td>
<td>.1</td>
</tr>
<tr>
<td>How many decades do you want plotted?</td>
<td>3_</td>
</tr>
</tbody>
</table>

The "BIG picture" choice for the Nyquist simply causes the program to automatically select a broad range of frequencies and use a large scale plot. This can be a good initial selection if the user does not know how large a window will be needed to show the plot. The user may also choose an open or closed loop plot. If closed loop is chosen, a negative unity-feedback path is added around the overall block diagram, an equivalent transfer function is calculated, and the corresponding diagram is drawn.

For the dc motor example, both the Nyquist plot and the Bode diagram are shown in Figures 2-11 and 2-12 respectively.
Figure 2-11. DC Motor Nyquist Plot

Figure 2-12. DC Motor Open-Loop Bode Diagram
From the Bode diagram it can be seen that the phase margin is low, only about ten degrees. The gain crossover frequency is near thirty radians per second. This means that the uncompensated system will be stable, but probably will have a long settling time.

Next, the root locus can be examined. With two real roots, it may be expected that with increasing gain, the roots will converge along the real axis, then separate and move in opposite directions parallel to the imaginary axis. To verify this, the program is used to generate the root locus graph.

From the Main Menu the selection now will be

<R> Root Locus

and will produce the screen shown in Figure 2-13.

*** ROOT LOCUS PLOTTING ROUTINE ***

What STARTING value for variable gain do you wish?: .001
What ENDING value for variable gain do you wish?: 1

*** VIEWING COORDINATES FOR ROOT LOCUS GRAPH ***
X-Minimum: -10
X-Maximum: 1
Y-Minimum: -20
Y-Maximum: 20
Positive or Negative Feedback? (P or N): N
Any changes to these parameters? (Y or N): N
Press <F1> to change previous entry

Figure 2-13. Root Locus Parameter Input Screen
The program requests starting and ending values for the variable gain. This gain is multiplied by the block gain for the system, that is, for the dc motor if variable gain of 0.001 to 1 is chosen, then the actual gain the system will be varied through will be $100 \times 0.001$ to $100 \times 1$, or 0.1 to 100. The program computes the unity-feedback closed loop transfer function for an increment of gain then calculates and plots the location of all the roots. This process continues through the user-supplied range of gains.

The user is also asked to provide maximum and minimum values for X and Y plot axes. Knowing that the closed loop roots (with unity variable gain) are at $-5.000 \pm j 31.225$, the X axis values of -10 to 1 can be expected to provide an adequate window. Y axis values of ± 20 may be adequate to show the root behavior. Once the required values are entered, the program will begin computation and plot a graph like the one shown in Figure 2-14.

Finally in the analysis, it is desirable to see the system's response in the time-domain to a typical input. For a dc motor, such an input may well be a step. This so-called step response can be calculated and plotted with the LCS-CAD program as can the system response to a sinusoid, ramp, or impulse input. All these inputs have user-selectable amplitudes and if the ramp input is selected, the user may select slope and dc-offset. In the case of the sinusoid, the radian frequency may be selected.
Figure 2-15 shows the time response parameter input screen and Figure 2-16 shows the unit step response curve for the dc motor. It can be seen from the time response that the motor's response overshoots excessively and has a relatively long settling time (as anticipated from the Bode diagram), although the system is stable and has no steady-state error.

![Figure 2-14. DC Motor Root Locus Diagram](image)

Figure 2-15. Time Response Parameter Input Page

*** Time Response Plotting Routine ***

What is the input to your system?  
STEP (S)  
RAMP (R)  
SIN WAVE (W)  
IMPULSE (I)  

Input amplitude? 1

Open (O) or Closed (C) Loop simulation? C

How many seconds of simulation would you like to see? 2

Figure 2-15. Time Response Parameter Input Page
C. APPLYING THE SPECIFICATIONS

To use the motor in a larger system, the designer must either determine or be given the performance specifications for the motor. These specifications will usually outline the minimum acceptable performance the designer can tolerate from the system component.

For the example motor, suppose that the motor-controller combination must have a settling time of no more than 0.4 seconds and a first-overshoot of less than fifty-percent of the input signal when subjected to a step input. These constraints will allow demonstration of LCS-CAD in designing a simple cascade (series) compensator.
To assist the designer, curves for general second-order system response are available in virtually any basic control theory textbook. From these curves we can determine that for a fifty-percent overshoot, the required zeta(z), or damping factor, must be greater than 0.23. It then follows that a required phase margin $\Phi$ must be

$$\Phi = \tan^{-1}\left(\frac{2z}{\sqrt{-2z^2 + \sqrt{4z^4 + 1}}}ight)$$

$$\Phi \geq 25.9^{\circ}.$$ 

Also, to meet the other specification of a settling time of 0.4 seconds, we need the relationship between settling time (ts) and natural frequency (wn). This connection is

$$ts \leq 0.4 = \frac{4}{zwn},$$

or $$wn \geq 43.5 \text{ rad/sec} \quad (\text{since } z \geq 0.23)$$

Since wn is difficult to plot on the Bode graph, a more convenient frequency is needed. This is bandwidth frequency, or -3dB frequency, wb.

$$wb \geq wn \sqrt{1 + 2z^2 + \sqrt{2 - 4z^2 + 4z^4}}$$

or $$wb \geq 65.1 \text{ rad/sec}.$$ 

Armed with the appropriate wb, the compensator design can be attempted. On a large printout of the uncompensated Bode diagram as in Figure 2-17, which can be obtained using the "print screen" option (press the keys <Shift> and <PtrSc> (Print Screen) simultaneously), locate the wb point...
System Compensation by Approximating Asymptotes on Bode Diagram.

\[ G(s) = 100 \times \frac{1}{s(0.1s + 1)} \]

Figure 2-17. Working Bode Diagram with Asymptotes
and draw a -20 dB per decade asymptote through it. This will mark the crossover frequency and should provide good closed loop response. For simplicity, the compensator should only have one pole and one zero. This can be accomplished if the zero is located at the intersection point of the asymptote and the uncompensated system curve. As seen on the Bode diagram of Figure 2-17, the zero can be placed at \( w = 20 \) rad/sec. The pole can be placed at \( w = 100 \) rad/sec to give the -20 dB asymptote a reasonable length to ensure a large enough phase margin (\( \Phi_m \)) and consequently, a good closed loop response.

Having determined pole and zero locations for the compensator, we can input the design into the LCS-CAD program and verify the solution quickly. From the Input Menu the option to “Add a Block” can be selected and the compensator input as block number two. In order not to change the motor gain, and therefore the steady state error, the compensator itself should have an offsetting gain of 100/20 or 5.0. Once entered, system analysis can begin as before.

The open loop Bode diagram shown in Figure 2-18 shows that indeed the crossover frequency was increased to approximately 45 rad/sec and the phase margin to near 50 degrees. According to the earlier calculations, these parameters should ensure that the system is well within specifications. The root locus plot of Figure 2-19 also
helps to confirm this. As a final, conclusive check, however, the system response to a unit step, shown in Figure 2-20, can be seen to have a settling time of less than 0.2 seconds and a very small first overshoot. The design is, therefore, probably satisfactory. Since the compensated system performance is considerably better than the original specifications, it may be more costly to build than one that has slightly worse, but still satisfactory, performance. If this is the case, or if the first pole and zero placement not produced a satisfactory system, additional design trials could have been run quickly and easily with LCS-CAD.

![Figure 2-18. Compensated Open-Loop Bode Plot](image-url)
Figure 2-19. Compensated System Root Locus

Figure 2-20. Compensated System Time Response
D. SUMMARY

In this chapter, a review of system design and an overview of LCS-CAD was provided. Several noteworthy program features are listed below.

* The Input/Change Routines enable the user to conveniently input and change transfer function block descriptions.

* The automatic block manipulator synthesizes the user's input blocks into an equivalent loop transfer function.

* The user has the ability to quickly and easily generate Bode, Nyquist, Root Locus, and Time Response plots.

Additional program features are available which were not discussed here, most importantly, the "two parameter root locus". This procedure as well as the other program features will be discussed in detail in the following chapter.
III. DETAILED PROCEDURE MODULE DESCRIPTIONS

A. PROGRAM OVERVIEW

LCS-CAD has a hierarchical, menu-based structure that allows the user general freedom to choose which design tool he wishes to use. The "Main Menu" is the starting point of the program and all other menus and utilities are available from here. In fact, the body of the main program, "CAD.PAS", simply calls the procedure "MainMenu" repeatedly until the user indicates that he is finished by typing <Q>.

The main menu procedure simply displays the menu shown in Figure 2-3, and provides for branching to other subroutines as requested by the user. A hierarchical representation of the Main Menu structure is shown in Figure 3-1 below.

![Figure 3-1. Main Menu - Functional Block Diagram.](image-url)
As seen in Figure 3-1, the routine MainMenu can invoke two other menus, the Input Menu and the Utilities Menu. These will be discussed in detail later in this chapter. Also reachable directly from the Main Menu are the major analysis tools provided by the program, namely, frequency analysis, both single and two parameter root locus, and time response procedures. The frequency analysis portion of the program offers both Bode and Nyquist plots. The former represents system response as curves of phase (degrees) and magnitude (dB) versus radian frequency (logarithmic scale), and the latter displaying this information as a polar plot of imaginary and real parts of the magnitude.

The root locus programs include the classical "gain locus" where the system gain is varied over a user-selectable range. The two-parameter root locus allows the user to input the coefficients of a system's characteristic equation with two unknown parameters. The program then increments each of the parameters through ranges specified by the user and plots the resulting family of root locus curves.

The time response routine allows the user to select from a number of typical system inputs including step, ramp, impulse, and sinusoid as inputs to the current loop. The program then calculates the system's response to the input and plots it.
B. INPUT/CHANGE MENU ROUTINES

1. Input Menu Hierarchy

The input and change routines are a collection of procedures which allow the user to input and change the block transfer functions of his system. There are thirteen functional Pascal routines and numerous utility procedures which implement the input, change, and block manipulation functions. Figure 3-2 shows the hierarchy of these functional procedures within the larger input routine.

![Input Routine Functional Block Diagram](image)

Figure 3-2. Input Routine Functional Block Diagram.
2. **Input Utility Functions**

One of the more difficult tasks in making a program "user-friendly" is validation of the user's input. This includes, but is not limited to, checking for the correct type of input (e.g., numbers, letters, or symbols), range checking numeric input, and checking to ensure that the answer makes "sense" (i.e., a "K" input does not satisfy a "Y" or "N" question). If the validation fails, an intuitive and graceful procedure must alert the user that a mistake has been made and re-prompt for a proper response. Some of these tasks have been accomplished in LCS-CAD through a group of very useful input routines available in the public domain. The routines used include a menu-generation procedure ("MainMenu"), procedures to write and center messages on the screen ("Msg" and "Center"), and two very powerful input routines called "Input" and "Input_Handler".

The procedure "Input" is a simple procedure which can be called by the programmer, but serves as the base routine for the more sophisticated routine "Input_Handler". Alone, Input will prompt for a single user-supplied entry, check it for type and length, and even provide a default answer if so programmed. Additionally, the routine provides the user with rudimentary text editing capabilities such as backspace, insert, and delete during
keyboard entry. The syntax for a call to Input is given below:

Input(type, default, col, row, length, uppercase, F1, F10);

where  type = 'A' for alphanumeric,
    'N' for numeric,
    'F' for formatted (not used in LCS-CAD)
default = text string to display default value
col,row = column and row on screen for prompt
length = number of character spaces in prompt field
uppercase = boolean. True to convert input to all u.c.
   F1,F10 = boolean. True if function keys F1 or F10
         pressed. Used only for Input_Handler
         calls.

Type checking is automatically performed by the
procedure and if the user attempts to input an alphabetic
character in a numeric (N) field, for example, a "beep"
will be generated to alert the user and the character will
not be accepted. Once a valid input is supplied to the
procedure and the <Return> key is pressed (signalling the
end of user input), the routine returns the user's input in
the global variable "Answer". Answer is always returned as
a string-type variable, so if a number is expected, the
programmer must provide for conversion to a numeric type
via the built-in Pascal function "val".

The procedure "Input_Handler" is a sophisticated,
full-page, input editor. The programmer is required to
predetermine each page layout and generate the necessary
textual prompts. A set of array elements describing the
desired input format is then generated in the form shown
below:
P[1] := '2505A02501T010102'

P[n] where n is limited to 40 elements.

The contents of each P[] element is a coded, column-dependent, descriptor field explained below:

<table>
<thead>
<tr>
<th>Column Numbers</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-2</td>
<td>column for input field to start</td>
</tr>
<tr>
<td>3-4</td>
<td>row for input field</td>
</tr>
<tr>
<td>5</td>
<td>input type - A: alphanumerics, N: numeric, F: formatted (not used), $: dollar (not used)</td>
</tr>
<tr>
<td>6-8</td>
<td>length of the input field</td>
</tr>
<tr>
<td>9-10</td>
<td>element of the output global array Filvar[] to store the user's input</td>
</tr>
<tr>
<td>11</td>
<td>set to T if Caps Lock is to be set</td>
</tr>
<tr>
<td>12-13</td>
<td>default item number</td>
</tr>
<tr>
<td>14-15</td>
<td>prompt item number</td>
</tr>
<tr>
<td>16-17</td>
<td>validation number</td>
</tr>
</tbody>
</table>

Default, prompt, and validation numbers call the procedures "Get_Default", "Say_Prompt", and "Do_Validation" respectively. The procedures are primarily comprised of Pascal "case" statements that use the associated item numbers to perform some programmer defined default, prompt, or validation written into the case-statement. These procedures will be described more fully when discussing the LCS-CAD routines which use them.
Input_Handler can now be called with the syntax:

```c
Input_Handler( '5-character string' );
```

<table>
<thead>
<tr>
<th>String Column</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 . . . . N</td>
<td>new entries</td>
</tr>
<tr>
<td>C</td>
<td>changes to old entries</td>
</tr>
<tr>
<td>D</td>
<td>re-display of entry (not used)</td>
</tr>
<tr>
<td>2-3 . . .</td>
<td>first element of P[] array to use</td>
</tr>
<tr>
<td>4-5 . . .</td>
<td>last element of P[] array to use</td>
</tr>
</tbody>
</table>

If calling the procedure with "N" for new entries, the contents of the corresponding output array element, "Filvar[]", is cleared and readied for new input. If "C" is used, the old value of the associated Filvar variable is displayed as the default input for the field. This is essential for full-screen editing of inputs. Input_Handler calls the Input routine as described above, but uses the boolean variables Fl and F10 for access to all fields on a screen. In the "N" - new entries mode, a user prompt is displayed in the lower left corner of the screen instructing the user to press Fl to edit the previous entry on the screen. If in the "C" - change mode, an additional prompt tells the user to press F10 to discontinue editing on that screen. By using the Fl, <Return>, and F10 keys, the user can quickly and effectively edit previously entered screens of data to either change a problem or correct erroneous entries.
Input_Handler returns the user's input to the calling program in the global variable array "Filvar". As with the Input routine's Answer variable, all the Filvar elements are string-type and must be converted if numeric input is needed.

These utility routines were helpful in developing LCS-CAD and are used extensively throughout the program for user input. Several other subroutines are available in the package either for programmer use or to be called by the routines described here. Appendix A is a descriptive excerpt from the user's guide supplied with these subroutines.

3. Blocks Record

LCS-CAD uses the transfer function block as the basic foundation element of the program. Not only does this have the advantage of being intuitive to the designer accustomed to classical block manipulation design, but facilitates algorithm design which is also simple for the engineer to follow. Pascal handles this block-by-block design particularly well with its "record" structure. Each transfer function block in the user's current loop is described by one corresponding Pascal record. Each record is a logical grouping of all the parameters necessary to describe the block. This record structure is shown below in Figure 3-3.
The record contains the order of the numerator and denominator polynomials in the variables NZeros and NPoles respectively. The coefficients of the polynomials are contained in the arrays (of type PolyArray) NumCoeff and DenCoeff. The factors of the numerator polynomial are held in the arrays RealPartZero and ImagPartZero for the real and imaginary portions of the complex conjugate factors respectively. Likewise, the denominator factors are stored in RealPartPole and ImagPartPole.

"LeadNumCoeff" and "LeadDenCoeff" are used to "normalize" and "un-normalize" the leading coefficients of the polynomials. That is, for calculations, the routines require polynomials whose leading coefficient is one, but for display, the polynomials must be converted back to the original input form. These two real variables facilitate those conversions.
"FeedBack" is a boolean variable that is true when the block is in the feedback path and false when it is in the forward path of the loop. "Factored" is true if the transfer function was initially input in factored form and false if the input was in coefficient form.

If the user inputs a block transfer function into the program in coefficient form, the routine "RootFinder" is called to generate the factors of the polynomials and store them in the appropriate record variables. In the case of factored input, the routine "Expand_Poly" provides the coefficients of the expanded polynomials and stores them in the record variables "NumCoeff" and "DenCoeff". RootFinder and Expand_Poly will be discussed later.

Once all the blocks in a loop have been input, the procedure "Make_Geq" is invoked which reduces the transfer functions of all the blocks in the loop into a single "equivalent" block with an identical record structure as all the other blocks. Most plotting functions and analysis tools use this equivalent block record for all their computations.

4. RootFinder Module

One of the most important routines in LCS-CAD is the procedure called "RootFinder". This sub-program's function is to find the factors, or roots, of any given polynomial. The procedure is used by the Input routine to find the poles and zeros of a transfer function input as a
quotient of two polynomials. It is also called by the root-locus and two-parameter root locus programs to find successive roots of the transfer function while changing the system gain or other parameters.

Procedure RootFinder uses Bairstow's method to find the roots of a polynomial with real coefficients numerically [Ref. 1]. This algorithm iteratively searches for a quadratic factor of a given polynomial and, when it finds the factor, deflates the original polynomial and repeats the process. The algorithm is outlined in Figure 3-4. Procedure RootFinder is called and passed the order of the polynomial (N), the polynomial coefficients (Coeff) in array form, the initial guesses for P and Q (P1 and Q1), and returns two arrays of real and imaginary roots (RealPartRoot and ImagPartRoot). RootFinder first normalizes the input polynomial to have a leading coefficient of one and loads the polynomial into an array A. The B and C arrays are initialized to contain all zeros.

The procedure checks the input polynomial order and handles the simple cases of a zero, first, or second order polynomial. If higher order, the procedure invokes the Bairstow algorithm and computes a quadratic factor. When a factor is found RootFinder calls a supplementary procedure named Solve_Quadratic to determine the complex conjugate roots using the classical quadratic equation. With the "B"
Given an \( n \)th degree polynomial:

\[
ax^n + a_4x^{n-1} + \ldots + an+2x + an+3
\]

and the initial coefficients \( P \) and \( Q \) of the quadratic factor:

\[
x^2 - Px - Q
\]

then

Set \( B(1), B(2), C(1), \) and \( C(2) = 0. \)

DO WHILE DeltaP > epsilon (tolerance value), or DeltaQ > epsilon,

DO FOR \( J = 3 \) to \( n + 3 \) step 1,

Set \( B(J) = A(J) + P \times B(J-1) + Q \times B(J-2). \)

Set \( C(J) = B(J) + P \times C(J-1) + Q \times C(J-2). \)

Set \( \text{DENOM} = C(N+1)^2 - C(N+2) \times C(N). \)

IF \( \text{DENOM} = 0 \) THEN

Set \( P = P + 1. \)

Set \( Q = Q + 1. \)

Repeat from beginning.

ENDIF.

Set DeltaP = \[
\begin{bmatrix}
-\text{B}(N+2) \times \text{C}(N+1) + \text{B}(N+3) \\
\times \text{C}(N)
\end{bmatrix}
/ \text{DENOM}.
\]

Set DeltaQ = \[
\begin{bmatrix}
-\text{C}(N+1) \times \text{B}(N+3) + \text{C}(N+2) \\
\times \text{B}(N+2)
\end{bmatrix}
/ \text{DENOM}.
\]

Set \( P = P + \text{DeltaP}. \)

Set \( Q = Q + \text{DeltaQ}. \)

ENDDO.

Figure 3-4. Bairstow's Algorithm

array now holding the deflated polynomial of order \( n-2, \)
the procedure checks again for the simple low-order cases
and solves the remainder of the problem or continues
searching for another quadratic factor. To look for
another quadratic factor, the \( B \) matrix coefficients are
loaded into the \( A \) matrix and the entire procedure is
repeated.

RootFinder operation is affected by two parameters
which are somewhat arbitrarily selected. These are
Epsilon, the acceptable error, and IterationCount, the number of times the procedure will repeat the search for a factor. If Epsilon is chosen to be very small, and IterationCount very large, the accuracy of the solution should improve, but the execution time of the procedure will be degraded. Currently, Epsilon is 0.00001 and IterationCount is 40. In program tests against known results these two constants have provided satisfactory results without noticeable execution time degradation.

5. Polynomial Expansion Routine

The procedure Expand_Poly takes as input a set of real and complex conjugate factors and expands them into a polynomial with real coefficients. There are numerous algorithms to accomplish this, but the most intuitive one was selected for use in LCS-CAD. The process here follows closely the steps one would take if performing the operation long-hand. The steps are outlined in Figure 3-5.

The process is best explained by example. Assume that the following set of factors is to be operated on by the subroutine Expand_Poly algorithm.

\[(s + 1 + j1.414)(s + 1 - j1.414)(s + 4 + j0)\]

The first two factors constitute a complex conjugate pair since their real and imaginary parts have the same magnitude with the imaginary parts differing in sign. When these two factors are multiplied together, the
result will be a real-coefficient, quadratic polynomial. For this particular example the product of the conjugate pair will be

$$(s^2 + 2s + 3).$$

The program will recognize that, since there is a non-zero imaginary part of the first factor, a complex conjugate pair is present in the problem and resolve this first by forming the quadratic as above. To accomplish this, TEMP[1] will be set to $1^2 + 1.414^2$ or 3.0, and TEMP[2] will be set to $(2 * 1)$ or 2.0, in accordance with the algorithm.

Assignments to POLY will be identical and the order of the system, originally three, will be reduced to one. This will leave only the $(s + 4)$ factor to deal with in the subsequent step. So the program will examine this next factor and determine it to be real. HOLD[1] will be set to three and HOLD[2] will be set to zero. This initial setup is shown in Figure 3-6.

Now the polynomial in POLY is shifted left one place to simulate multiplying it by the unity coefficient of the $s$-term in the $(s + 4)$ factor. Finally, the contents of TEMP are multiplied by HOLD[1] and added to the shifted POLY contents as seen in Figure 3-7.

The addition indicated in Figure 3-7 yields the polynomial with coefficients

$$1s^3 + 6s^2 + 11s + 12.$$
These coefficients are stored back into the array TEMP and the process continues if there are more factors.

Given n real or complex factors of a polynomial:

\[(s + a_1 \pm jb_1)(s + a_2 \pm jb_2)\ldots(s + a_n \pm jb_n)\]

where \(a\)’s are real and \(jb\)’s are imaginary or zero, then

Set Temp[i] and Poly[i] = 0. \{for all \(i\) to \(n\}\}

order = \(n\).

IF \(b = 0\) THEN \{real value\}

Set TEMP[1] = POLY[1] = \(a_1\).
Decrement order by 1.

ELSE

Set TEMP[1] = POLY[1] = \(a_1^2 + b_1^2\). \{conjugate multiply\}
Set TEMP[2] = POLY[2] = 2 \* \(a_1\).
Decrement order by 2.

ENDIF.

WHILE order > 0 DO

IF \(b = 0\) THEN

Set HOLD[1] = \(a_1\).
Set HOLD[2] = 0.
Set POLY = ShiftLeft(POLY).
Set POLY[1] = 0.
Set POLY = POLY + (TEMP \* HOLD[1]).
Set TEMP = POLY.
Decrement order by 1.

ELSE

Set HOLD[1] = \(a_1^2 + b_1^2\). \{conjugate multiply\}
Set HOLD[2] = 2 \* \(a_1\).
Set POLY = ShiftLeft(POLY).
Set POLY[1] = 0.
Set POLY = POLY + (TEMP \* HOLD[2]).
Set POLY = ShiftLeft(POLY).
Set POLY[1] = 0.
Set POLY = POLY + (TEMP \* HOLD[1]).
Set TEMP = POLY.
Decrement order by 2.

ENDIF.
ENDDO.

Figure 3-5. Expand_Poly Algorithm
In the actual program implementation, the conjugate multiplication is developed as a separate procedure called "Conjug_Mult" for program readability. Also, the coefficients in the final result stored in the POLY array are reversed to conform to the established storage order protocol.
6. Building the Equivalent Loop Block

An essential element of the LCS-CAD program is the "Make_Geq" procedure. This routine calculates the equivalent block transfer function from all the blocks in the user input loop. Conceptually, the process is a simple one, and because of the block record structure, the algorithm, too, is relatively straightforward.

For a given loop each of the transfer function blocks will be in either the forward or feedback path. The first step will be to condense these blocks into single forward and feedback blocks. To combine blocks in the same path the transfer functions are multiplied, or more simply, the factors of the transfer functions are just lumped together as shown below in Figure 3-8.

\[
\frac{(s + a)(s + b)}{(s + d)(s + e)(s + f)} \times \frac{(s + c)}{(s + g)(s + h)}
\]

Figure 3-8a. Two Path T.F. Blocks

\[
\frac{(s + a)(s + b)(s + c)}{(s + d)(s + e)(s + f)(s + g)(s + h)}
\]

Figure 3-8b. Equivalent Block Diagram

Figure 3-8. Transformation of Two Path Transfer Functions to a Single Equivalent Transfer Function Block
Once the forward and feedback path blocks have been reduced to two equivalent blocks, they can easily be reduced to one single block by application of Mason's rule. This is demonstrated in Figure 3-9.

The Make_Geq procedure closely follows these steps to reduce the loop block to a single "G-equivalent" block. The simplified algorithm is shown below in Figure 3-10. In the actual implementation, additional arrays and temporary variables were required to hold the real and imaginary parts of each pole and zero, and the forward and feedback block gains.

If N's are numerator and D's are denominator polynomials, then by Mason's Rule the equivalent transfer function is:

$$G_{eq}(s) = \frac{G_1(s)}{1 + G_1(s)G_2(s)}$$

{Multiplying through by $D_1D_2$ gives the right-hand block expression.}

Figure 3-9. Reduction of a Forward and Feedback Block to a Single Equivalent Block
Given NBlocks transfer functions
then

Initialize Fwd_Zero_Count, Fwd_Pole_Count,
  Feedback_Zero_Count, Feedback_Pole_Count = 0.

FOR i = 1 TO NBlocks STEP 1 do
  IF FeedbackBlock THEN
    COLLECT(Feedback_Zeros and _Poles) in Feedback_Array.
    INCREMENT(Feedback_Zero_Count & _Pole_Count).
  ELSE
    COLLECT(Forward_Zeros and _Poles) in Forward_Array.
    INCREMENT(Forward_Zero_Count & _Pole_Count).
  ENDIF
ENDFOR.

WITH G_eq DO
  {construct Geq Zeros}
  COLLECT(Forward_Array_Zeros & Feedback_Array_Zeros) in
    Geq_Zeros.

  {construct Geq Poles}
  COLLECT(Forward_Array_Zeros & Feedback_Array_Zeros) in
    Temp1.
  EXPAND(Temp1) {to Polynomial and store} in TempPoly1.
  COLLECT(Forward_Array_Poles & Feedback_Array_Poles) in
    Temp2.
  EXPAND(Temp2) {to Polynomial and store} in TempPoly2.
  Set Geq_Denominator_Coefficients to TempPoly1+TempPoly2
  FACTOR(Geq_Denominator_Coefficients) in Geq_Poles.
ENDWITH.

Figure 3-10. Make_Geq Procedure Algorithm


The basic input program relies on four separate
procedures, namely, Trans_FUNCTION_Input, Block_Input,
Input_Factored, and Input_Coeff. Trans_FUNCTION_Input is
the procedure invoked from the Input Menu. It serves two
functions:
(1) Query the user for number of blocks in the loop, then execute repeat a call to Block_Input until all the blocks are entered,

(2) Call the Make_Geq procedure after all blocks are input to form the equivalent transfer function.

The Block_Input procedure is the heart of the input routine. It queries the user for block attributes like number of poles and zeros, block gain, and whether the user will input the block transfer functions as factors or polynomial coefficients. Block_Input relies heavily on the services provided by the procedures Input and Input_Handler described earlier. These utility procedures give the user a great deal of flexibility with their built-in error checking and validation and the full-screen editing capabilities.

Based on the user's choice of factored or coefficient form input, Block_Input calls Input_Factored or Input_Coeff respectively. These routines display user prompts for their particular kind of input and also use the Input_Handler routine extensively.

An unusual segment of the program implementation worthy of note involves the coefficient form of input. If the leading coefficient input by the user is not unity, then the program will normalize the polynomial. This involves dividing the polynomial through by the leading coefficient, then correcting the block gain to compensate for the change. Several algorithms in LCS-CAD, in particular the Root_Finder procedure, expect a unity...
leading coefficient on a block polynomial. The normalization process does produce unwanted side-effects, however. If the user decides to view or change a transfer function that was entered in coefficient form, then he could be confused by the changes to block gain and the normalized polynomial. To alleviate these problems, two variables for each block, LeadNumCoeff and LeadDenCoeff, hold the original values of the user input leading coefficients of the numerator and denominator polynomials respectively. If the user chooses to change the block, the normalization process can be reversed and the coefficients and gain restored to their input values.


One of the most important features of the LCS-CAD user interface is its ability to allow changes to the block descriptions quickly and easily. The major tool which supports this facility is the Input_Handler procedure discussed earlier. The LCS-CAD input and change procedures expand Input_Handler's full screen editing support to include the capability to change the actual structure of a previously entered block. That is, the user can change the order of the block transfer function as well as the coefficients or factors that were previously entered. This very powerful combination facilitates both input error correction and changes made during the design process.
To make these changes possible, the procedure Change_Block sets a boolean variable called "Change" to true when the procedure is entered. This alerts the Block_Input procedure, which is also used for initial user input, that changes are to be made to the block whose index is passed as a parameter. The Block_Input routine then reinitializes the appropriate Filvar[] array elements to the previously entered values and uses them as default values to each query (see Section 2. Input Utility Functions for Filvar explanation). The process then continues exactly as for first-time input.

9. Add and Delete Blocks

The procedure Add_Block allows the user to add a block to the current loop. The routine simply increments the block counter variable, NBlocks, and calls the block input procedure, Block_Input. On return from Block_Input, a new G-equivalent is calculated.

Delete_Block is a slightly more complicated procedure. It must first ask the user to identify the block number of the block to delete, check that the block exists, and then remove it from the current loop. The block is effectively removed from the loop by deleting its block index number and adjusting the remaining block indices. A new G-equivalent block is computed after a block is deleted.
10. Save Blocks to Disk

The procedure used to store a problem to a floppy or hard disk file is `Save_Block`. It queries the user for drive designator (A through D), and filename for the loop blocks to be saved. Only the first eight characters of an MS-DOS filename are allowed; the program appends a filename extension of ".BLX" to each loop. This extension is used to later to limit the disk search for legitimate block files when retrieving the data. Once a drive and filename are supplied the procedure opens the file and stores the loop data. The data file is a sequential file of records stored as illustrated in Figure 3-11.

Note from Figure 3-11 that the equivalent block, Geq, is stored first at the zero file-pointer position. This is convenient since, regardless of the number of blocks in the loop, the equivalent block description is always at position zero and easy to access. There are two opportunities to read this file during the program, the
first available from the single block input routine when the program asks whether the block will be read from a file or from the keyboard. If the file option is taken, then only the Geq block is accessed. The other time the file can be read is from the "Retrieve Problem from Disk File" option on the Input Menu. This option will read all the blocks into memory, including the Geq block. This option is discussed in detail in the next section.

11. Retrieve Problem from Disk File

Like the Save_Block procedure, the Retrieve_Problem procedure first queries the user for the data drive where the block descriptions are located. Once this is done, the program calls another public domain procedure called "Directory". This routine uses MS-DOS function calls to query the disk drive and read the directory listing. It then displays the directory in a window on screen with a moveable cursor activated with the arrow keys on the numeric keypad. The cursor can be moved to point to the desired file and, when the <Return> key is pressed, the filename is returned to the Retrieve_Problem procedure. Retrieve_Problem then opens the file and reads the contents.

The Directory procedure will only request the disk filenames with the MS-DOS extension ".BLX". This eliminates the possibility that the user can attempt to
read in a block that is not a data block (recall that the Save_Block procedure automatically appended this extension onto the user-supplied filename).

12. Using Input Routines to Build Complex Systems

The tools described thus far should be sufficient to build arbitrarily complex systems if the proper approach is used. The powerful block manipulator is capable of reducing a loop with up to nine blocks. Block size is limited to a ninth order numerator or denominator, due to screen display limits, and overall system size is limited to a polynomial of thirtieth order, based on the compiler-imposed memory size limit. The greatest limitation of the LCS-CAD program for the designer is the ability to only handle one loop in memory at one time. This can be overcome by working a complex problem "from the inside-out".

Given a complex problem like the one shown in Figure 1-11, the problem can best be solved by the following steps:

(1) Enter block B and C as a system and save to disk;

(2) Enter block D from the keyboard and the BC equivalent from the disk file. Save result to disk;

(3) Enter blocks A, E, and F from keyboard and the BCD equivalent from diskfile. Analyze the system.

If this were a typical feedback design problem, the engineer would need to analyze the interior loops for stability before proceeding to the outer loops as a
standard practice, so the inside-to-out limitation would be minimized. At any rate, this limitation is at worst an annoyance and should have no serious effect on overall design performance.

Figure 3-11a. Complex Block Diagram

Figure 3-11b. Reduced BC Loop

Figure 3-11b. Reduced BCD Loop

Figure 3-11c. Equivalent Block.

Figure 3-11. Reduction of a Complex Block Structure with LCS-CAD
C. LOCATION OF CHARACTERISTIC EQUATION ROOTS

As discussed in Chapter 2, one way to quickly check the stability of a system is to examine the closed-loop roots of the system's characteristic equation. If all the roots are in the left-half of the S-plane, i.e. real parts of the roots are negative, then the system is stable.²

The extreme usefulness of such a utility was the impetus for placing it in the Main Menu. Once the user has input his block diagram and returned to the main menu, only a keystroke is required to check the loop for stability. If the loop is not stable, then the user knows immediately that compensation is required and can begin the design process.

The procedure which computes and displays the closed-loop roots is called "ShowRoots". Negative unity feedback is used to close the loop on the system. The algorithm is shown in Figure 3-12. Recall that in a unity feedback system, the closed-loop denominator polynomial is equal to the sum of the open-loop denominator coefficients and the block gain times the numerator coefficients, since

\[ G_c(s) = \frac{G_o(s)}{1 + G_o(s)} \]

The closed-loop zeros are identical to the open-loop zeros.

² It can be established for a stable, causal system that the roots will all be in the left-half of the S-plane. Since, from an engineering standpoint, all physically realizable systems will be causal, therefore this generalization can be made.
Given the G-equivalent block for the system:

CALCULATE the on-screen display position.
DISPLAY Geq.NumCoeff ROOTS. {C.L. zeros = O.L. Zeros}

SET ClosedLoopPoly = (Geq.NumCoeff * K) + Geq.DenCoeff.
CALL RootFinder.
DISPLAY ROOTS of ClosedLoopPoly.

Figure 3-12. ShowRoots Algorithm.

D. FREQUENCY ANALYSIS

Frequency-domain analysis is essentially the examination of a system's response to input sinusoids of varying frequency. Not only can this so-called "frequency response" be computed analytically from the system transfer function, but the model of a complex physical system can be obtained experimentally by injecting input sinusoids and observing the output response. For these reasons, frequency domain design techniques have been developed and used for many years.

Among the most popular design tools in the frequency-domain are graphical techniques known as Bode and Nyquist plots. Historically, the graphical methods allowed the engineer to visualize the system's behavior. He could then apply various heuristic methods and approximations to obtain a qualitative "best guess" at how the system would respond to a given input. The other alternative of deriving the quantitative response was often mathematically complex and time-consuming if it was possible at all.
Now with the advent of digital computers, and especially the desktop "personal computer", the engineer has a device that can provide the quantitative information while still providing the familiar graphical tools that he is accustomed to designing with. The end result should be faster, easier, and more accurate designs.

1. **Nyquist Plot**

The Nyquist plot is a graph of the magnitude of $G(j\omega)$ versus the phase angle of $G(j\omega)$ as $\omega$ is varied through a range of frequencies. The graph is plotted in polar coordinates. For each frequency, the phase and magnitude is calculated and plotted on the Nyquist plot. Figure 3-13 shows a graphical interpretation of magnitude and phase.

![Nyquist plot](image)

Figure 3-13. System Magnitude and Phase
For computation purposes the magnitude of a single pole or zero can be written as

\[
\text{Magn} = \sqrt{(\text{RealPart})^2 + (w - \text{ImagPart})^2}
\]

and its phase is

\[
\text{Phase} = \tan^{-1}\left[\frac{w - \text{ImagPart}}{\text{RealPart}}\right].
\]

For multiple poles or zeros, the magnitude and phase are simply computed for each individual pole or zero, then the magnitudes multiplied and the phases and added together.

The overall system magnitude can be computed as

\[
\text{Magn}_{\text{system}} = \frac{\text{Magn}_{\text{zeros}}}{\text{Magn}_{\text{poles}}}
\]

and the overall system phase is

\[
\text{Phase}_{\text{system}} = \text{Phase}_{\text{zeros}} - \text{Phase}_{\text{poles}}.
\]

The LCS-CAD procedure Bode performs precisely these calculations to compute both the Bode and Nyquist system response. The algorithm is outlined in Figure 3-14. Once the system phase and magnitude are computed, the simple transformation into cartesian coordinates is required to plot the information.

The user is given several options to select from regarding the plotting and calculation parameters for the Nyquist plot. The beginning frequency can be selected as can the number of decades to be plotted. The user can choose to manually select a window size (maximum and minimum values on the plotting scale) or can alternately choose to see the “big picture”. This option will plot a
Given the poles and zeros of a transfer function:

\[ G(s) = \frac{(s + a_1 \pm j b_1)(s + a_2 \pm j b_2) \ldots (s + a_m \pm j b_m)}{(s + c_1 \pm j d_1)(s + c_2 \pm j d_2) \ldots (s + c_m \pm j d_m)} \]

and a range of frequencies \( w_1 \ldots w_k \), then

**Figure 3-14. Nyquist Algorithm**

\[ \text{WHILE} \ w < w_k \ \text{DO} \]

\[ \text{SET} \ Z\text{Magn} = PM\text{agn} = 1.0; \ \text{SET} \ Z\text{Phase} = P\text{Phase} = 0.0; \]

\[ \text{FOR} \ i = 1 \ \text{TO} \ m \ \text{STEP} \ 1 \ \text{DO} \ \{\text{compute for zeros}\} \]

\[ \text{SET} \ Z\text{Magn} = Z\text{Magn} \ast \sqrt{a_i^2 + (w - b_i)^2}. \]

\[ \text{SET} \ Z\text{Phase} = Z\text{Phase} + \text{ArcTan}(\frac{w - b_i}{-a_i}). \]

\[ \text{ENDFOR}. \]

\[ \text{FOR} \ i = 1 \ \text{TO} \ n \ \text{STEP} \ 1 \ \text{DO} \ \{\text{compute for poles}\} \]

\[ \text{SET} \ P\text{Magn} = P\text{Magn} \ast \sqrt{c_i^2 + (w - d_i)^2}. \]

\[ \text{SET} \ P\text{Phase} = P\text{Phase} + \text{ArcTan}(\frac{w - d_i}{-c_i}). \]

\[ \text{ENDFOR}. \]

\[ \text{SET} \ Magn\text{sys} = Z\text{Magn}/P\text{Magn}. \ \text{SET} \ Phases\text{ys} = Z\text{Phase} - P\text{Phase}. \]

\[ \text{SET} \ PlotX = Magn\text{sys} \ast \cos(\text{Phasesys}). \]

\[ \text{SET} \ PlotY = Magn\text{sys} \ast \sin(\text{Phasesys}). \]

\[ \text{INCREMENT} \ w. \]

\[ \text{ENDWHILE}. \]

range of frequencies from \( 10^{-3} \) radians/second to \( 10^5 \) radians/second and automatically plot the resultant Nyquist graph on a 50 x 50 scale. The user may also choose to plot open-loop or closed-loop response. If the open-loop response is desired, the G-equivalent block is used. If the closed-loop response is selected, then the procedure described in Section C. is used to compute the equivalent unity-feedback, closed-loop system transfer function prior to calculating the Nyquist response.

The program implementation consists of two procedures, "Bode" and "Plot_Nyquist". As described above,
the procedure Bode doubles as both the Bode and Nyquist calculation procedure. The routine Plot_Nyquist actually implements the graphics which plots the Nyquist numbers generated in the Bode subprogram.

Within the Plot_Nyquist routine is a call to the procedure "Graph_Menu". Graph_Menu is called by all procedures which produce a graph. It provides a pop-up menu offering the user the opportunity to add a title to the graph just plotted, print the graph on a printer, print the numbers used to generate the graph, or quit and return to the Main Menu.

If the user elects to title the graph, the plot is swapped onto a "virtual screen" in memory and a text screen appears offering the user three blank lines to type in his title. When the title is completed the plot is returned to the physical screen and a window is drawn with the title text inside. Since there is no way to tell a priori where on the screen the important part of the plot will be located, the title block can be relocated anywhere on the screen by using the cursor arrow keys. When the title box is where the user wants it, the <Return> key is pressed thus freezing its position and recalling the graph options menu to the screen. The option which allows the user to print the graph will first remove the menu options block from the graph and dump the remaining screen information to
the printer. If the user decides to instead print the numbers, the procedure is slightly more complicated.

The "dump numbers" mode saves the current graphics screen and offers the user a choice to print the data to the printer or a pre-selected data filename on disk. For most of the graphics routines, the number of points is too large to store in a matrix when they are computed to plot the graph. Therefore, when the option to print the numbers is selected, the same computations are repeated again, and the numbers are printed or stored to a disk file rather than used to plot the graph. The user should be cautioned that because of the large number of points calculated and plotted, the length of the printer listing can be excessive. If only a few data points are of interest, the print to a file option is the better choice. Then the user can scan that file with a word processor or the MS/DOS "type" command and examine the points of interest.

2. Bode Plot

Most of the features of the Bode algorithm has been discussed in the above explanation of the Nyquist plot. The major difference between the Bode and Nyquist routines is the manner in which the information is displayed. The Bode graph displays two separate plots of magnitude and phase versus radian frequency. The magnitude plot is converted to decibels using the relation
\[ \text{Magn} = 20 \log_{10}(\text{Magn}) \]

and the phase is converted to degrees by computing
\[ \text{Phase}_{\text{deg}} = \frac{(180/\pi)(\text{Phase})}{\text{Phase}}. \]

The frequency is plotted on a logarithmic scale along the abscissa. The plots of magnitude and phase are superimposed on the same graph with the ordinate values of -180° phase and 0dB magnitude aligned. This is convenient for the designer when reading gain and phase margin from the graph. Since phase margin is read at the zero crossover of the magnitude curve and gain margin is read at the -180° crossover of the phase curve, these values can be read directly without shifting either curve up or down.

The program implementation is much like that used for the Nyquist procedure. The Bode routine calculates the numbers required for the plots and the procedure "Plot_Bode" then converts the numbers to a graphical display. The same procedure applies with regard to the supplemental graph options menu as with the Nyquist routine.

E. ROOT LOCUS

The root locus procedure implemented in this routine is a "single parameter", or "gain locus". It graphically represents the movement of the roots of the system's closed-loop characteristic equation on the S-plane while varying the system gain. Since the location of the
characteristic roots is an indication of the system response, the designer can examine the effect of increasing gain on system behavior.

The algorithm is straightforward as can be seen in Figure 3-15. The user selects the range of variable gain values to use in the computation and the display window size. The program then plots the roots of the open-loop G-equivalent block as zeros and begins to iterate through the range of variable gains while computing the roots of the unity-feedback, closed-loop system. These roots are then plotted.

Given an open-loop equivalent transfer function:

\[ G_{o.l.}(s) = \frac{K \cdot P(s)}{Q(s)} \]

where \( P(s), Q(s) \) are polynomials.

and

\[ G_{c.l.}(s) = \frac{K \cdot K_v \cdot P(s)}{Q(s) + K \cdot K_v \cdot P(s)} \]

where \( K_v \) is the variable gain.

then

PLOT \{open loop\} Zeros.

WHILE \( K_v < K_{max} \) THEN DO

SET ClosedLoopPoly = DenCoeff + \([K \cdot K_v \cdot NumCoeff]\).

CALL RootFinder(ClosedLoopPoly).

PLOT Poles.

INCREMENT \( K_v \).

ENDWHILE.

Figure 3-15. Root Locus Algorithm
F. TWO-PARAMETER ROOT LOCUS.

The two-parameter root locus is one of the more interesting routines in the LCS-CAD package. It is useful in a number of design situations including the most simple case of designing a cascade compensator. An example is given in Figure 3-16. The system is to be compensated with a single cascade compensator with one pole and one zero. If the blocks are combined into a single equivalent block and the closed loop characteristic equation is derived, the resultant polynomial will be a function of two unknown parameters "a" and "b". These two parameters define the location of the pole and zero of the compensator and as they are varied, the response of the system is changed. If a locus of the roots is plotted, the designer can see the effects of changing the compensator pole and zero on the overall system response.

The LCS-CAD procedure works in much the same manner as in the preceding example. The program must, however, be able to parse the user's characteristic polynomial coefficient equations in order to "understand" the relations and be able to iteratively substitute in values for a and b. The simplified algorithm is outlined in Figure 3-17.

The user is requested to provide the desired limits on the parameters a and b and to decide whether to "step" a or b.
Given a system $G(s)$ and a compensator $G_c(s)$

$$G(s) = \frac{100}{s(0.1s + 1)} \quad \quad G_c(s) = \frac{s + a}{s + b}$$

then the closed loop equivalent transfer function is

$$G_{eq}(s) = \frac{1000(s + a)}{1000(s + a) + s(s + 10)(s + b)}$$

and the characteristic polynomial is

$$C.P. = s^3 + (10 + b)s^2 + (10b + 1000)s + 1000a$$

If "a" is to be varied from, say, 10 to 20 and "b" from 50 to 200 then, substituting the initial values into the characteristic equation yields

$$C.P. = s^3 + 60s^2 + 1500s + 10000$$

the roots of which are

$$(s + 10)(s + 25 \pm j19.37).$$

These roots are plotted and a or b is incremented and the process repeated through the range.

Figure 3-16. Two-parameter Root Locus Example

This input page is shown in Figure 3-18 for the example problem given earlier. The parameter that is stepped is varied through five discrete values within its range and for each of these values, the other parameter is incremented fifty times. This produces a "family" of locus curves. Either a or b may be stepped at the user's option.
Given a system's characteristic polynomial:

\[ C.P. = E_n a^n + E_{n-1} a^{n-1} + \ldots + E_1 a + E_0 \]

where \( E_n, E_{n-1}, \) etc. are algebraic expressions in \( a \) and \( b. \) Also \( a \) is to be stepped from \( a_1 \) to \( a_2 \) and \( b \) varied from \( b_1 \) to \( b_2. \)

Then

\[
\begin{align*}
&\text{SET } a = a_1. \quad \text{SET } b = b_1. \\
&\text{SET } \delta A = (a_2 - a_1)/5. \quad \text{SET } \delta B = (b_2 - b_1)/50. \\
&\text{WHILE } a < a_2 \text{ DO} \\
&\quad \text{WHILE } b < b_2 \text{ DO} \\
&\quad \quad \text{FOR } i = 1 \text{ TO } n \text{ STEP } 1 \text{ do } \{ \text{for each coefficient} \}
\quad \quad \quad \text{CONVERT } E_i \text{ from INFIX to POLISH.} \\
\quad \quad \quad \text{SUBSTITUTE values for } a \text{ and } b. \\
\quad \quad \quad \text{COMPUTE } E_i. \\
&\quad \text{ENDFOR.} \\
&\quad \text{CALL RootFinder.} \\
&\quad \text{PLOT Roots.} \\
&\quad \text{SET } b = b + \delta B. \\
&\text{ENDWHILE.} \\
&\text{SET } a = a + \delta A. \\
&\text{ENDWHILE.}
\end{align*}
\]

Figure 3-17. Two-Parameter Root Locus Algorithm

<table>
<thead>
<tr>
<th>*** Parameter Selection Page ***</th>
</tr>
</thead>
<tbody>
<tr>
<td>You will be varying the two parameters, ( A ) and ( B, ) through a range of values you select. You will also choose to STEP either ( A ) or ( B ) which means the chosen parameter's range will be divided into five (5) equal increments to plot; the other parameter varies smoothly through its range</td>
</tr>
<tr>
<td>A-minimum: 10</td>
</tr>
<tr>
<td>B-minimum: 50</td>
</tr>
<tr>
<td>Step A or B? : A_</td>
</tr>
<tr>
<td>Press &lt;F1&gt; to change previous entry</td>
</tr>
</tbody>
</table>

Figure 3-18. Two-parameter Root Locus Parameter Input Page
Next the user must provide a window size for viewing the root locus. This is handled in much the same manner as described for the single-parameter root locus routine. Finally, the coefficient equations must be input. Figure 3-19 shows a completed page for the example third order system equation input.

**Three Parameter Root Locus - Coefficient Input**

<table>
<thead>
<tr>
<th>3</th>
<th>s = 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>s = 10 + B</td>
</tr>
<tr>
<td>1</td>
<td>s = (10*B) + 1000</td>
</tr>
<tr>
<td>0</td>
<td>s = 1000*A</td>
</tr>
</tbody>
</table>

Press <F1> to change previous entry

Figure 3-19. Two-parameter Root Locus Coefficient Input

Note that the equations are input in algebraic, or "infix" notation. The available operators include (+) addition, (-) subtraction, (*) multiplication, (/) division, and (^) exponentiation. These operators follow a hierarchical precedence with exponentiation operations being done first, followed by multiplication and division, and finally addition and subtraction. Operations like
multiplication and division which have the same precedence are performed from left to right when conflicts arise. To change this order of precedence, parentheses may be used around any set of operations. These parenthetical expressions have the highest priority and, when nested, the innermost operations within parentheses are done first.

This scheme follows closely the protocol used in most calculators and high level programming languages.

Infix notation, while convenient for the program user, does not lend itself well to computer manipulation. A better way to represent equations for the computer is the so called "reverse Polish notation". In reverse Polish notation, the operands of an equation are entered first followed by the operator. For example, the infix expression

\[ 3 \times 4 + 5 \]

would be represented as

\[ 3 \ 4 \ \times \ 5 \ + \]

in reverse Polish notation. The numbers 3 and 4 are entered and multiplied, then 5 is entered and added to the previous result. Using the concept of a "stack" the reverse Polish expression is easy to evaluate.

Recall that a stack is a last-in-first-out queue whose operation is analogous to a stack of trays. To operate the stack, the program calls a "push" procedure to place an item on the stack, and a "pop" procedure to remove the top
item. Now, using the example given above, an arithmetic evaluation procedure can be illustrated. Figure 3-20 demonstrates such an implementation.

![Figure 3-20. Stack Operation Example](image)

The basic equation evaluation algorithm can be outlined then in three steps:

1. Scan the reverse Polish equation term-by-term.
2. If the term is a constant then push it onto the stack.
3. If the term is an operator then pop the first two items off the stack, apply the operator, and push the result back onto the top of the stack.

When the algorithm is completed the answer to the expression will be on the top of the stack.

To get the infix equation into reverse Polish form is a bit more difficult than simply evaluating the Polish expression. Of special consideration when building the Polish form of the equation are the operator priorities and the use of parenthesis to change those priorities. A set of rules can be written which outline the conversion procedure [Ref. 2]. These rules are listed in Figure 3-21 along with an illustrative example of their application.
Given the infix expression:

\[ 6 + (5 - 4/2) \times 3 \]

and the following operator priority table:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>, , / ,</td>
<td>4</td>
</tr>
<tr>
<td>+ , -</td>
<td>3</td>
</tr>
<tr>
<td>operands</td>
<td>2</td>
</tr>
<tr>
<td>( , ), ,</td>
<td>1</td>
</tr>
<tr>
<td>, , , ,</td>
<td>0</td>
</tr>
</tbody>
</table>

Using a result string called RPN, and an operator stack the rules for infix to reverse Polish conversion follow:

**RULE**

(1) If an operand is encountered, move it to RPN.
(2) If an operator is encountered, move all higher priority operators on the stack to RPN and push the new operator onto the stack.
(3) If a left parenthesis is encountered push it onto the stack.
(4) If a right parenthesis is encountered, pop all operators off the stack and append them to RPN until a left parenthesis is encountered. Discard both parentheses.
(5) When finished with the infix expression, pop all remaining operators from the stack and append them onto RPN.

Applying these rules to the example problem above gives:

<table>
<thead>
<tr>
<th>RPN</th>
<th>STACK</th>
<th>INFIX</th>
<th>RULE</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td></td>
<td>6+(5-4/2)*3</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>+</td>
<td>+(5-4/2)*3</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>+</td>
<td>(5-4/2)*3</td>
<td>3</td>
</tr>
<tr>
<td>6 5</td>
<td>+</td>
<td>-4/2)*3</td>
<td>1</td>
</tr>
<tr>
<td>6 5</td>
<td>+</td>
<td>4/2)*3</td>
<td>2</td>
</tr>
<tr>
<td>6 5 4</td>
<td>+</td>
<td>-4/2)*3</td>
<td>1</td>
</tr>
<tr>
<td>6 5 4</td>
<td>+</td>
<td>2)*3</td>
<td>2</td>
</tr>
<tr>
<td>6 5 4 2</td>
<td>+</td>
<td>-/)*3</td>
<td>1</td>
</tr>
<tr>
<td>6 5 4 2</td>
<td>/</td>
<td>+</td>
<td>4</td>
</tr>
<tr>
<td>6 5 4 2</td>
<td>/</td>
<td>-</td>
<td>3</td>
</tr>
<tr>
<td>6 5 4 2</td>
<td>/</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>6 5 4 2</td>
<td>/ -</td>
<td>+</td>
<td>1</td>
</tr>
<tr>
<td>6 5 4 2</td>
<td>/ -</td>
<td>*</td>
<td>1</td>
</tr>
<tr>
<td>6 5 4 2</td>
<td>/ -</td>
<td>*</td>
<td>1</td>
</tr>
</tbody>
</table>
The LCS-CAD routine to compute and display the two-parameter root locus is called "TwoParameterRootLocus". It calls several sub-programs including "Infix_to_Polish" which does the conversion of the coefficient expressions to the reverse Polish form, and "Compute_Polish" which evaluates the reverse Polish expressions. Other procedures, like "Coeff_Input", "Select_Parameter_Range", and "Select_Window_Size", prompt the user for information required by the program.

Figure 3-22 shows the two-parameter family of curves generated by the program for the sample problem introduced at the beginning of this section. The designer wishes to place the system poles at a specific location on the S-plane, the plot will enable him to see qualitatively whether or not it is possible with a given set of parameters varying over specified ranges. A printout of the numbers used to generate the plot can then be examined for the values of a and b necessary to place the poles in the desired locations.

As with all the other graphing routines, the options menu appears after the plot is generated giving the user the opportunity to title the graph or print the graph or its corresponding numbers. An additional window appears on this graph to show a legend for easy curve identification.
G. TIME RESPONSE

The time response module gives the user the capability to see how the system will respond to a "standard" input in the time domain. The user can subject his system to a choice of step, ramp, impulse, or sinusoid inputs. Since most design work will be done in the frequency domain, this module will give the designer a tie-in to the time domain and thereby a physical interpretation of the system's response.

The time response algorithm first converts the continuous-time, input-output description of the system into a discrete-time, state-space equivalent. The states of the system at any time and the output can then be generated knowing only the input and the previous system
states. This process is repeated for discrete time steps until the desired end time is reached. This method is, of course, not the only approach available. Numerical differential equation solvers, particularly the Runge-Kutta method, are often used to solve this type of problem. The "sampled-data" method was compared to the Runge-Kutta and performed favorable when using a small step size and a large number of samples.

The first step in solving the time-response problem is to convert the input-output representation into a matrix form. An example of this conversion procedure is shown in Figure 3-23. The form of the A-matrix is always the same with one's along the upper co-diagonal and the transfer function denominator coefficients along the bottom row. This is the so-called "companion form" [Ref. 3]. Likewise, the C-matrix contains the coefficients of the denominator polynomial. The single B-matrix entry indicates that the system only has one input as expected.

To convert the system to a discrete-time equivalent, the continuous-time system

\[ \dot{x}(t) = Ax(t) + Bu(t) \]
\[ y(t) = Cx(t) \]

must be "mapped" into the system

\[ x(k+1) = \Phi x(k) + \Gamma u(k) \]
\[ y(k) = Cx(k). \]
Knowing the relationship between the two systems, it can be shown that

$$\Phi = I + A \mu(T)$$

and $$\Gamma = \mu(T)B$$
where $T$ is the sampling period and $\mu$ is defined as

$$\mu(T) = \int_0^T e^{At} \, dt = T \sum_{k=0}^{\infty} \frac{A^k T^k}{k!}$$

This infinite series expansion for $\mu$ can be approximated by taking enough terms to ensure precision to some acceptable value. Once the $\mu$ matrix has been computed, then the $\Phi$ and $\Gamma$ matrices can also be calculated. Assuming zero initial conditions for all the states at time $t=0$ gives a starting point from which the states at $t = 0 + t$ may be computed, and so on until $t = t_{\text{max}}$. Since the output of the system is simply one of the states, it can be extracted from the computations and plotted.

The LCS-CAD procedure which implements the time response module is called "TimeResp". The algorithm is outlined in Figure 3-24. The actual program implementation appears more complex because of the mechanics of performing the matrix and vector operations. This is simplified somewhat by the procedures "Matrix_Multiply" which multiplies two square matrices together, "Matrix_Vector_Mult" to multiply a matrix and a vector together, and "Scalar_Mult" which multiplies each of the elements of a matrix by a scalar constant.

The time increment is set to a constant value of 0.0005 seconds giving 2000 calculation increments. Because of the time required to plot all these increments, only one in five points are actually graphed.
Given a closed-loop transfer function of the form:
\[ G(s) = K \frac{a_n s^n + \cdots + a_1 s + a_0}{b_m s^m + b_{m-1} s^{m-1} + \cdots + b_1 s + b_0} \]

where \( m \geq n \) and \( K \) is block gain. If \( T \) is to be the sampling time, then

```plaintext
*** Fill the A-matrix ***
FOR i,j = 1 TO m DO
  IF j = i+1 THEN SET A[i,j] = 1. {1's in upper
  ELSE SET A[i,j] = 0. co-diagonal}
ENDIF.
ENDFOR.
FOR i = 1 TO m DO
  A[m,i] = -b_i. {denom. coeff along bottom row}
ENDFOR.

*** Fill the C-matrix ***
FOR i = 1 TO n DO
  SET C[i] = a_i * K.
ENDFOR.

*** Initialize Psi and Atemp ***
SET Atemp = A.
SET Psi = I + (A * T/2). {this is first term in series}

*** Compute additional series terms ***
WHILE Maxrowsum >0.1% DO {check contribution of next
term in series}
  SET k = 2.
  SET Psi = Psi + (A^k * T^k)/(k + 1)!. 
ENDWHILE.
SET Psi = Psi * T.

*** Calculate the Phi Matrix ***
SET Phi = I + (A * Psi).

*** Calculate the Gamma Vector ***
SET Gamma = Psi[i,m] {since input vector U is
[0 0 ... 0 1]^T, Gamma is last
column of Psi matrix}

*** Compute the next state vector ***
SET xold = 0. SET t = 0.
WHILE t < tmax DO
  SET xnext = Phi * xold + Gamma * input.
  SET y = C * xold.
  PLOT y.
  SET xold = xnext
  SET t = t + t.
ENDWHILE.
```

Figure 3-24. Time Response Module Algorithm
H. UTILITIES

Several utility procedures are available from within the LCS-CAD program. These enable the user to display the current loop blocks in either polynomial or factored form, or to enter an arbitrary polynomial and have the program find and display the roots. There are three procedures which implement these functions. The first is "ShowFactors" which, as the name implies, allows the user to view the loop blocks in factored form. The procedure is essentially an on-screen formatting routine since the factors of every block have already been calculated and stored in the block record.

The second procedure is "ShowPoly". It is the complementary routine to ShowFactors. It formats the polynomial form of the loop’s block transfer functions on-screen for viewing. Here too, no numeric calculation is required as the block polynomial coefficients are available in the block record.

The third procedure is called "UserPoly". This routine allows the user to input the coefficients of a polynomial and then calls the subprogram "RootFinder" (discussed earlier) to find the roots of the polynomial. The program then displays the factors in much the same manner as the ShowFactors routine.
I. HELP SCREENS

LCS-CAD has on-line help available from any of the three menus. The help screens are invoked from menus which resemble the program menu. From the help menus the user may select any of the choices normally available to him from the program menus and help screens will be displayed with help information specific to the function selected.

From the Main (program) Menu, the user may select <H>-HELP. This will bring up the Main (help) Menu. If the user select help with the <I> Input/Change function, a second help menu is displayed much as it is with the main menu selections. From there, the user may select any of the input menu choices and one or more help screens for that choice will be displayed. A similar secondary help menu is displayed if the <U> Utilities help is requested.

Unlike the Main Menu help, which can get help for any of the other menu items, if the user is in the Input Menu or the Utilities menu and calls for help from there, only the Input Help Menu or Utilities Help Menu respectively will be displayed. Figure 3-25 shows the relationship of the help screens to the program menus.
Figure 3-25. Help Available within LCS-CAD
IV. SUMMARY AND RECOMMENDATIONS

The LCS-CAD program is a useful and powerful tool for the designer and a simple and easy to use package for the student. The program supports most of the major design and analysis tools required for continuous-time, linear control system work including Bode, Nyquist, single- and two-parameter root locus, time response, and automatic transfer function block manipulation.

The program does, however, have its shortcomings and areas where improvements are possible. Several of these areas are discussed below.

(1) Currently, the program's size is constrained by the limits of the Turbo Pascal compiler. The routines could be made to run faster and the code size streamlined if the code were translated into another language which supported memory usage beyond the 64K limit imposed by Turbo Pascal. This would eliminate the need for chaining program segments into and out of memory to disk. This project should not be undertaken lightly, since Turbo Pascal is unique in its support for low-level system calls and in-line assembly language code, both of which are used in LCS-CAD routines. The best potential alternative choice for a language now appears to be Modula-2. It is similar to Pascal in construct, but allows full memory use and at least one such compiler now available (LogiTech Modula-2) appears to support most Turbo Pascal functions.

(2) Additional modules could be added to the software. A Parameter Plane module, for instance, could be added to enhance the value of the package. This is yet another tool which is a bit more sophisticated and can be applied to more difficult design problems than the tools currently in the LCS-CAD system. Another possible candidate module for inclusion in the package is the "Function Minimization"
subroutine. This routine allows the user to select a "cost function" and the program will vary the parameters of the problem to minimize the cost function. This is a type of optimal control method.

(3) Several improvements to existing routines could be made. These are listed below in no particular order or significance.

a. Adding a moveable cursor to the graphics screens to allow the user to "point" to an area of interest on a curve and have the important parameters printed on-screen. For instance, in the Bode routine, the user could move a cursor to the gain crossover point and have the frequency and the phase margin printed.

b. Adding the ability to handle the state-space form of input. Currently, only the input-output description of a system is allowed, although the state-space form is used extensively in the time response module for computation. This would add even greater flexibility to the user's choice of input.

c. Additional error trapping routines need to be added to the program. Fairly extensive efforts have been applied to keep user inputs from "bombing" the program, however, when an error occurs and aborts the program, it can be most frustrating to a user. DOS-level interrupt handlers can be written to take care of most common errors like divide-by-zero and I/O faults. These programmed interrupt handlers can override the default DOS handlers which issue a cryptic message and abort to the operating system.

d. A "zoom" feature is needed on the graphics routines which will allow the user to change the plotting scales without rerunning the entire graphics routine again. This may be accomplished with strategically placed IF/THEN statements.
e. A more sophisticated input routine may be devised to allow the user to input the ENTIRE block diagram with multiple loops. The difficulty of implementing this option is to maintain a user interface which is simple and intuitive. Block interconnections and feedback path descriptions can easily become confusing and ambiguous.

f. Add the capability of the program to handle discrete time systems described by Z-domain functions.

These suggestions for future improvements to the LCS-CAD program represent specific changes as well as areas for broader, more in-depth work. The modular nature of the program structure and its menu-driven format make it a simple matter to append additional subroutines. The Pascal language with its inherent top-down structure makes for a more readable program and enables follow-on programmers to quickly and easily understand the algorithm implementation and alter it to suit their needs.

LCS-CAD is a full-featured engineering program written to operate on the most popular desk-top computer available today. With periodic maintenance and updates, this program can meet the needs of control system designers and student for many years to come.
APPENDIX A. INPUT UTILITIES

Appendix A is an excerpt from the user's documentation for a set of Turbo Pascal input and menuing utilities. These utilities, called TURBO-UT.PAS, were written by:

Donald R. Ramsey
Larry Romero
727 Bunker Hill #70
Houston, Texas 77024

and distributed through the public domain. The utilities consist of a number of procedures and functions which allow the programmer to easily write sophisticated menuing systems and input routines. These utilities were used extensively in building LCS-CAD.

The following documentation describes how the programmer should use the procedures and function in a program including calling methods and parameter constraints.

DOCUMENTATION FORM:

The procedures will be given in the following form:

Procedure name(Variable 1,    (type var: description of var)
             ,
             Variable n) (type var: description of var)

Description of routine and calling method.
Discussion if necessary.
What is returned to the main program.

PROCEDURES:

Msg( String         (str: string to be displayed )
    Col,Row)      (int: Column; row for display)

Description: Display a message at the specified column and row of the video.
Center( String, Col, Row, LineLength )

Description: clear the line to center on from the col, row to the line-length, then center the string on this line.

InvVideo( String )

Description: display a string in inverse video and return to the calling routine with the background set to black and the text white.

Color( Background, Text )

Description: color the video as desired.

Box( X1, Y1, X2, Y2, Z )

Description: draw a box that has an optional line as a divider. If you do not wish to have the extra line divide the box, just set Z = Y1.

Option;

Description: allow the user to press a key and return that character as an upper character to the calling routine.

Returned: Ch ( upper case )

StripSpaces( String, NewStr )

Description: Strip spaces from the end of a string.

Returned: NewStr is a variable parameter that is returned to the calling routine.
ClrWnd: X1, Y1, (int: upper left of area to clear)
X2, Y2) (int: lower right of area to clear)
Description: this is an alternate method to that of
defining a window and doing a CLRSCR for that window.
The advantage of this method is that the original
window is left intact and operations can be performed without
keeping track of the original window. The disadvantage is
that it is a little slower than CLRSCR.

SaveScreen
Description: save a video display in memory for a quick
flash back when necessary.

FlashScreen
Description: this routine will re-display the screen saved
by the SaveScreen procedure.

BeepTone, (int: the tone to sound (Ex: 350 ))
Duration: (int: the time to delay (Ex: 500 ))
Description: turns on a tone for the desired duration
then terminates the sound. You may test the sounds
by running TESTSCRN.COM.

Set_cap_num
Description: display on the 25th line of the screen
the current status of the CAP, NUM, Ins keys.

Set_cap_num(Caps, ( ch: set to 'C' for caps )
Num, ( ch: set to 'N' for nums )
Ins, ( ch: set to 'I' for insert )
Description: set the keyboard as desired by the programmer.
Ex: Set_cap_num('C', 'N', 'I')
will set the cap lock, the num lock, and turn
insert on.
Ex: Set_cap_num('C', 'N', 'I')
will set only the num lock.
Ch = edit_key(Ch)  (ch; variable returned)
Description: A routine to determine if an edit key (Home, End,
arrow key, etc.) or a function key was pressed.

Discussion: IBM returns a two byte string for any edit
or function key that is pressed. In order to test for this
2 byte string you must read the keyboard twice, testing
the second byte for the edit or function key value. So,
you must read the KBD for a character and if a key is
pressed, call the routine (the procedure does the second
READ(kbd, ch)). At present, only the edit keys and the
10 function keys are checked. However, you could easily
expand the procedure to check the shift states, etc.
Example call:
READ(kbd, ch);
if keypressed Ch = edit_key(Ch);

Returned: the procedure will modify the variable
Ch if one of the edit or function keys was pressed or
leave Ch as it was in the first READ(kbd, ch); if no edit or
function key was pressed, you should look at the routine
to determine what Ch will return if it has been modified
(I always have to lock, but, I know the function keys
will return 'a for (F1), ...,' for (F9), ' for (F10).

Get_template( template_num (int: the number of the template )
template; (str: variable returned to caller)
Description: get the required template for use in the
input procedure.

Input type, (ch: A for alphanumeric,
N: for numbers,
F: for formatted entries )
default, (str: the default string that will be displayed
in the input field )
col, row, (int: the position for the input )
length, (int: the field length for the input or
the template number if a formatted)
uppercase (bool: true for uppercase letters,
false otherwise)
F1 (bool: variable returned true if the F1 key
was pressed for the entry )
F10 (bool: variable returned true if the F10 key
was pressed for the entry )
Description: This is the main procedure for getting input from a user. The numeric keypad, the Ins key, and the CAP lock keys are set as desired by the programmer and these keys are constantly monitored to see if their status has been changed by the user. The key status is displayed on the 25th line of the screen in inverse video. As presently configured, the status of these three keys can be changed at any time by the user.

The routine allows the user to use the keyboard as he would normally expect. The arrow keys function when the NUM lock is not set. The Home and End keys respond to send the cursor to the start or end of the input line, and the Ins key state will allow a letter to be inserted in the input string if it is set to ON.

Returned : the entry of the user is return to the calling procedure in the global variable ANSWER.

Prompt: string1,  \( \text{str: string to be displayed on line 22} \)  string2  \( \text{str: string to be displayed on line 23} \)  
Description: clear line 22 and 23 of the video then display string1 and string2 there.

Say_prompt(prompt_num) \( \text{int: the prompt to display} \)  
Description: specify string1 and string2 to be displayed in the Prompt procedure. This is used primarily by the Input_handler.

Get_default(Default_num, \( \text{int: the default for an input} \)  Default \( \text{str: the variable returned} \)  
Description: provide the defaults to display in the input field for the Input_handler.

Do_validation(Valid_num, \( \text{int: the number of the validation routine} \)  Valid \( \text{bool: variable returned false if invalid entry} \)  
Description: provide a routine to validate any entries that were made from the Input_handler. If the entry is invalid on return from this procedure, the Input_handler will require the user to re-enter the data.

Returned : the boolean variable VALID is returned.
Input_handler: string, (str: 5 character string (Ex: 'N010B')

1st ch: N for new entries
       C for changes
       D for re-display
nums 1-2: first element number of
         P[] array to use
nums 3-4: last element of the
         P[] array to use
Escape): bool: variable returned true if <F1> was
         pressed at the first entry

Description: this handler provides for full screen editing of
user inputs, provides a means for changing entries, and
redisplays a record.

Discussion: the programmer must provide the P[] array,
specifying each input that is required. The form for
P[] is

P[1] := '2505A02501T010102'

P[n] where n is limited to 35 (35 entries per page)

Each element of P[] is defined below:

<table>
<thead>
<tr>
<th>element No.</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-2</td>
<td>the column for input</td>
</tr>
<tr>
<td>3-4</td>
<td>the row for input</td>
</tr>
<tr>
<td>5</td>
<td>type input: A for alphanumerics</td>
</tr>
<tr>
<td></td>
<td>N for numbers</td>
</tr>
<tr>
<td></td>
<td>F for formatted entries</td>
</tr>
<tr>
<td></td>
<td>$ for dollar entries</td>
</tr>
<tr>
<td>6-8</td>
<td>the length of the input field or</td>
</tr>
<tr>
<td></td>
<td>the template number of a formatted</td>
</tr>
<tr>
<td></td>
<td>entry</td>
</tr>
<tr>
<td>9-10</td>
<td>array element of Filvar[] array.</td>
</tr>
<tr>
<td></td>
<td>Filvar[] is the global variable</td>
</tr>
<tr>
<td></td>
<td>that is returned to the calling</td>
</tr>
<tr>
<td></td>
<td>routine</td>
</tr>
<tr>
<td>11</td>
<td>set to T if you wish the Caps</td>
</tr>
<tr>
<td></td>
<td>lock set</td>
</tr>
<tr>
<td>12-13</td>
<td>the default number (see Get_default)</td>
</tr>
<tr>
<td>14-15</td>
<td>the prompt number (see Say_prompt)</td>
</tr>
<tr>
<td>16-17</td>
<td>the validation No. (see Do_validation)</td>
</tr>
</tbody>
</table>

The programmer must also provide for defaults, prompts,
and validation. The placement of these procedures is provided
in the utility.
Example: the following procedure will call the handler:

```pascal
procedure Get_inputs;
  procedure Get_variables;
  begin
    P[1] := '2505N00801F010201';
  end;
  begin
    Get_variables;
    input_handler('N0102');
  end;
```

This example will provide input for two variables. On return from the handler, FILVAR[1] will contain the input from the parameters specified in P[1] and FILVAR[2] will contain the input from P[2].

Changes to be made to Filvar[i..n] by calling the handler in the change mode. For example, if there were 7 variables in the Filvar array, the call
```
input_handler('C0207')
```

The handler may also be called in the Display mode. It will then display all variables except the numbers. The numbers may be displayed formatted using the FMT_REAL function.

```
FUNCTION

FMT_REAL(number, length, decimals) { real: number to format }
```

Main Menu

Description: provide a skeleton for a main menu. The procedure will draw a box around the menu items and verify the choice of the user. All that is required of the programmer, is the menu selections and a list of Okchoices ( Okchoices is a list of all the choices that may be selected by the user )
Description: a function to format a real number with a comma
and the decimals as desired.
Example: Fat_Real( 1010.268,7.2 ) would return 1,010.26

UpcaseStr(S)  ( str: string to convert to upper case )
Description: to convert any string to upper case characters.
This function may be useful when using the Turbo Toolbox for
converting an index string to all upper case letters.

ConstStr( Character, \( [\text{the character to fill the string with }] \)
Number)  \( [\text{the number of characters in the string}] \)
Description: fill a string with the character of the programmer's
choice. This would be useful for drawing a line of characters on
the screen.
Example: gotoXY(1,4); write(ConstStr("-",20));
This example would draw a line of equal signs at line 4 of
the video.
APPENDIX B

Appendix B is a listing of the Pascal source code which makes up the major modules of the LCS-CAD program. In general, driver programs are included first followed by their supporting subprograms. Not included in the listings are the source modules for the Borland International Turbo Graphix Toolbox and the input routines described in the thesis text and in Appendix A.
(* Program Control_CAD contains the calls for the main menu proc *)
(* and also the files for the single-parameter root locus program *)
(* ***************************************************************)

program Control_CAD;
(* ***************************************************************)
(* The following include-files contain routines and procedures *)
(* to handle graphics calls from the main procedures of this *)
(* program. They are a part of Borland International's Turbo *)
(* Graphix Toolbox, a commercially available product. *)
(* ***************************************************************)

{$1 typedef.sys)
{$1 graphix.sys)
{$1 kernel.sys)
{$1 windows.sys)
{$1 polygon.hgh)
{$1 axis.hgh)
(* ***************************************************************)
(* These include files are utility procedures and functions to *)
(* facilitate input handling. They are public domain procedures *)
(* ***************************************************************)

{$1 UT-HOD01.INC )
{$1 UT-HOD02.INC )
{$1 UT-HOD03.INC )
(* ***************************************************************)
(* Graphenu contains procedures to open a window on each graphic *)
(* screen with a menu of use options. The selections allow *)
(* printing a hardcopy of the graph on a dot-matrix printer, *)
(* dumping the data points to the printer, or constructing and *)
(* placing a title block on the graph. *)
(* ***************************************************************)

{$1 Graphenu.inc)
(* ***************************************************************)
(* The include-file Roots.Inc is a set of procedures to find *)
(* the roots of a given polynomial in coefficient form. The *)
(* algorithm used is a modified version of the Bairstow method. *)
(* ***************************************************************)

{$1 Roots.INC )
(* ***************************************************************)
(* The Include-file SHOWROOT.inc will display the input or *)
(* calculated roots of the G-equivalent block diagram. Both *)
(* ***************************************************************)
(**** poles and zeros will be displayed.  ****)
(*****************************************************************)

(*1 ShowRoot.inc)

(*****************************************************************)

(*1 RootLoc.inc)

(*****************************************************************)

The include-file ROOTLOC.INC is a procedure to draw the
plot of a single-parameter root locus through a user-supplied
range of gain (K).
(*****************************************************************)

(*****************************************************************)

Procedure MainMenu;

var
    I, Tab : integer;
    Dkchoices : set of char;
    Helpfile : file;

begin
    procedure ProgramExit; (displays warning about program about to end)
     —begin
        Clrscr; Center('This Program is about to end',1,11,80);
        highvideo; Center('Verify Ok (Y/N)',1,13,80); lowvideo;
        repeat
            Option; if not (Ch in [T,'N']) then beep(350,150);
        until Ch in ['Y'/N ];
     —end;

    procedure MenuItem(pick:char; description:string; color:integer);
     (allows easy selection of menu colors)
     —begin
        textcolor(color);
        writeln(':Tab, (', pick, ')'); textcolor(White); writeln(pick);
        textcolor(color); writeln('-',description);
     —end;

    begin
        Clrscr; TextColor(White);
        gotoXY(1,24); Write('Revision date: 9/04/86');
        Center('MAIN MENU ',1,4,80); (display main menu)
        for I:= 1 to 4 do writeln('');
        Tab:= 23;
        for I:= 1 to 4 do writeln('');
        writeln('Input/Change Transfer Function(s)',green);
        writeln('Location of Char. Eq. Roots',green);
        writeln('Frequency Analysis',green);
        writeln('Root Locus Analysis',green);
        writeln('Two Parameter Root Locus',green);

96
MenuItem('T', 'Te
c Response', green);  
MenuItem('U', 'Utilities', green);  
writeIn;
MenuItem('H', 'Help', blue);  
MenuItem('Q', 'Exit Program', lightmagenta);
HighVideo;
TextColor(Yellow);
Box(120, 26, 25, 6); writeln('');
Set_Cap_Num('C'; ' ', ''); Say_Cap_Num;
TextColor(White); Center('Press Your Selection', 21, 22, 36); LowVideo;
(set legal choices depending on whether user has entered  
block information or not)
if not NBBlocks in [1..99] then DKchoices := ['T', 'U', 'H', 'Q', 'P']
else DKchoices := ['T', 'L', 'F', 'R', 'U', 'T', 'H', 'Q', 'P'];
repeat
Option; if not Ch in DKchoices then
begin
Beep(350, 150); TextColor(Yellow);
if NBBlocks <= 0 then
begin
msgWARNING : First INPUT the block descriptions!', 1, 25);
end;
end;
until Ch in DKchoices;
case Ch of
'I' : Begin
Assign(InputFile, 'Input.chn');
Chain(InputFile);
end;
'R' : Root_Locus(G_e);  
'B' : Begin
Assign(TwoParamFile, 'TwoParam.chn');
Chain(TwoParamFile);
end;
'L' : ShowRoots(G_e);  
'F' : Begin
Assign(FreqFile, 'Freq.chn');
Chain(FreqFile);
end;
'T' : Begin
Assign(TimeFile, 'TimeResp.chn');
Chain(TimeFile);
end;
'U' : Begin
Assign(UtilFile, 'UtilMenu.chn');
Chain(UtilFile);
end;
'H' : Begin
Assign(HELPfile, 'HELPMenu.chn');
Chain(HELPfile);
end;
'Q' : begin
ProgramExit; if Ch='Y' then Exit := true;
end;
end;
171 (**********************************************************************************************)
172 (*Program Starts Execution*)
173 (**********************************************************************************************)
174 175 B---------------------------------
176 : begin
177 : InitGraphic; LeaveGraphic; ClrScr; (initialize screen)
178 : Fillchar($,100,$205); S:= copy($,1,20); (use special character to draw line)
179 : Exit:=false; (initialize boolean)
180 : NBlocks := 0; (reset problem)
181 : -end.
182 : ----------------------------------
183 : repeat
184 : MainMenu; (repeatedly call main menu until user wants to quit)
185 : until Exit = true;
186 : Set_Cap_Num(' ',' ',' '); Say_Cap_Num; (set caps, insert, and num lock off)
187 : NBlocks := 0; (reset problem)
188 : end.
Program Input(input, output);

(*$ TYPEDEF.SYS)  (program type & variable definitions - same as CAD.PAS)
(*$ DIRECTY.INC)  (displays directory of available blocks)
(*$ UT-MOSO1.INC)  (input utility routines)
(*$ UT-MOSO2.INC)  (input utility routines)
(*$ UT-MOSO3.INC)  (input utility routines)
(*$ EXPAND.INC)  (expands factors into polynomials)
(*$ ROOTS.INC)  (finds factors (roots) of polynomials)
(*$ MAKSEDG.INC)  (combines loop blocks into single equivalent block)
(*$ INFUTHUMNC)  (help screen driver)

(******************************************************************************)
(** Block input is called from input, add, and change procedures to input **)  
(** a simple block. **)  
(******************************************************************************)

Procedure Block_input(var Block_Description: Block; BlockIndex: integer; readError: boolean);
begin
var
  i,j,code : integer;  (i,j; loop counters; codes error code from Var proc)
  temp,temp2 : char;  (temporary holding var for input)
  VertPos,
  HorizPos;
  PostCounter : integer;  (screen prompt position handler variables)
  Exponent : string[2],  (exponent of S for coeff input form display)
  Specification: string[5];  (Input_Handler calling string)
  Strg : string[2];  (Input_Handler calling string)
  Filename : string[2];  (file name for storing block data)
  ReadFile : file of Blocks;  (file for reading in block filenames)
  NZeros_old,
  NFoles_old : integer;
  test : real;  (temporary variable to hold results of 'val'
  (until conversion is validated))

(******************************************************************************)
(** Begin processing Factored form Input-Internal routine **)  
(******************************************************************************)

Procedure Input_Factored (zeros_or_Poles: String; Factors: integer; var RealPart,
  ImagPart; PolyArray);
var
  i,j : integer;
  test : real;  (holds 'val' results until conversion is validated)
begin
ClrScr; TextColor(White);   {write screen titles}
Center("Block Transfer Function Input",1,1,80);
HighVideo;
writeln writeln(s);TextColor(green)
if Zeros_or_Poles = 'ZEROS' then
  writeln('NUMERATOR Transfer Function Input — FACTORED Form')
else
  writeln('DENOMINATOR Transfer Function Input — FACTORED Form');
HighVideo;
for j:=-1 to NFactors do   {type prompt strings}
begin
  writeln(' s = +j'};
  writeln;
end;
str(NFactors*2+10:2,strg)   {select proper input handler}
if Zeros_or_Poles = 'ZEROS' then
  strg = 'NUMERATOR Transfer Function Input — FACTORED Form'
else
  strg = 'DENOMINATOR Transfer Function Input — FACTORED Form';
HighVideo;
for j:=-1 to NFactors do   {call the input handler}
begin
  if Change then specification := concat('Cir,strg)   {builds string to call}
  else specification := concat('NUMERATOR Transfer Function Input — FACTORED Form');
  Input_handler(specification, escape);   {call the input handler}
for j:=1 to NFactors do   {compute the zero values from}
begin
  val(filvar[2*j+9], test, code);
  if code = 0 then RealPart[j] := test;  {val conversion successful if code}
  if code = 0 then ImagPart[j] := test;
end;
procedure Input_Coeff(zeros_or_poles:Str5; NCoeff:integer);
var  Coeff: Polyarray;

var  i,j, (counters)
    NCoeff_old : integer; (holds old poly order if changing order)
    test : real; (holds "val" results until validated)

begin
P121 := '0408N01021-000101'; (Input-Handler descriptors for coeff)
P122 := '1808N01022-000101'; (form input)
P123 := '3208N01023-000101';
P124 := '4608N01024-000101';
P125 := '0408N01025-000101';
P126 := '1808N01026-000101';
P127 := '3208N01027-000101';
P128 := '4608N01028-000101';
P129 := '0408N01029-000101';
P130 := '1810N01030-000101';
NCoeff_old := NCoeff;
ClrScr; TextColor(White); (print screen titles)
Center('Block Transfer Function Input - COEFFICIENT Form', 1, 2, 80);
writeln; writeln;
TextColor(limegreen);
if zeros_or_poles = 'ZEROS' then
  writeln('NUMERATOR Transfer Function Input - COEFFICIENT Form');
else
  writeln('DENOMINATOR Transfer Function Input - COEFFICIENT Form');
HighVideo;

if NCoeff < NCoeff_old then
begin
  for j:=1 to NCoeff_old - NCoeff do
    for i:=NCoeff_old + 1 downto 1 do
      F[i] := F[i+1]; (shift the coeff down to )
  (location proper exponent)
end;

if NCoeff > NCoeff_old then
begin
  for j:=1 to NCoeff_old - NCoeff do
    for i:=1 to NCoeff + 1 do
      F[i] := F[i+1]; (shifts the coeff up to proper exp)
for j=NCoeff+1 downto 1 do
begin
  j:=NCoeff+1 - i;
  PosCounter := (i and 4) + 1; HorizPos := PosCounter * 14;
  if PosCounter = 1 then VertPos := VertPos + 2;
if i<> 1 then (prompts for coeff input)
begin
  msg('s v',HorizPos,VertPos);
  strl(i-2,Exponent);
  msg(Exponent,HorizPos+i,VertPos-1);
end;
end;
end;
end;
E
end;

strl(20+NCoeff+1),strg); (sets up and calls input handler)
if Change then specification := concat('Z1',strg)
else specification := concat('NZ1',strg);

Input_handler(specification,escape);
for j=NCoeff+1 downto 1 do
begin
val(Filvar[NCoeff+22-j],test,code);
if code = 0 then Coeff[j]:=test;
end;
end;

 Begin the Block Input Routine. This allows input of a generic block GENERATED
from either input or change routines. GENERATED

begin (procedure Blockinput)
with Block_Description do    (Block_Description is Blocks-type variable)
begin
Clrscr;
if Change then temp2 := 'K' (if Change option always from keyboard)
else
begin
  gotany(1,71); textcolor(Bred); writeln('Block No. ',BlockIndex);
  textcolor(yellow);
  msg('Will you input this block from the Keyboard (K) or Diskfile (D) ? ','1,10);
repeat
  Input('K','K',73,102,true,Fl,F10); (validate for 'K' or 'D')
  temp2:=copy(answer,1,1);
  if not(temp2 in ['K','D']) then beep(500,150);
  until temp2 in ['K','D'];
end;
end;
end;

if temp2 = 'D' then (if input from diskfile, then prompt for drive)
begin
  and display directory of blocks available)
ClsScr; HighVideo;
Msg('Current Data Drive is . Press (esc) to change it***',1,11);
repeat
input('A',copyDrive,1,11,36,11,2,true,F1,F10);
ch:= copyanswer,1,11;
if ch in ['A', 'B', 'C', 'D']) then beep(350,150);
until ch in ['A', 'B', 'C', 'D'])
Drive := concat(ch,'!');
Highvideo;
Directory(drive,extension,filename,readerror); (call Directory display)
if not(readerror) then

begin
Assign(Readfile,filename);      (open the selected file passed back)
Reset(Readfile);                (from Directory in variable Filename)
clrscr;
msg('Is the block read from disk in the Forward (F) or Feedback (B) path?',1,10);
repeat
Input('F',75,1,10,1,true,F1,F10);
if not(ch in ['F', 'B']) then beep(350,150);
until ch in ['F', 'B']
if ch = 'B' then Feedback := true
else Feedback := false;
end
else
begin
delay(1500);  (wait until directory error is displayed)
window(1,1,80,25);  (expand out of directory window)
clrscr;
end;
end;
else
begin
ClrScr; TextColor(White);
Center('Block input Segment',1,2,80); HighVideo;
writeln; writelnls); writeln; TextColor(Blue); 
writeln('Block ', BlockIndex,' '); HighVideo;

P[1] := '601002017000100';  (Fbk/Fwd path block)
P[2] := '6011NO020200100';  (Nzeros prompt)
P[3] := '6012NOG0205000100';  (NPoles prompt)
P[4] := '6011SW099990000100';  (Block gain prompt)
P[5] := '60182020G1000011';  (Fact. or Coeff. form input)
msg('Is block in Forward (F) or Feedback (B) Path?',1,7); 
msg('What is the order of the Numerator?',10,11);
msg('What is the order of the Denominator?',10,13);  (# poles)
msg('What is the block gain constant?',10,15);
if not(Change) then
begin
if Change then Input_Handler('CD0104',escape) else Input_Handler('MO105',escape);

if copyFilvar(01,1,1) = 'B' then Feedback:= true else Feedback:= false;

val(Filvar[01],NZeros,code); (convert NZeros to number)
val(Filvar[03],NPoles,code); (convert NPoles to number)
val(Filvar[04],test ,code); (convert gain to number/validate)
if code = 0 then K:=test;

if copyFilvar(05,1,1) = 'F' then Factored:= true else Factored:= false;

if Factored then     (set up to call Input_factored routine)

begin

if Change then   (if Changing an old entry, then put all old values)
    (into associated Filvar[j] to be displayed)
for j:=1 to NZeros do

    begin

    str(RealPartZero[j]:10:2,filvar[2tj+9]);
    str(ImagPartZero(j]:10:2,filvar[2tj+10]);

    end;

if NZeros = 0 then NumCoeff[l]:=1.0 (if order zero, then assign )
else
 END; (otherwise prept for input)

    Input_Factored('ZEROS',NZeros,RealPartZero,ImagPartZero);

    Expand_Poly(RealPartZero,ImagPartZero,NumCoeff,NZeros);

    end;

if Change then (repeat above process for Poles/denom input)

    for j:=1 to NPoles do

    begin

    str(RealPartPole(j]:10:2,filvar[2k+j+9]);
    str(ImagPartPole(j]:10:2,filvar[2k+j+10]);

    end;

if NPoles = 0 then DenCoeff[l]:=1.0 else

begin

104
Input_Factored('POLES',NPoles,RealPartPole,ImagPartPole);
Expand_Poly(RealPartPole,ImagPartPole,
DenCoeff,NPoles);

LeadNumCoeff:= 1.0; LeadDenCoeff:= 1.0; (expanding factors will always
create unity leading coefficients. These variables are
used in Coeff_input to assist in normalizing/denormalizing)
(coefficients for display/change and computation)

else

(*-----------Begin Coefficient for input-------------*)

begin

if Change then (if changing entries, restore normalized coeffs to)
(orig. form - doesn't confuse user. Then restore into)
(proper FiltVarH for display)
for j:=NZeros+1 downto 1 do

begin

NumCoeff[j] := NumCoeff[j] * LeadNumCoeff;
str(NumCoeff[j]:10,filvar[NZeros+22-j]);

end;

Input_Coeff(Poles,NZeros,NumCoeff); (call Input_Coeff routine)

LeadNumCoeff:= NumCoeff[NZeros + 1]; (normalize polynomial)
for j:= 1 to NZeros + 1 do

NumCoeff[j] := NumCoeff[j] / LeadNumCoeff;
K := K * LeadNumCoeff; (milt K by leading numerator coeff to keep)
(polynomial's numeric integrity)

RootFinder(NZeros,NumCoeff,RealPartPole,ImagPartPole,0.0,0.0);

if Change then (repeat above process for denominator)
for j:=NPoles+1 downto 1 do

begin

str(DenCoeff[j]:10,filvar[NPoles+22-j]);

end;

Input_Coeff('POLES',NPoles,DenCoeff);

LeadDenCoeff:= DenCoeff[NPoles + 1]; (normalize polynomial)
for j:= 1 to NPoles + 1 do

K := K / LeadDenCoeff; (divide K by lead Denom. coeff to keep)
(polynomial's numeric integrity)

RootFinder(NPoles,DenCoeff,RealPartPole,ImagPartPole,0.0,0.0);

end (else)

end; (if)

end; (with)

end; (procedure)
Procedure Trans_Funclion_1nput;

var
  j, i, code : integer;
  InputErr : boolean;
  LoopFeedBack : boolean;

begin
  Changes := false;
  Clrcscr;
  TextColor(White);
  Center('Transfer Function Input Routine***',1,2,80);
  HighVideo;
  Fillchar(5,100,1205); {construct horizontal line on screen}
  writeln;'begin user prompts'}
  msg('How many blocks in this loop?',10,6);
  Input('"N",1,45,A,2,true,F1,F10);
  val(Answer,NBlocks,Code);
  LoopFeedback := false;
  for i := 1 to NBlocks do
    begin
      repeat
        Block_input(Block[i],i,InputErr); {call Block_input until no error}
      until rjt(inputErr); {and all loop blocks have been made}
      if Block[i].FeedBack then LoopFeedback := true;
    Clrcscr;
    end;
  if LoopFeedback then
    begin
      msg('Is the Loop Feedback Negative (N) or Positive (P)?',1,10);
      repeat  {validate input answer}
        Input('"N","P","S",10,2,true,F1,F10);
        chrCopy(answer,1,1);
        if not (ch in ['N','P']) then beep(500,150);
        until ch in ['N','P'];
       if ch = 'N' then NegFeedback := true
       else NegFeedback := false;
    end;
  First_run := false;
  Hal;e_Geq;    {once loop finished, compute equivalent single block}
  end;
end;
The following procedure allows the user to change the contents of a block in the existing loop structure. It will not only allow changes to existing coefficients or factors of the user polynomial, but changes to the order of the polynomial as well.

Procedure Change_Block;

var
i,j,code,Block_num : integer;
inputerr : boolean;

begin
if NBlocks > 0 then
begin
  Changes := true;
  Clrscr;
  TextColor(White);
  Center('Transfer Function Change Routine',1,2,80);
  HighVideo;
  writeln;'Transfer Function Change Routine',(prompt for block to change)
  Input('Which Block do you wish to change?',10,6);
  repeat
    Input('N','r,43,6,2,true,Fl,F10); (inputs and validates block within range)
    val(FlnsHer,Block_num,Code);
    if not(Block_num in [1..NBlocks]) then beep(350,150);
  until Block_num in [1..NBlocks];

  (Initialize the Input handler holding variables to those of the block to be changed)
  with Block[Block_num] do
  begin
    if Feedback then Filvar[01] := 'B'
    else Filvar[01] := 'F';
    str(Wlzeros,Filvar[02]);
    str(Wlpoles,Filvar[03]);
    K := K * LeadDenCoeff(LeadNumCoeff); (de-normalize poly for display)
    str(K:10:2, Filvar[04]);
    if Factored then Filvar[05] := 'F'
    else Filvar[05] := 'C';
  end;
  repeat
    Block_Input(Block_num,Block_num,inputerr); (Call Block_Input to change)
    until not(inputerr);
  if Block[Block_num].Feedback then
  begin
    end;
    repeat
      Block_Input(Block_num,Block_num,inputerr); (Call Block_Input to change)
      until not(inputerr);
    if Block[Block_num].Feedback then
    begin
      end;
**Procedure Add_Block;**

var
  InputErr : boolean;

begin
  InputErr := false;
  Change := false;
  NBlocks := NBlocks + 1; (pick next block)
  repeat
    Block_Input(Block(NBlocks), NBlocks, InputErr); (input next block)
    until not(InputErr);
    if Block(NBlocks).Feedback then
      (***************************************************************************)
      begin
        msg 'Is the Loop FeedBack Negative (N) or Positive (P)?';
        repeat (validate input answer)
          if NegFeedback then ch := 'N';
          Input('A', ch, 35, 10, 2, true, F1, F10);
          ch := copyAnswer, 1, 1);
          if not(ch in ['N', 'P']) then beep(350, 150);
          until ch in ['N', 'P'];
        if ch = 'N' then NegFeedback := true
          else NegFeedback := false;
      end;
      Make_Beq; (compute new equivalent block)
      end;
      (***************************************************************************)
The following procedure deletes a block from the current loop:

```pascal
Procedure Delete_Block;
var
code,BlockNumber,i : integer;
Begin
  if NBlocks <> 0 then
    Begin
      clrscr; HighVideo;
      msg('Which block do you wish to delete from this loop? [0 to cancel]',1,10);
      input('N',',75,10,1,true,F1,F10);
      val(answer,BlockNumber,code);
      if BlockNumber in [1..NBlocks] then
        Begin
          NBloclS:=NBlocks-1;
          for i:=BlockNumber to NBlocks do
            Block[i]:=Block[i+1];
          Make_Geq;  (compute new equivalent block after deletion)
        End;
    End;
End;
```

The following procedure allows the user to save the current loop to a disk file:

```pascal
Procedure Save_Block;
Blockfile : file of Blocks;
filename : str20;
Prn : integer;
Begin
  clrscr; HighVideo; (let user change drive if necessary)
  Msg('***Current Data Drive is . Press (esc) to change it!***',1,11);
  repeat
    input('A',copy(drive,l,l),36,ll|2|true
    ch:= copy(answer,!,1);
    if notlch in ['A','B','C','D']) then beep(350,150);
    until ch in ['A','B','C','D'];
    Drive := concat(ch,':');
    clrscr;
    msg('Diskfile name to store the current blocks?',1,10);
    writeln; writeln; writeln('***Ensure that your DATA disk is in drive ' ,Drive,'***');
    input('A',',45,10,8,true,F1,F10);
    filename:= concat(Drive,copy(answer, 1,0)/.OLD;
    Assign(Blockfile,filename); (Open file and save _eq block first, then loop)
    Rewrite(Blockfile); (blocks)
  ```
627  "Write(0,B_eq); {store Geq at pointer position 0 in the file)
628  {This allows block input routine to load)
629  {the equivalent block as a single block)
630  for pnr:=1 to NBlocks do {store other block information sequentially)
631  begin
632  Bkehr(0,B_eq); {store other block information sequentially)
633  seek(0,B_eq);
634  write(Blkfile,Block(pnr));
635  end;
636  TextColor(Green); {alert user that loop has been saved)
637  Center('The current problem loop has been stored',1,20,80);
638  beep(350,150);HighVideo;
639  delay(200);
640  doselBlockfile); {close the file)
641  end;

642  {The following procedure allows the user to get the current)
643  {loop to a disk file.
644  Procedure Retrieve_Problem;
645  var
646  Readfile      : file of Blocks;
647  filename      ; str20;
648  readerror     : boolean;
649  begin
650 Clrscr; HighVideo; {allow change of disk drive if desired)
651  Msg(****Current Data Drive is . Press <esc> to change it****',10,11);
652  repeat
653  input('A',copydrive,1,1),36,11,2,true,F1,F10);
654  ch:= copyanswer,1,1);
655  if not(ch in ['A','B','C','D']) then beep(350,150);
656  until ch in ['A','B','C','D'];
657  Drive := concat(ch,':');
658  Directory(Drive,extension,filename,readerror);
659  if not(readerror) then
660  begin
661  Assign(Readfile,filename); {open the file and retrieve contents)
662  Reset(Readfile);
663  Read(Readfile,B_eq);
664  NBlocks := 0;
665  while not EOF(Readfile) do
666  begin
667  NBlocks:= NBlocks+1;
668  Seek(Readfile,NBlocks);
669  NBlocks := NBlocks;
670  Read(Readfile,B_eq);
671  end;
672  {call Directory to display eligible files)
673  Directory(Drive,extension,filename,readerror);
674  if not(readerror) then
675  begin
676  Assign(Readfile,filename); {open the file and retrieve contents)
677  Reset(Readfile);
678  Read(Readfile,B_eq);
679  NBlocks := 0;
680  while not EOF(Readfile) do
681  begin
682  NBlocks:= NBlocks+1;
683  Seek(Readfile,NBlocks);
procedure MenuItems(pick:char;description:string;color:integer);
716 -begin (displays menu items in colors desired)
717 textcolor(color);
718 write('\'Tab\', ' '); textcolor(White); writeln(pick);
719 textcolor(color); writeln(' ',description);
720 -end;
721
722 B---------------------------------------------------------------------
723 -begin (PROCEDURE INPUTENU)
724 ClrScr; TextColor(White); Finished := false;
725 Center(' *** INPUT/CHANGE MENU *** ',1,4,80);
726 for i:= 1 to 4 do writeln(' ');
727 Tab:= 25;
728 MenuItems('I', 'Input Block Transfer Function',green);
729 MenuItems('C', 'Change Block in Current Loop',green);
730 writeln;
731 MenuItems('A', 'Add a Block to Current Loop',green);
732 MenuItems('D', 'Delete a Block from Current Loop',green);
733 writeln;
734 MenuItems('S', 'Save Current Loop to Disk File',green);
735 MenuItems('R', 'Retrieve Problem from Disk File',green);
736 writeln;
737 MenuItems('H', 'Help',blue);
738 MenuItems('O', 'Exit to Main Menu',lightmagenta);
739 TextColor(Green); 
740 Read(20,2,85,22,6); writeln(' ');
741 Set_Cap_Num('C', ' ', ' '); Set_Cap_Num;
742 TextColor(White); Center('Press your Selection',21,21,38); LowVideo;
743 if NBlocks < 0 then Okchoices := ['I','C','O','S','R','H','W'];
744 else Okchoices := ['I','C','O','S','R','H','W'];
repeat (wait for user to input a keypress, check for validity, branch)

option; if not (Ch in O|choices) then

begin

Beep(150,150); TextColor(White);

if NBlocks <= 0 then

begin

msg('WARNING: First INPUT the block description!',1,25);

end;

end;

until Ch in OKchoices;

begin

'i' : Trans_function_Input;

'C' : Change_Block;

'A' : Add_Block;

'D' : Delete_Block;

'S' : Save_Block;

'R' : Retrieve_Problem;

'H' : InputMenuHelp;

'F' : Finished = true;

end;

until Ch in OKchoices;

begin

'i' : Trans_function_Input;

'C' : Change_Block;

'A' : Add_Block;

'D' : Delete_Block;

'S' : Save_Block;

'R' : Retrieve_Problem;

'H' : InputMenuHelp;

'F' : Finished = true;

end;

begin (Main Program Input) (due to large size of input routine, compiled as)

(A CHAIN program called from CAD.COM. When )

(repeat is complete, CAD.COM is re-executed. )

Driver = 'A';

InputMenu;

Until finished;

AssignCadFile('Cad.com');

Execute(CadFile);

end.
Program Utility Menu; (displays utility menu on screen and branches to
user-selected program)

1 typedef.sys
2 (1) ut-m011.inc
3 (1) ut-m002.inc
4 (1) ut-m003.inc
5 (1) roots.inc
6 (1) expand.inc
7 (1) ShowPoly.inc
8 (1) ShowFact.inc
9 (1) UserPoly.inc
10 (1) UtilHelp.inc

var
through: boolean;

procedure ShowMenu;
var
1,Tab : integer;
3 O!choices : set of char;
4 Input_routine : file;

begin

procedure MenuItem(pick:char;description:str80;color:integer);
(Takes writing colors easy)

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

48

49

50

51

52

53

54

55

56

Textcolor(color);
write('|':tab,''); textcolor(White); write(pick);
Textcolor(color); writeln(' ',description);

end;

begin (ShowMenu)
ClrScr; TextColor(White); Through := false;
Center(' *** UTILITIES MENU *** ',1,4,80);
for I:= 1 to 4 do writeln('');
Tab:= 2;

Menuitem('F', 'Show Current Loop Blocks (Factored)',green);
Menuitem('P', 'Show Current Loop Blocks (Polynomial)',green);
write;
Menuitem('U', 'Factor a User-input Polynomial',green);
write;
Menuitem('H', 'Help', lightblue);
write;
Menuitem('E', 'Exit To Main Menu', lightmagenta);
TextColor(Yellow);
Box(20,2,65,18,0:writeIn='');
Set_Cap_Num('C', ' ', ' '); Set_Cap_Num;
TextColor(White); Center(' Press Your Selection ',21,17,30); LowVideo;
sets legal choices if no blocks have been entered)
if NB0cks <= 0 then OKchoices := ['H', 'U', 'U']

113
else Okchoices := ['F', 'P', 'O', 'U', 'H'];

Option;
if not (Ch in Okchoices) then
begin
  begin
    msg('WARNING: First INPUT the block description!', 1, 20);
    end;
  end;
  until Ch in Okchoices;

case Ch of

  'F': ShowFactored_Roots;
  'P': ShowPoly;
  'O': User_Polynomial_Roots;
  'U': UtilitiesHelp;
  'H': Begin
    through := true;
    assign(cadfile, 'cad.com');
    execute(cadfile);
  end;
end;

(*--------------------------------------------------------------------*)
(* Program Starts Execution *)
(*--------------------------------------------------------------------*)

begin
  repeat
    ShowMenu;
    until through;
end.
Program Frequency_Response;

{# typedef.sys (graphics routines)
{# graphiz.sys
{# kernel.sys
{# windows.sys
{# polygon.hgh
{# axis.hgh

{# UT-MXDO1.INC  (I/O routines)
{# UT-MXDO2.INC
{# UT-MXDO3.INC

{# GraphMenu.inc (graph options menu)
{# PlotBode.inc (Bode plotting routine)
{# PlotNyq-inc (Nyquist plotting routine)
{# Roots-inc (RootFinder routine)

{# Bode.inc (Bode routine)

begin

BODE;   \(call the Bode calculation routine\)

 AssignCADfile,'CAD.COM';   \(re-execute the main menu routines\)

 ExecuteCADfile();

end.
File: BODE.INC

procedure Bode;

var

Code, l, Count, NumDecades, StartDecade, EndDecade : integer;
HI, Mo, N, DeltaN : real;
WReal, Wimag, Areal, Aimag : real;
PlotArray1, PlotArray2, MagPhaseArray : PlotArray;
HI, Mo, N, Phase : real;
Temp1, Temp2 : real;
Close : real;
Temp : char;
OpenLoop, Nyquist, BigPlot : boolean;

function Log(X:real): real;  (computes the base-10 logarithm of X)
BEGIN
IF X=0 then Log:=0 else
Log := Ln(X)/Ln(10);
END;

function Expont(Y,X:real): real;  (computes Y raised to X power)
BEGIN
Expont := Exp(X * (Ln(Y)))
END;

begin
ClrScr; TextColor{White);
Center("***Bode/Nyquist Plotting Routine***",1,2,80);
writeln;
BigPlot := false;
TextColor{Green};
Msg('Bode (B) or Nyquist (N) Plot?';5,5);
if temp = 'B' then BigPlot := true
else BigPlot := false;
end;
HsgCOpCT (0) or Closed (0 Loop FloL';?); repeat
input('A',",45,7,2,tru?,Fl,F10); (sets flag OpenLoop if)
temp := copy(answer,1,1); (user selects the open)
if not(temp in ['0','C']) then beep(150,150); (loop option for plot ) untill temp in ['0','C'];
if temp = '0' then OpenLoop := true
else OpenLoop := false;
if not(GigPic) then
begin
Msg('What is the first frequency to be plotted?',5,9);
Msg('e.g. .01, .001, 100, etc.') ,10,10);
Input('N',",45,10,8,true,F1,F10);
Val(\text{answer},No,code); (\text{No is the first plotted freq})
end
if lesp = '0' then OpenLoop := true
else OpenLoop := false;
if not(BigPic) then
begin
Msg('What is the first frequency to be plotted?',5,9);
Msg('e.g. .01, .001, 100, etc.') ,10,10);
Input('N',",45,10,8,true,F1,F10);
Val(\text{answer},No,code);
end

with G_eq do
begin
(Close closed-loop unity feedback) G equivalent)
called CLGEQ -- which is also a block type record)

CLEEQ.K := K; (closed loop K same as open loop K)
CLEEQ.NZeros:=Nzeros; (as are the zeros of the function )
for i:=1 to maxorder do CLGEQ.DenCoeff[i] := 0.0; (initialize)
for i:=1 to Nzeros do
begin
CLEEQ.RealPartZeros[i] := RealPartZeros[i]; (same zeros)
CLEEQ.ImagPartZeros[i] := ImagPartZeros[i];
end;
for i:=1 to Nzeros + 1 do
begin
CLEEQ.DenCoeff[i] := NumCoeff[i] * K; (C.L. denominator equals the )
CLEEQ.NumCoeff[i] := NumCoeff[i]; (sum of open loop denominator)
end;
for i:=1 to NPoles + i do CLGEQ.DenCoeff[i] := CLGEQ.DenCoeff[i] +
DenCoeff[i];
end;
if NPoles > Nzeros then CLGEQ.NPoles:=NPoles (NPoles should always be)
else CLGED.NPOLES:=Nzeros; (greater, but to be safe)

( compute new denominator roots)

rootfinder(CLGED.NPOLES,CLGED.DenCoeff,CLGED.RealPartPole,)

CLGED.ImagPartPole,0,0);

end; (with)

StartDecade := trunc(Log(Fo));

EndDecade := StartDecade + NumberDecades; (log numbers, also figure step)

Wf := Fo * Expon(10,0,NumberDecades);

delaw := Expon(NH/xhi,0,0125);

Wi := Fo;

for Count := 1 to 01 do

if OpenLoop then

Begin

if G_eq do

Begin

{do 100 iterations...arbitrary })

{compute linear scale to plot })

{compute bode numbers with G_eq if openloop)

{and later with CLGD if closed loop })

Begin

ZHagn:=1.0;ZFha5e!=0,0i;ZPha5e:=1.0;PPhase:=0.0;   {initialize)

for I := 1 to NZeros do    {compute «agn and phase of zeros for freq step)

begin

ZPha5e:=ZPha5e+SqrtlSqrlRealPartZero(l)+SqrtlWmaqPartZero(l)];

if RealPartZero(l) = 0.0 then ZPha5e:=ZPha5e+pi/2.0 else

ZPha5e:=ZPha5e+ArcTan((Hi-lmagPartZero(l))/(-RealPartZero(l)))]

—-^end;

—If Nyquist then Begin

Phase := Frac(ZPhase - FFhasB)/12*(pi)) » 12*pi);

if (BigPic) and (TempX > 100) then TempX := 100;

if (BigPic) and (TempY > 100) then TempY := 100;

if Phase>0 then Phase:=Phase+2*pi;

if (Phase<pi/2) and (Phase>(3*pi/2)) then

Temp:= -Temp;

if (Phase>pi/2) and (Phase<(2*pi)) then TempY := -TempY;

MagPhaseArray[Count,2] := Phase;

MagPhaseArray[Count,1] := K2Magn/PMagn;

PlotArray[Count,1] := TempX;

PlotArray[Count,2] := -TempY;

end

else

Begin

PlotArray[Count,1] := Log(Ni);   (fill plotting matrix)

end

118
PlotArray1[Count,2] := 20*Log[N*(Min/Magn)];

PlotArray[Count,1] := Log[W];  
PlotArray[Count,2] := (180/pi)*[Phase-FPhase];

if PlotArray[Count,2] > 0 then
    PlotArray[Count,2] := PlotArray[Count,2] - 360;

else
    for I := 1 to NZeros do
        if RealPartZero[I] = 0.0 then ZPhase := ZPhase + pi/2.0 else
            ZPhase := ZPhase + Arctan((Hi - ImagPartZero[I])/(RealPartZero[I]));

        if NewPhase[I] = 0.0 then NewPhase[I] := NewPhase[I] + 2*pi else

        ZHagn := ZHagn * Sqrt(Sqr(RealPartZero[I]) + Sqr(Wi - ImagPartZero[I]));
        if NewPhase[I] = 0.0 then NewPhase[I] := NewPhase[I] + 2*pi else

        for I := 1 to NPoles do
            if RealPartPole[I] = 0.0 then NewPhase[I] := NewPhase[I] + pi/2.0 else
                NewPhase[I] := NewPhase[I] - Arctan((Hi - ImagPartPole[I])/(RealPartPole[I]));

            if NewPhase[I] = 0.0 then NewPhase[I] := NewPhase[I] + 2*pi else

            TempX := abs(Hi) * ZHagn/PHagn * cos(NewPhase[I])
            TempY := abs(Hi) * ZHagn/PHagn * sin(NewPhase[I]);

            if (TempX > 100) then TempX := 100;
            if (TempY > 100) then TempY := 100;

            if NewPhase[I] > (3*pi/2) then TempX := -TempX;
            if NewPhase[I] < (pi/2) then TempY := -TempY;

        TempK := abs(Hi) * ZHagn/PHagn * cos(NewPhase[I])
        TempY := abs(Hi) * ZHagn/PHagn * sin(NewPhase[I]);

        if (TempX > 100) then TempX := 100;
        if (TempY > 100) then TempY := 100;

        if NewPhase[I] > (3*pi/2) then TempX := -TempX;
        if NewPhase[I] < (pi/2) then TempY := -TempY;

        PlotArray[Count,1] := TempX;
        PlotArray[Count,2] := TempY;

    end

else
    for I := 1 to NPoles do
        if RealPartPole[I] = 0.0 then NewPhase[I] := NewPhase[I] + pi/2.0 else
            NewPhase[I] := NewPhase[I] - Arctan((Hi - ImagPartPole[I])/(RealPartPole[I]));

        if NewPhase[I] = 0.0 then NewPhase[I] := NewPhase[I] + 2*pi else

        TempX := abs(Hi) * ZHagn/PHagn * cos(NewPhase[I])
        TempY := abs(Hi) * ZHagn/PHagn * sin(NewPhase[I]);

        if (TempX > 100) then TempX := 100;
        if (TempY > 100) then TempY := 100;

        if NewPhase[I] > (3*pi/2) then TempX := -TempX;
        if NewPhase[I] < (pi/2) then TempY := -TempY;

        PlotArray[Count,1] := TempX;
        PlotArray[Count,2] := TempY;

    end

end

if Nyquist then
    Phase := Frac((ZPhase - FFhase)/(2*pi)) * (2*pi);
    TempX := abs(1*(ZMagn*ZMagn) * cos(Phase));
    TempY := abs(1*(ZMagn*ZMagn) * sin(Phase));

    if (TempX > 100) then TempX := 100;
    if (TempY > 100) then TempY := 100;

    if NewPhase[I] > (3*pi/2) then TempX := -TempX;
    if NewPhase[I] < (pi/2) then TempY := -TempY;

    PlotArray[Count,1] := TempX;
    PlotArray[Count,2] := TempY;

else
    PlotArray[Count,1] := Log[H];  
    PlotArray[Count,2] := 20*Log[1*(ZMagn*ZMagn)];

    if PlotArray[Count,2] > 0 then
        PlotArray[Count,2] := PlotArray[Count,2] - 360;

    end

end
Hi := Hi + DeltaHi;

if Nyquist then
    Plot_Nyquist(StartDecade, EndDecade, NumberDecades, PlotArray1, MagPhaseArray, BigPic, OpenLoop);
else
    Plot_Bode(StartDecade, EndDecade, NumberDecades, PlotArray1, PlotArray2, OpenLoop);

end;
procedure PlotBode(StartDecade, EndDecade, NumberOfDecades: Integer;
PlotArray1, PlotArray2: PlotArray; OpenLoop: Boolean);

(* ******************************************************)
(** Procedure to Plot the LIN-LOG chart for the Bode diagram. **)
(* ******************************************************)

const
MagnitudeArray: array[1..12] of char = ('M','A','B','N','I','T','U','D','E','E','g','B');
PhaseArray: array[1..12] of char = ('P','H','A','S','E','D','E','E','E','E','E','E','E');
FreqArray: string[19] = 'FREQUENCY (rad/sec)';

var i,j,n: integer;
ch: char;
x1,x2: integer;
Delta: real;
MagLabel: string[3];
PhaLabel: string[4];
DecLabel: string[3];
Title1,
Title2: string[10];
NumGraph: Boolean;
x: real;
quit: Boolean;
list: text;

function Log(X:real):real;
begin
if X = 0 then Log := 0 else
Log := Ln(X)/Ln(10);
end;

function Expon(Y,K:real):real;   (computes Y raised to K power)
begin
Expon := exp(K*(ln(Y)));
end;

Procedure PrintGraphData; {prints numbers to a file or printer}
begin
LeaveGraphic;
Clrscr;
Center('*** TURN ON PRINTER AND ALIGN PAPER ***',1,10,80);
TextColor(green);
writeln('Press <P> to continue, <F> to list to file, <Q> to quit print',1,13);
TextColor(ltblue);
msg('File option prints numbers to a file named "BODE.NUNIT",1,17);
msg('on the current drive. Browse this file off-line using the DOS "type" command.','1,18);
repeat
Readln(ch);
if (ch = 'C') or (ch = 'c') or (ch = 'P') or (ch = 'p') then
begin
if (ch = 'F') or (ch = 'f') then
begin
121
begin
assign(list,'fode,M,N');(assign file)

rewritelst);

end
else

begin
assignlist,'LIST';(otherwise assign printer)

rewritelist);

end;

end;

TitleI:='w (rad)

Gain (db)

Phase (deg');

Title2:='Hriteln(list,Titlei);

writeln(list,Title2);

for i:= 1 to 81 do

begin

writeln(list,'EXI',N;',N:113 ',

Gain Idb)

Phase (deg'));

if i= 50 then

begin

writeln(chr(12));

end;

end;

end;

until ch in ['F', 'I', 'G', 'P', 'p']

EnterGraphic; (when finished printing, go back to graphics mode and)

SnapScreen; (display graph)

close(list);

end;

begin

initgraphic;

( set-up windows for display)

Definewindow(1,0,0,MaxGl',MaxGl);

Definewindon(2,5,15,MaxGl5,MaxGl5); 11

Definewindon(3,5,15,MaxGl5,MaxGl5); 11

Definewindon(1,0,100,100);

Definewindon(2,StartDecade,60,EndDecade,-60);

Definewindon(3,StartDecade,0,EndDecade,-360);

SelectWorld(l);

SelectWindow(l);

SetBackground(lO); 11

SelectWorld(2);

SelectWorld(2);

DrawBorder;

SetLineStyle(l); 11

For i:=1 to 5 do (draw horizontal graph lines)

DragLine(StartDecade,-60*(20*I),EndDecade,-60*(20*I)); 11

For j:=0 to NumberDecades-1 do (draw vertical logarithmic graph lines)

122
For I:= 1 to 10 do

Begin

Delta:=StartDecade + (Log(I) + J);

Drawline(Delta,-60,Delta,60);

End;

SelectWindow(1);     \( y\)-axis titles

For I:= 1 to 12 do

Begin

DrawText(i,55+6*I,MapArray[I]);

DrawText(i,60+6*I,PhaseArray[I]);

End;

DrawText(250,195,FreqArray);    \( x\)-axis title

For I:= 0 to 6 do

Begin

Str(60-20*I,3,Label);    \( y\)-axis scale label

Str(0-60*I,4,Label);

Str(360-20*I,3,Label);

Str(360-60*I,4,Label);

End;

For I:= 0 to NumberDecades do

Begin

Str(Trunc(StartDecade)+13,DecLabel);

DrawText(36+1570 div NumberDecades) *I,185,1,DecLabel);

End;

SetLineStyle(0);

SelectWindow(2);

DrawPolygon(PlotArray1,-81,0,1,0);   \( x\)-axis (plot the magnitude)

SelectWindow(3);

DrawPolygon(PlotArray2,-81,0,1,0);   \( y\)-axis (plot the phase)

CopyScreen;    \( x\)-axis (save screen to memory)

Repeat until keypressed;

quit := false;

Repeat

\( call \) \ for \ graph \ options \ menu

if OpenLoop then Graph_Menu(Open Loop Bode Plot, DumpGraph, quit)

else Graph_Menu(Closed Loop Bode, DumpGraph, quit);

If DumpGraph then PrintGraphData;    \( x\)-axis (dump numbers if desired)

Until quit;

LeaveGraphic;    \( y\)-axis (leave graphics mode)

End;
file: plotincs.inc

cross-reference & block listing

date: 9/10/b6

1 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
2 (** Plot_Nyquist is a routine to draw the Nyquist plot from the data **)
3 (** generated in the Bode procedure. **) 
4 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
5
6 procedure plot_nyquist(start_decade, end_decade, number_decades: integer;
7     plot_array, mag_phase_array, plot_array;
8     bigpic, open_loop: boolean)
9
10 var
11     xmin, ymin, ymax, xmax: real;
12     x_label, y_label: string[32];
13     title1, title2: string[80];
14     dump_graph, quit: boolean;
15     m: real;
16     xmin, ymin: real;
17     graph_width1, graph_width2: real;
18
19 function expon(y, x: real): real; (computes Y raised to X power)
20 begin
21     expon := exp(x * (ln(y)));
22 end;
23
24 procedure print_graph_data; (dump data used to make graph to printer)
25 begin
26     leavegraphic;
27     clrsr;
28     center('### TURN ON PRINTER AND ALIGN PAPER ###', 1, 10, 80);
29     textcolor(green);
30     msg('press <P> to print, <F> to list to file, <Q> to quit print', 1, 13);
31     textcolor(white);
32     repeat
33     msg('File option prints numbers to a file named "NYQUIST_MASTER", 1, 17);
34     msg('on the current drive. Browse this file off-line using DOS "type" command.', 1, 18);
35
36     read(kbd, ch);
37     if (ch = 'F') or (ch = 'f') or (ch = 'p') or (ch = 'P') then
38         begin
39     end;
40     if (ch = 'Q') then
41         quit := true;
42     else
43         msg('DUMP DATA TO FILE', 1, 11);
44     end;
45     msg('DUMP GRAPH TO PRINT', 1, 11);
46     msg('DUMP GRAPH TO FILE', 1, 11);
47     msg('DUMP GRAPH TO PRINT AND FILE', 1, 11);
48 end;

124
if (ch = 'F') or (ch = 'f') then
   begin
      assignlist, Nyquist.NYM';
   rewrite(list);
   end;
else
   begin
      assignlist, LST';
   rewrite(list);
   end;
Title1:=('w (rad) Magnitude Xplot Yplot');
Title2:=('Phase (rad)');
write(list, Title1); write(list, Title2);
for i := 1 to 81 do
   begin
      w := expont(100, PlotArray(i,1));
      write(list, w, ListArray[i,2], PhaseArray[i,3],
            PlotArray(i,1), PlotArray(i,2));
   end;
until ch in ['F', 'f', 'F', 'f', 'P', 'p'];
EnterGraphic;
sleepscreen;
close(list);

if not(BigPic) then
   begin
      PT51 := '1512NOV05-0101.w'
      PT52 := '1512NOV05-0101.b'
      Clrsr; TextColor(LightBlue);
      Center('*** NYQUIST PLOTTING ROUTINE ***',1,2,80);
      HighVideo;
      FillChars(100,32(5)); S := copy(s,1,80);
      write(list); writeln;
      msg('Viewing Coordinates for Nyquist Plot',1,61);
      TextColor(white);
      msg('Enter X and Y values (graph will span -X to X, -Y to Y)',1,71);
      TextColor(yellow);
      write(list); writeln;
      writeln('X-Maximum: ');
      writeln('Y-Maximum: ');
      125
Input_Handler('W5506',Escape);

while input; writeln;

writeln('Any changes to these parameters? (Y or N):');

Input('Y', ',5,14,2,TRUE,'F',10);

if (answer = 'Y') or (If) then input_Handler('C05506',Escape);

Val(livar(53)),Xex,code);

Val(livar(63)),Yex,code);

Xmin := Xmax; Ymin := Ymax;

end
else

Xmin := -50;    (set default values for "big picture" plot)
Xmax := 50;
Ymin := -50;
Ymax := 50;

end;

INITERPHIC;

(Define world/window and draw X Y axes)

DefineWindow(1,0,0,200,100);

DefineWindow(2,0,0,200,100);

DefineWorld(1,Xmin,Xmin,100,0);

DefineWorld(2,0,0,100,100);

SelectWorld(1); SelectWindow(1); DrawBorder;

DEFINEPLINE;

DrawLine(Xmin,0,Xmax,0);

DrawLine(0,Ymin,0,Ymax);

n := 1;

for i := 1 to 80 do

if (abs(plotarray[i,1]) > Xmax) or (abs(plotarray[i,2]) > Ymax) then

n := n + 1;

if n != 1 then n := n - 1;    (use 1 extra point beyond graph border)

DrawPolygon(plotarray,n-80,0,1,0);    (draw graph on screen)

Xexponent := 0;

graphwidthX := Xmax - Xmin; XminL := Xmin;

while graphwidthX < 1 do

begin

graphwidthX := graphwidthX * 10.0;

XminL := XminL * 10.0;

Xexponent := Xexponent - 1;

end;

Yexponent := 0;

graphwidthY := Ymax - Ymin; YminL := Ymin;

while graphwidthY < 1 do

begin

graphwidthY := graphwidthY * 10.0;

YminL := YminL * 10.0;

Yexponent := Yexponent - 1;

end;

str(Xexponent:5,XexLabel);

str(Yexponent:5,YexLabel);
Xint:=Round((MaxX/10+1)/10);
Yint:=Round((MaxY/10+1)/10);

Select: t'(2);
Select: x'(2);
For i:= 1 to 9 do  (label X,Y axes with scale)
Begin
str(Round(Xint*GraphWidthX/10 + i)/3,XLabel);
str(Round(Yint*GraphWidthY/10 + i)/3,YLabel);
DrawText(W300 + i,47,i,XLabel);
if i <> 5 then
  DrawText((100 + i),1,YLabel)
End;

if Xexponent <> 0 then  (print X exponent on graph)
Begin
  DrawText(W92,43,i,'x 10');
  DrawText(W95,45,i,XexpLabel);
End;

if Yexponent <> 0 then  (print Y exponent on graph)
Begin
  DrawText(W155,95,i,'x 10');
  DrawText(W158,98,i,YexpLabel);
End;

DrawText(W92,55,i,'REAL');  (label real/imag axes)
DrawText(W44,75,i,'IMAG');

Repeat until Keypress;  (Put option menu on screen)
quit:= false;
Repeat
  if OpenLoop then Graph_Menu('Open Loop Nyquist',DumpGraph,quit)
  else Graph_Menu('Closed Loop Nyquist',DumpGraph,quit);
  if DumpGraph then PrintGraphData;
  until quit;
LeaveGraphic;
End;
Procedure Root_Locus(G,eq,blocks);

Label

OnErr;

Var

I,j,code : integer;
PlotPole, PlotZero : PolyArray;
TempPoly,
HoldPoly,
PlotRealPole,
PlotImgPole : PolyArray;
DeltaGain,
StartGain,
EndGain,
Variable_Gain,
Xmin,Xmax,
Ymin,Ymax : Real;
Neg_feedback : boolean;
pg,pg : real;
DumpGraph,quit : boolean;
list : text;
LinCount : integer;

Procedure PrintGraphData;  (dumps rootlocus data to printer)

Begin

LeaveGraphic;

Center('*** TURN ON PRINTER AND ALIGN PAPER ***',1,10,60);

TextColor(green);

'press <P> to Print, <F> to list to a file, <Q> to quit print',1,13);

TextColor(white);

asgn('File option prints numbers to a file named "ROOTLOC.NUM",1,17);

asgn('File option prints numbers to a file named "ROOTLOC.NUM",1,17);

repeat

Read(&bd,ch);

if (ch = 'F') or (ch = 'F') or (ch = 'P') or (ch = 'P') then

if (ch = 'F') or (ch = 'F') then

assign(list,'ROOTLOC.NUM');

rewrite(list);

LineCount := LineCount + 6;

LineCount := 0;

write(list,'ZEROS ');

write(list,'REAL ');

write(list,'IMAGINARY');

write(list); LineCount := LineCount + 6;
with G_eq do
for i := 1 to NZeros do
  begin
    writeln(list, 'RealPartZero[i]:10:3', ',');
    writeln(list, 'ImagPartZero[i]:10:3');
  LineCount := LineCount + 1;
  end;
end;
end;
end;
end;

until ch in ['F','f','q','p','?'];
EnterGraphic;
swapscreen;
dosedist);
end;

Variable Gain := StartGain;
{compute root locations for varying values of gain and print them}

DeltaGain := (EndGain - StartGain)/100;
p := 0; q := 0;
for J := 1 to 100 do
  begin
    Variable_Gain := Variable_Gain + DeltaGain;
    With G_eq do
      begin
        HoldPoly := DenCoeff;
        if Neg_Feedback then
          for I := 1 to NZerosM do
            HoldPoly := HoldPoly[i] + (K * Variable_Gain) * NumCoeff[i];
        else
          for I := 1 to NZerosH do
            HoldPoly := HoldPoly[i] + (K * Variable_Gain) * NumCoeff[i];
        RootFinderNPoles, HoldPoly, PlotRealPole, PlotImagPole, pp, qq);
        writeln(list, Variable_Gain:10:4); LineCount := LineCount + 1;
        for i := 1 to NPoles do
          begin
            writeln(list, 'i:2,"PlotRealPole[i]:10:3",
            'PlotImagPole[i]:10:3');
            LineCount := LineCount + 1;
          end;
      end;
  end;
if LineCount > 50 then
  begin
    writeln(chr(12)); LineCount := 0;
  end;
end;
BEGIN (Root Locus Procedure)

P[1]:= '5504N01001-010101';  /* input handler driver strings */
P[2]:= '5588N01002-010103';
P[3]:= '1512N0050J-01010r';
P[4]:= '4513N005O4-010103';
P[5]:= '1514N005O5-01010r';
P[6]:= '4515N006-010103';
P[7]:= '4517N0207010101';

Clrscr; TextColor('White');
Center('*** ROOT LOCUS PLOTTING ROUTINE ***',1,2,80);
TextColor('Yellow');
Fill(5,100,11205):S:=copy(5,1,80);
writeln; writeln(S); writeln;
TextColor('Green');
writein('What STARTING value for variable gain do you wish?');
write;
writeln('What ENDING value for variable gain do you wish?');
TextColor('Yellow');
writein; writeln infl;

Center('*** VIEWING COORDINATES FOR ROOT LOCUS GRAPH ***',1,11,80);
TextColor('Green');
writeln('X-Minimum: ');
writeln('X-Maximum: ');
writeln('Y-Minimum: ');
writeln('Y-Maximum: ');
writeln; writeln('Positive or Negative Feedback? (P or N):');
Input_Handler('NO107',Escape);  /* prompts for NEW inputs */
writeln; writeln;
writein('Any changes to these parameters? (Y or N):');
Input('N',**'K$,Y,2',true,P,1,10);
If ans[Her]= 'Y' then Input_Handler('D0107',Escape);  /* prompts for changes */

Val[filvarll,StartGain,code);  /* converts input strings into */
Val[filvarl2,EndGain,code);  /* numeric values */
Val[filvarl3,Min,code);  
Val[filvarl4,Max,code);  
Val[filvarl5,Min,code);  
Val[filvarl6,Max,code);  

If copy[filvarl7,1,1] <> 'N' then Neg_feedback := false
else Neg_feedback := true;

INITGRAPHIC;  /* define values for graphics routine */
DefineWindow(1,0,0,MaxGlb,MaxGlb);
DefineWorld(ll,Min,Min,Max,Max);
SelectWindow(ll,SelectWorld(ll))
**Begin computation of closed loop roots and build plotting arrays.**

For \( I = 1 \) to \( N_{\text{zeros}} \) do:

\[
\begin{align*}
B &\quad \begin{cases}
\text{PlotZero}(2,1) := \text{PlotZero}(1,1); \\
\text{PlotZero}(2,2) := \text{PlotZero}(1,2); \\
\text{PlotZero}(3,1) := \text{PlotZero}(1,1); \\
\text{PlotZero}(3,2) := \text{PlotZero}(1,2); \\
\text{DrawPolygon(PlotZero,1,-3,-3,3,0);}
\end{cases} \\
\text{(DrawPolygon is graphics routine to plot an array of points - PlotZero)}
\end{align*}
\]

**Begin computing values of closed loop roots with varying gain**

\[
\begin{align*}
\text{Variable Gain} &:= \text{StartGain} \\
\text{DeltaGain} &:= (\text{EndGain}-\text{StartGain})/100; \quad \text{(divide gain to plot 100 points)} \\
\text{pg}=0; \text{qq}=0; \quad \text{(initial values for P and Q in rootfinder procedure)} \\
\end{align*}
\]

For \( j = 1 \) to 100 do (calculate and plot 100 points per graph):

\[
\begin{align*}
B &\quad \begin{cases}
\text{Variable Gain} := \text{Variable Gain} + \text{DeltaGain}; \\
\text{With G_eq do (use G_equivalent block and make closed loop - with unity feedback)}
\end{cases} \\
\end{align*}
\]

**Begin computing values of closed loop roots with varying gain**
For l:=1 to NZeros +1 do 
    HoldPoly[l] := HoldPoly[l] * (K»Variable_Bain » NumCoeff[l]);

RootFinder(NPoles,HoldPoly,PlotRealPole,PlotImagPole,pg,qg); 

For l:=1 to NPoles do  (fill plotting matrix with poles)
    Begin
        PlotPole[l,1] := PlotRealPole[l,1];
        PlotPole[l,2] := PlotImagPole[l,2];
        AxisGlib := true;
    end;

Case NPoles of 
    artificially fill plotting array if fewer than 3 points)
    begin
        PlotPole[2,1],PlotPole[2,2],PlotPole[3,1],PlotPole[3,2],PlotPole[4,1],PlotPole[4,2];
        DrawPolygon(PlotPole,1,-3,-1,2,0);
    end;
    begin
        PlotPole[3,1],PlotPole[3,2],PlotPole[4,1],PlotPole[4,2];
        DrawPolygon(PlotPole,1,-3,-1,2,0);
    end;
    else
        DrawPolygon(PlotPole,1,-NPoles,-1,2,0);
    end;
end;  (root locus procedure)

Repeat until KeyPressed;

quit := false;
repeat
    Graph_Menu('Root Locus',DumpGraph,quit);  (calls print/title menu)
    If DumpGraph then PrintGraphData;
    until quit;
LeaveGraphic;
end;
Program time_response;

(*$1 typedef.sys*)
(*$1 graphix.sys*)
(*$1 kernel.sys*)
(*$1 windows.sys*)
(*$1 axis.hgh*)
(*$1 polygon.hgh*)
(*$1 ut-mod01.inc*)
(*$1 ut-mod02.inc*)
(*$1 ut-mod03.inc*)
(*$1 Graphenu.inc*)

type
BigMatrix = array[1..30,1..30] of real;
BigVector = array[1..30] of real;

var
Psi ,Phi ,A,Temp : BigMatrix;
temp, inputtype : char;
Offset,Slope,Imax,
NumSum,MaxNumSum,1,
T,OldT,MaxT,
Plottime,Limit,Phi,,
hold,Imax,Imin,TPlot,
Amplitude, Freqy : real;
Factorial,Freqx,
Ninc, code, n, j, i, e, n : integer;

DupsGraph,BoodNumbers,
ClosedLoop, quit : boolean;
TSEG : blocks;
C, Hold, Next, Gamma : BigVector;
GraphArray, inputarray, plotarray;
List : text;

Procedure PrintGraphData; (dumps time-response data to printer)
Begin

LeaveGraphic;
Clrscr;
Center(*** TURN ON PRINTER AND ALIGN PAPER ***',1,10,80);
TextColor(green);
msg('press (P) to Print, (F) to list to file * , (Q) to quit print',1,13);
TextColor(white);
msg('File option prints numbers to a file named "THERESP.NUM",1,17);
msg('on the current drive. Browse the file off-line using the DOS "type" command.',1,18);

Repeat
Read(kbd,ch);
If (ch = 'F') or (ch = 'f') or (ch = 'p') or (ch = 'P') then
Begin
If (ch = 'F') or (ch = 'f') then
assign(list,'TimeResp.NUM');
Procedure Matrix_Mult(Matrix1,Matrix2:BigMatrix; var AnswerMatrix:BigMatrix; Order:integer);
  var
    i,j : integer;
  begin
  for i:=1 to order do
    for j:=1 to order do
      AnswerMatrix[i,j] := 0; (initialize the answer matrix)
  for i:= 1 to order do
    for j:= 1 to order do
      for L := 1 to order do
  end;

Procedure Scalar_Mult(Matrix1 : BigMatrix; scalar : real; var AnswerMatrix:BigMatrix;Order:integer);
  var i,j : integer;
  begin
    for i:= 1 to order do
      for j:= 1 to order do
        AnswerMatrix[i,j] := AnswerMatrix[i,j] + scalar*Matrix1[i,j];
  end;
for j:=1 to order do
    AnswerMatrix[i,j]:= AnswerMatrix[i,j] * scalar;
  end;

Procedure Matrix_Vector_Mult(Matrix: BigMatrix; Vector: BigVector;
  var AnswerVector: BigVector; Order: integer);

  var i,j : integer;

begin
  for i:= 1 to order do
  begin
    hold:= 0;
    for j:= 1 to order do
      hold:= hold + Matrix[i,j]*Vector[j];
    AnswerVector[i]:= hold;
  end;
end;

Begin
  initgraphic; leavegraphic;

  (********** Prompt user for desired input/time limit ***********)

  clrscr;
  TextColor(white);
  center('*** Time Response Plotting Routine ***',1,2,80);
  fillchar(s,100,'205'); S:= copy(s,1,80);
  writeln; writeln(s);
  TextColor(yellow);

  msg('What is the input to your system? STEP (S) ','1,6);
  msg(' RAMP (R) ','1,7);
  msg(' SIN WAVE (W) ','1,8);
  msg(' IMPULSE (I) ','1,9');

  InputType := temp;

  case InputType of
    1: Input('A','S','50,4,2,255,F1,F10); temp := copyanswer,1,1);
    1: if not (temp in ['S'/R'/H'/I']) then beep(350,150);
    1: until temp in ['S'/R'/H'/I'];
    1: InputType := temp;
    1: msg('Input amplitude? ','1,11');
    1: Input('W','1',20,11,2,255,F1,F10);
    1: val(answer,Amplitude,code);
    1: case InputType of

135
R: begin

msg('DC offset? ',1,13);
Input('N', 'O', 22,13,2,true,F1,F10);
val(answer,Offset,code);

msg('Slope? ', 25,13);
Input('N', '1', 22,13,3,true,F1,F10);
val(answer,slope,code);
end;

W: begin

msg('Frequency? (rad/sec)',1,13);
Input('N', '1', 23,13,5,true,F1,F10);
val(answer,Freq,code);
end;

end;

msg('Open (O) or Closed (C) Loop simulation? ',1,16);

repeat
Input('N', '1', 45,16,2,true,F1,F10);
temp := copy(answer,1,1);
if not (temp in ['0','C']) then beep(350,150);
until temp in ['0','C'];
if temp = 'C' then ClosedLoop:= true
else ClosedLoop:= false;

msg('How many seconds of simulation would you like to see? (99 max)',1,20);

repeat
Input('N', '1', 65,20,5,true,F1,F10);
val(answer,Tmax,code);
if Tmax > 99 then beep(350,150);
until Tmax <= 99;

c1rsr:
TextColor(white);
center('Calculating the Time Response — Please wait ***',110,80);
TextColor(yellow);

with G_eq do
for i:= 1 to NPoles do
   DenCoeff[i] := DenCoeff[i]/ DenCoeff[NPoles]; (normalize polynomial)

(*************************************************** Make Closed Loop G equivalent ***************************************************)

if ClosedLoop then
   begin
      with G_eq do
         (compute closed-loop (unity feedback) G equivalent)
      begin
         (called TRREQ -- which is also a blocks type record)
         TRREQ.K := K; (closed loop K same as open loop K)
         TRREQ.NZEROS:=NZeros; (as are the zeros of the function )
for i:=1 to maxorder do TRBEQ.DenCoeff[i]:= 0.0;  (initialize)

for i:=1 to NZeros + 1 do

begin
TRBEQ.DenCoeff[i]:= NumCoeff[i] * K;  (C.L. denominator equals the )
TRGEQ.NumCoeff[i]:= NumCoeff[i];  (sum of open loop denominator)
end;

if NPoles > NZeros then TRGED.NPOLES:=NPoles  (NPoles should always be)
else TRGED.NPOLES:=NZeros;  (greater, but to be safe)

end; (End)

else

TRGEQ := G_eq;

{The following is the A-matrix}

(* Fill the A-matrix *)

for i:= 1 to NPoles-1 do  (fill all but the bottom row)
for j:=1 to NPoles do
  if j = i+1 then AL[i,j]:= 1
  else AL[i,j]:= 0;

for j:=1 to NPoles do
  AL[NPoles,j]:= -DenCoeff[j];

(* Fill the C matrix *)

for i:= 1 to NPoles do

begin
if i > NZeros + 1 then C[i]:= 0.0
else C[i]:= NumCoeff[i] * K
if NZeros = NPoles then C[i] := C[i] * K * NumCoeff[NZeros+1]*AL[NPoles,i];
end;

(* Select a sampling time interval T *)

Nincr := 1000;
T := (Tmax/Nincr);
{Initialize psi & atemp (this will initialize psi to the value)
\begin{align*}
\text{Atemp} &:= \text{A}; \\
\text{Psi} &:= \text{A}; \\
\end{align*}
(of the infinite series after the first)
\begin{align*}
\text{Scalar Mult(\text{Psi}, \text{T/2}, \text{Psi}, \text{NPoles})}; \\
\text{for } i := 1 \text{ to NPoles do} \\
\text{Psi}_{i,i} &:= \text{Psi}_{i,i} + 1.0; \\
\end{align*}
(two terms \(1 + \text{Atemp} / 2\))

(Compute more terms of the series & truncate)
\begin{align*}
\text{Factorial} &::= 2; \\
\text{T1} &::= \text{T}; \\
\text{Oldmaxrowsum} &::= 0.0; \\
\end{align*}

\begin{align*}
\text{for } j := 1 \text{ to NPoles do} \\
\text{Rowsum} &::= 0.0; \\
\text{for } i := 1 \text{ to NPoles do} \\
\text{Rowsum} &::= \text{Rowsum} + \text{Psi}_{i,i}; \\
\text{if Rowsum > maxrowsum then maxrowsum := Rowsum; } \\
\text{end}; \\
\text{if } \left(\text{abs}(\text{maxrowsum} - \text{oldmaxrowsum}) / \text{maxrowsum} < 0.001 \right) \\
\text{then finished := false} \\
\text{else finished := true;} \\
\text{end}; \\
\text{Oldmaxrowsum} &::= \text{maxrowsum}; \\
\text{end}; \\
\text{end}; \\
\text{until Finished;} \\
\text{Scalar Mult(\text{Psi}, \text{T}, \text{Psi}, \text{NPoles});}
\end{align*}

(Calculate Phi matrix)
\begin{align*}
\text{Matrix Mult(\text{A}, \text{Atemp}, \text{Phi}, \text{NPoles});} \\
\text{Atemp} &::= \text{Phi}; \\
\text{Scalar Mult(\text{Phi}, \text{TI/Factorial}, \text{Phi}, \text{NPoles});} \\
\text{for } j := 1 \text{ to NPoles do} \\
\text{for } m := 1 \text{ to NPoles do} \\
\text{Psi}_{j,m} &::= \text{Psi}_{j,m} + \text{Phi}_{j,m}; \\
\text{end}; \\
\text{end}; \\
\text{end};
\end{align*}

(Calculate Gamma vector)
\begin{align*}
\text{for } i := 1 \text{ to NPoles do} \\
\text{Gamma}_{i} &::= \text{Psi}_{i,
\text{NPoles}}; \\
\text{\text{B}} &= \{ 0 0 0 0 \ldots 0 1 \} \text{(transpose)};
\end{align*}
Compute the next state of \( x \) given \( x(0) = 0 \)

\[
\text{Plottime} := 0.0; \quad \text{Plotindex} := 1; \quad \text{(initialize)}
\]

\[
\text{for } i := 1 \text{ to } \text{NPoles} \text{ do } \text{Xold}[i] := 0.0; \quad \text{(init. prev. state)}
\]

\[
\text{Ymax} := 0.0; \quad \text{Ymin} := 0.0;
\]

\[
\text{TextColor(white);} \quad \text{gotoxy(15,10)};
\]

\[
\text{write('*** Calculating Points — Please wait ***'); (countdown)}
\]

\[
\text{TextColor(yellow);} \quad \text{gotoxy(32,10)}; \quad \text{write(''); (display downcounter)}
\]

\[
\text{for } N := 1 \text{ to } \text{Nincr} \text{ do } \text{(begin calculating next state and } y\text{)}
\]

\[
\text{gotoxy(32,10)}; \quad \text{write(''); (display downcounter)}
\]

\[
\text{if abs(y) < 1.0E07 \text{ then } y := y + K[i] \times Xnext[i]; \quad \text{(max } y \text{ limit)}
\]

\[
\text{if } \text{NZeros} = \text{NPoles} \text{ then } y := y + K[i] \times \text{NumCoeff[Nzeros]}; \quad \text{Uinput};
\]

\[
\text{if } y > \text{Ymax} \text{ then } \text{Ymax} := y;
\]

\[
\text{if } y < \text{Ymin} \text{ then } \text{Ymin} := y;
\]

\[
\text{if } N \text{ mod 5} = 0 \text{ then } \text{(plot every 5th point)}
\]

\[
\text{GraphArray[Plotindex,1]} := \text{Plottime};
\]

\[
\text{GraphArray[Plotindex,2]} := y;
\]

\[
\text{Plotindex} := \text{Plotindex} + 1;
\]

\[
\text{y} := 0.0; \quad \text{for } i := 1 \text{ to } \text{NPoles} \text{ do }
\]

\[
\text{if } \text{abs}(y) < 1.0E07 \text{ then } y := y + K[i] \times Xnext[i]; \quad \text{(max } y \text{ limit)}
\]

\[
\text{if } y > \text{Ymax} \text{ then } \text{Ymax} := y;
\]

\[
\text{for } i := 1 \text{ to } \text{NPoles} \text{ do } \text{Xold}[i] := Xnext[i];
\]

\[
\text{Ymax} := 1.1 \times \text{Ymax};
\]
411 E----------------- repeat until keypressed;
412 B----------------- repeat
413 I                      Graph_Menu('Time-Response',DumpGraph,quit); (calls print/title menu)
414 I                      until quit;
415 I                      LeaveGraphic;
416 E----------------- end;
417 I                      assign(cadfile,'Cad.com');
418 I                      execute(cadfile);
419 I                      end.
420
421
422
423
424
Program TwoParameterRootLocus;

(* graphdef.sys *) (graphics type declarations)

const
maxorder = 20; (* maximum order for two parameter system *)

type
str80 = string[80];
stringarray = array[1..20] of str80;
PolyArray = array[1..20] of real;

var
answer : str80;
polish : str80;
eval : real;
InfixArray = array[1..20] of str80;
DeltaStep,
Increm,
a,b : real;
i,j,k,order : integer;
max,min,
ymax,ymin,
max,min;
Delta,bin : real;
DumpGraph,
Quit,Change,
StepA : boolean;
EvalArray,
RealPart,
ImgPart : PolyArray;
PlotPole : PlotArray;
symbol : integer;
hold : string[103];
Line : array[1..53] of string[14];
twochars : string[21];
fivechars : string[53];
CafFile : file;

(* graphix.sys *)
(* kernel.sys *)
(* windows.sys *)
(* polygon.ghh *)
(* axis.ghh *)
(* im-mod000.inc *) (slightly modified version of ut-mod000.inc)
(* ut-mod001.inc *)
(* ut-mod002.inc *)
(* ty-mod003.inc *) (customized version of ut-mod003.inc)
(* roots.inc *)

Procedure Infix_to_Polish(Answer: str80; var RPN : str80); forward;
(forward declaration of proc to convert infix notation to reverse Polish)

Procedure Compute_Polish(Polish: str80; a,b:real; var evaluation:real); forward;
(forward declaration of the proc to evaluate the reverse Polish expression)
Procedure Infix_to_Polish;
var
    Stack : Array[1..50] of char;
    Top, P, I : Integer;
    ch : char;
    Firstchar, Prev_Digit : Boolean;

Function Priority(Ch:char): Integer;
Begin
    Case ch of
        '"': Priority:= 4;
        '+', '-', '\', '^', '\': Priority:= 3;
        '0': Priority:= 2;
        ':': Priority:= 1;
        'A'. . . 'Z', 'a'. . . 'z': Priority:= 1;
        '0'. . . '9': Priority:= 0;
    End;
End;

Begin {procedure Infix_to_Polish)
    RFN := ''; (output string)                      (initialization)
    Top := 0; (operator stack pointer)
    Firstchar := True; (Firstchar helps to find unary minus signs)
    Prev_Digit := False; (Prev_Digit keeps spaces out of numeric constants)
    for i:= 1 to Length(Answer) do
        Begin {begin

ch := copy(AnsHer,i,l); (scan the infix expression char by char)
if Firstchar and (ch = '-') then P:= 1; (unary minus sign)
if P = 1 then (if the character is part of a constant, or variable)
begin
if Prev_digit then (suppresses blank spaces inside numeric)
RPN := concat(RPN,ch) (constants)
else
RPN := concat(RPN,' ',ch);
firstchar := False; Prev_Digit := True;
end;
if P > 1 then (if an operator)
begin
(checks priority of operator and arranges it on the stack)
while (Top > 0) and (Priority(Stack[top]) := P) do
begin
RPN := concat(RPN,' ',Stack[top]) (if smaller priority, then)
Top := Top - 1; (operator placed on stack)
end;
(otherwise operator on top is placed)
Top := Top + 1; (in output string and then current)
Stack[top] := ch; (operator is placed on stack)
Firstchar := True; Prev_Digit := False;
end;
if ch = '(' then (if opening paren then keeps inside stuff together)
begin
Top := Top + 1;
Stack[top] := '('.
FirstChar := True; Prev_Digit := False;
end;
if ch = ')' then (closing paren causes inside stuff to be put on)
begin
while Stack[top] <> '(' do
begin
RPN := concat(RPN,' ',Stack[top]);
Top := Top - 1;
end;
(skip over opening paren)
FirstChar := False; Prev_Digit := False;
end;
while Top > 0 do (put remaining operators on output string)
begin
RPN := concat(RPN,' ',Stack[top]);
Top := Top - 1;
end;
end; (procedure infix_to_polish)
Procedure Compute_Polish;

var
  i,code : integer;
  NumStack : array[1..40] of real;
  Top : integer; (NumStack pointer)
  ch : char;
  temp : string[201];
  Value1,Value2, Value3 : real;

  Procedure PUSH(Number:Real); (push a number onto numeric stack)
  begin
    Top := Top + 1;
    NumStack(Top) := Number;
  end;

  Procedure POP(var Number:Real); (pop a number off the numeric stack)
  begin
    Number := NumStack(Top);
    Top := Top - 1;
  end;

  function Expon(Y,1:real); (computes Y raised to 1 power)
  begin
    Expon := exp(1 * (ln(Y)));
  end;

  begin   {procedure Compute_Polish}
    temp := "";Top := 0;  (initialize)
    for i := 1 to Length(polish) do (do one char at a time)
      begin
        ch := copy(polish,i,1); (get a character)
        case ch of
        '0'..'9': temp := concat(temp,ch); (real constant)
        ' ': temp := concat(temp,ch);
        else
          begin
            if copy(polish,i+1,1) <> ' ' then (unary minus)
              begin
                ch := copy(polish,i,1);
                if ch = '-' then
                  begin
                    Value3 := Value2 - Value1;
                    PUSH(Value3);
                  end
                else
                  begin
                    FOP(Value1);
                    FOP(Value2);
                    Value3 := Value2 - Value1;
                    PUSH(Value3);
                  end
            end
          end;
        end;
    end;
begin (put variables a and b onto numeric stack)
PUSH(a);
PUSH(b);
end;

begin (add two numbers)
PUSH(Value1); POP(Value2);
Value3 := Value2 + Value1;
PUSH(Value3);
end;

begin (multiply two numbers)
PUSH(Value1); POP(Value2);
Value3 := Value2 * Value1;
PUSH(Value3);
end;

begin (divide two numbers)
PUSH(Value1); POP(Value2);
Value3 := Value2 / Value1;
PUSH(Value3);
end;

begin (exponentiation)
PUSH(Value1); POP(Value2);
Value3 := Expn(Value2,Value1);
PUSH(Value3);
end;

begin (space is divider between numbers/operators)
if temp <> " then
  begin
    val(temp,Value1,code);
    PUSH(Value1);
    temp := ";
  end;
end; (case)
end; (for)

if temp <> " then (lone constant in expression)
begin
  val(temp,Value1,code);
PUSH(Value1);
end;
end; (procedure)

end;
Procedure Coef_input;

var
code,i,linecount : integer;
validated : boolean;

begin
Clrscr; TextColor(White);
Center('*** Two Parameter Root Locus ***',1,2,80);
Fillchar(s,100,'#'); S := copy(s,100,'#'); writeln(s);
msg('What is the highest order coefficient in the Characteristic Equation?',1,8);
n:=input('N',8,2,134,Fl,FlO);
val(answer,Order,code);
clrscr;
Center('*** Two Parameter Root Locus - Coefficient Input ***',1,1,80);
writeln(s); writeln(s);
P[1]:= 'OB05A07201001001';
P[2]:= 'OB07A07202010102';
P[3]:= 'OB09A07203010103';
P[4]:= 'OB11A07204010104';
P[5]:= 'OB13A07205010105';
P[6]:= 'OB15A07206010106';
P[7]:= 'OB17A07207010107';
P[8]:= 'OB19A07208010108';
P[9]:= 'OB21A07209010109';
P[10]:= 'OB23A07210010110';
P[11]:= 'OB25A07211010111';
P[12]:= 'OB27A07212010112';
P[13]:= 'OB29A07213010113';
P[14]:= 'OB31A07214010114';
P[15]:= 'OB33A07215010115';
P[16]:= 'OB35A07216010116';
P[17]:= 'OB37A07217010117';
P[18]:= 'OB39A07218010118';
P[19]:= 'OB41A07219010119';
P[20]:= 'OB43A07220010120';
for i:= 1 to order + 1 do
  begin
    writeln('order',i);
    writeln('s = ',i);
  end;
str(order+12,'wchars');
if order+1 < 10 then
  (computes involving string for input_handler)
end;
File: TKOHARM.PAS  Cross-Reference & Block Listing  Date: 9/10/86  Page 7

342  | B------------------------------------begin
343  |     delete(twochars,1,11);
344  |     insert('0',twochars,11);
345  | E------------------------------------end;
346  |
347  | if order + 1 > 10 then fivechars := 'WHO'
348  | else fivechars := concat('WHO',twochars);
349  | Input_handler(fivechars,escape);  (call the input_handler)
350  |
351  | if order + 1 > 10 then      (if greater than 10th order, write last 10 on next page)
352  | B------------------------------------begin
353  |     fivechars := concat('WHO',twochars);
354  |     clrscr;
355  |     Center('*** Two Parameter Root Locus - Coefficient Input ***',11,80);
356  |     writeln(strin),writeln;
357  |     Input_handler(fivechars,escape);
358  | E------------------------------------end;
359  | for i := 1 to order + 1 do
360  |     Input_reader(varl,2:i-1 := Filvarl);
361  | E------------------------------------(procedure Coeff Input)
362  |
363  |
364  |
365  |
366  |
367  | {**************************************************************************}
368  | (** Select_Parameter_Range prompts the user for values of A and B to be ***)
369  | (** used for plotting the root locus. One parameter is stepped through ***)
370  | (** five equal increments of range and the other is plotted "smoothly" ***)
371  | (** by computing and plotting fifty values. ***)
372  | {**************************************************************************}
373  |
374  | Procedure Select_Parameter_Range;
375  | var
376  |   code  : integer;
377  |
378  | B------------------------------------begin
379  |     clrscr; TextColor(white);
380  |     Center('*** Parameter Selection Page ***',11,80);
381  |     fillchars,s,100,42,150); S:= copy(s,180);TextColor(yellow);
382  |     writeln(strin),writeln;
383  |     TextColor(green);   (print message explaining input for 2param proc)
384  |
385  |     writeln('You will be varying the two parameters, A and B, through a range');
386  |     writeln('of values you select. You will also choose to STEP either A or B');
387  |     writeln('which means that the chosen parameter's range will be divided into');
388  |     writeln('five (5) equal increments for plotting while the other parameter
');
389  |     writeln('varies smoothly through its range.');
390  |     writeln(strin),writeln;
391  |
392  |     Fil1:= '15161001001-010103';
393  |     Fil2:= '15171001002-010103';
394  |     Fil3:= '15181001003-010103';
395  |     Fil4:= '15201001004-010101';
396  |     Fil5:= '15221002050110101';
397  |
398  |     TextColor(Yellow); (prompt for user inputs of parameter values)
writeln('A-minimum');
writeln('A-maximum');
writeln('B-minimum');
writeln('B-maximum');

Input_Handler('NO105', Escape); (prompts for NEW inputs)

repeat
  ch := copyVar[OS][l, l];
  if not ch in ['A', 'B'] then
    Input_Handler('NO105', Escape);
  until ch in ['A', 'B'];

if ch = 'A' then stepA := true
else stepA := false;

val(var[01], min, code);
val(var[02], max, code);
val(var[03], bin, code);
val(var[04], bmax, code);

end;

procedure Select_Window_Size;

var
code : integer;

begin
  Clrscr; TextColor(white);
  Center('*** Window Size Selection Page ***', 1, 2, 80);
  Fillchars(100, #0(5)); S= copy(1, 80); TextColor(yellow);
  writeln writeln(var[01]); writeln writeln(var[02]);
  writeln P[7] := '1510BN0100-010103';
  writeln P[8] := '1513BN0100-010103';
  writeln P[9] := '1513BN0100-010103';
  writeln writeln('X-minimum'); (prompts for window size)
  writeln('X-maximum');

  if change then Input_Handler('NO105', Escape) (prompts for CHANGE inputs)
else Input_Handler('00609',Escape);  {prompts for NEW inputs}

val[06], xmin, code);
val[07], ymin, code);
val[08], ymax, code);

change := true;

--------------- end;

else

Procedure Make_Legend;
var
line1, line2, line3, line4, line5, line6 : string[20];

begin
    copyscreen; {make a window}
    SetLineStyle(10);
    DefineWindow(2,11,20,30,80);
    DefineWorld(2,0,30,20,0);
    SelectWorld(2); SelectWindow(2);
    DefineHeader(2,'Legend');
    SetBackground(10);
    SetHeaderOn; DrawBorder;

    Linel := concat( 'KKK ',line1); {legend lines}
    Line2 := concat( 'XXX ',line2);
    Line3 := concat( 'YYY ',line3);
    Line4 := concat( '000 ',line4);
    Line5 := concat( '111 ',line5);
    Line6 := 'Press F1 to Zoom';

    DrawTextW(1,1,Line1); {write lines to window}
    DrawTextW(1,2,Line2);
    DrawTextW(1,3,Line3);
    DrawTextW(1,4,Line4);
    DrawTextW(1,5,Line5);
    DrawTextW(1,6,Line6);

    SetBreakOff; SetMessageOff;
    Set_Cap_Num(' ',' ',' '); Say_Cap_Num;

repeat
    read(kbd,ch);
    case ord(ch) of
        72 : MoveVert(-4,true); {up arrow}
        75 : MoveHor(-1,true); {left arrow}
        77 : MoveHor(1,true); {right arrow}
80  ; Move Vert(4, true); (down arrow)
514  B-------------57 : begin quit := false; (FI pressed indicating that user wants)
515  B-------------5f : begin leavegraphic; (to change graph scales)
516  E-------------end;
517  E-------------end;
518  E-------------until (ord(ch) = 13) or (ord(ch) = 59); (quit when (return) or (FI) pressed)
519  E-------------end;

****** Start of main program. This part increments through the ranges of **
520  B-------------Start of main program. This part increments through the ranges of **
521  B-------------A and B and plots the graph. The above procedures are called as **
522  B-------------needed by the main program.  **
523  B-------------***********************************************************************
524  B-------------begin (Main Program)
525  B-------------InitGraphic; LeaveGraphic; (initialize graphics routines & screen)
526  B-------------Select_Parameter_Range; (prompt for A & B parameter ranges)
527  B-------------
528  B-------------if stepa then (set up steps and increments for A & B)
529  B-------------begin
530  B-------------DeltaStep := abs((amax-amin)/4);  
531  B-------------Increm := abs((bmax-bmin)/50);  
532  B-------------end
533  B-------------else
534  B-------------begin
535  B-------------DeltaStep := abs((bmax-bmin)/4);  
536  B-------------Increm := abs((amax-amin)/50);  
537  B-------------end  
538  B-------------CoeffInput; (input the coefficients of the characteristic equation)
539  B-------------Change := false; (initialize change boolean)
540  B-------------Repeat
541  B-------------Select_Window_Size; (prompt for window size - x and y min's and max's)
542  B-------------a := amin; b := bmin; (initialize a and b)
543  B-------------INIGRAPIC; (define values for graphics routine)
544  B-------------DefineWindow(l,0,0,MmaxBib,MmaxBib);  
545  B-------------DefineWorld(l,min,ymax,xmax,ymin);
546  B-------------SelectWindow(l);SelectWorld(l);
547  B-------------DrawAxis(S,5,0,0,0,0,0,0,false);
548  B-------------
549  B-------------for j := 1 to 5 do (step through 5 values of selected stepping parameter)
550  B-------------begin
551  B-------------case j of (select which graph symbol will be used)
552  B-------------1 : symbol := 1; (* ) 
553  B-------------2 : symbol := 2; (x ) 
554  B-------------3 : symbol := 7; ( ) 
555  B-------------4 : symbol := 0; ( o )
for k := 1 to 50 do (smoothly move through the other parameter)
for i := 1 to Order + 1 do (compute each coefficient value)

for k := 1 to 50 do (range in 50 increments)

for i := 1 to Order do (compute each coefficient value)

infix_to_polish(InfixArray[i], polish); (parse infix)
Compute_Polish(polish, a, b, EvalArray[i]); (evaluate polish)
end;

RootFinder(Order, EvalArray, RealPart, ImagPart, 0, 0); (find eq roots)

For i := 1 to Order do (fill plotting matrix with poles)

RootFinder(Order, EvalArray, RealPart, ImagPart, 0, 0); (find eq roots)

Case Order of
0 : (of a and b)

1 : begin
PlotPole[1, 1] := RealPart[1];
PlotPole[1, 2] := ImagPart[1];
end;

AccessFlag := true;

B := Case Order of (artificially fill plotting array if fewer
than 3 points)

0 : ; (% plot the array for one value)

if stepA then b := b + increment (increment one parameter)
else a := a + increment;

if stepA then

str(a:10:4, hold); (build legend string)
Line[j] := concat('A = ', hold);
a := a + DeltaStep;

b := bmin;

end; (next step)

else

str(b:10:4, hold); (setup legend string)
Line[j] := concat('B = ', hold);

end;

E; (case)
627 | b := b + DeltaStep;
628 | a := amin;
629 | end;
630 | end;
631 | quit := true;
632 | Make_Legend; (draw legend box on screen)
633 | until quit; (quit when finished with legend, repeat if user wants to)
634 | rescale graph by pressing Fl in Make_Legend routine)
635 | quit := false; (this is a different use of quit - signalled when user)
636 | done with graph options menu)
637 | repeat
638 | Graph_Menu('2-Param.Root Locus',DumpGraph,quit); 
639 | Until quit;
640 | LeaveGraphic;
641 | Assign(CadFile,'Cad.COM'); (re-execute the main menu program when finished)
642 | Execute(CadFile);
643 | end.
const MaxWorldsGlb = 4;
MaxWindowsGlb = 16;
MaxFilesGlb = 10;
MaxPlotGlb = 500;
MaxOrder = 9;
MaxBlocks = 10;
StringSizeGlb = 80;
HeaderSizeGlb = 10;
RamScreenGlb: boolean = true;
CharFilestring[StringSizeGlb] = 'full.exe';
MaxProcsGlb = 27;
MaxErrsGlb = 7;

type
    worldstring = string[StringSizeGlb];

B------ WorldType = record
    x1,y1,x2,y2:real;
end;

E------ WindowType = record
    x1,y1,x2,y2:integer;
    header: worldstring;
    drawn,top: boolean;
    size: integer;
end;

E------ worlds = array [0..MaxWorldsGlb] of WorldType;

E------ windows = array [0..MaxWindowsGlb] of WindowType;

E------ PlotArray = array [0..MaxPlotGlb, 1..2] of real;

E------ character = array [0..31] of byte;

E------ CharArray = array [32..126] of character;

B------ Pietype = record
    area: real;
    text: worldstring;
end;

E------ PieArray = array [0..MaxFilesGlb] of Pietype;

E------ BackgroundArray = array [0..7] of byte;

E------ LineStyleArray = array [0..7] of boolean;

str255 = string[255];
str80 = string[80];
str40 = string[40];
str20 = string[20];
str8 = string[8];
str5 = string[5];
str4 = string[4];
str2 = string[2];
PolyArray = array[1..MaxOrder] of Real;

var

XWorld,X2World,Y1World,Y2World,XRef,Glb,YRef,Glb,XGlb,YGlb :real;

LineStyleGlb,MaxHorGlb,MaxWindowGlb,WindowModeGlb :integer;

XGlb,YGlb,X2Glb,Y2Glb :integer;

TextStyleGlb,TextStyleGlb,TextStyleGlb :integer;

PieGlb,DirectModeGlb,ClippingGlb,AxisGlb,HatchGlb :boolean;

MessageGlb,PrGlb,Glb,ToolGlb,GraphModeGlb :boolean;

CntGlb,CDlarGlb :byte;

ErrCodeGlb :byte;

LineStyleGlb :array[0..MaxProcesses] of 'HrkString;

ErrorCode :array[0..MaxErrors] of 'HrkString;

PsGlb :string[40];

AspectGlb :real;

GrafBase :integer;

world :worlds;

window :windows;

CharSet :CharArray;

Ch :char;

Answer,Previous_Answer,S :str80;

Template :str80;

P,File :array[1..35] of str40; (menu prompts)

Insert,On,Exit,

Escape,

F1,F10,

Use_Default,

First_Run :Boolean;

NBlocks :Integer;

Block :Array[1..MaxBlocks] of Blocks;

G_eq :Blocks;

ConvergenceError :Boolean;

Change :Boolean;

Finished :Boolean;

InputFile,CadFile,TimeFile,

FreqFile,UtilFile,TwoParaFile :File;

Drive :str2;

NegFeedBack :Boolean;

end;
Procedure RootFinder (Integer; var Coeff,RealPartRoot,ImagPartRoot: PolyArray; P1,Q1: Real);

(Const Epsilon = 0.00001; (acceptable computation error)
Var
A,B,C : PolyArrays; (calculation arrays)
P,Q,Deltap,Deltar,Denom : Real; (P,Q: coeff of quadratic factor)
(Deltap,Deltar: iteration increment)
(Denom: Denom used to compute )

IterationCount,1 : Integer; (IterationCount: iteration counter)
(i: misc. loop counter)

finished : Boolean; (finished: flag to tell when done)

PROCEDURE Solve_Quadratic (var b,c,real1,imag1,real2,imag2:real);

(Const Epsilon = 0.00001; (acceptable computation error)
Var
S,radical : real;

B--------------------------------------------- (procedure solves quadratic equation)
S:= -b/2.0;
radical := (b*b)/4.0 - c;
if radical > 0 then
    real1 := S+sqrt(radical);
    real2 := S-sqrt(radical);
    imag1 := 0.0;
    imag2 := 0.0;

E---------------------------------------------

else

B--------------------------------------------- (img roots)
real1 := S;
real2 := S;
imag1 := sqrt(-radical);
imag2 := -sqrt(-radical);

E---------------------------------------------

End;

B---------------------------------------------
End;  

B--------------------------------------------- (PROCEDURE ROOTFINDER)

(Const Epsilon = 0.00001; (acceptable computation error)
Var
A,B,C : PolyArrays; (calculation arrays)
P,Q,Deltap,Deltar,Denom : Real; (P,Q: coeff of quadratic factor)
(Deltap,Deltar: iteration increment)
(Denom: Denom used to compute )

IterationCount,1 : Integer; (IterationCount: iteration counter)
(i: misc. loop counter)

finished : Boolean; (finished: flag to tell when done)

PROCEDURE Solve_Quadratic (var b,c,real1,imag1,real2,imag2:real);

(Const Epsilon = 0.00001; (acceptable computation error)
Var
S,radical : real;

B--------------------------------------------- (procedure solves quadratic equation)
S:= -b/2.0;
radical := (b*b)/4.0 - c;
if radical > 0 then
    real1 := S+sqrt(radical);
    real2 := S-sqrt(radical);
    imag1 := 0.0;
    imag2 := 0.0;

E---------------------------------------------

else

B--------------------------------------------- (img roots)
real1 := S;
real2 := S;
imag1 := sqrt(-radical);
imag2 := -sqrt(-radical);

E---------------------------------------------

End;

B--------------------------------------------- (PROCEDURE ROOTFINDER)

(Const Epsilon = 0.00001; (acceptable computation error)
Var
A,B,C : PolyArrays; (calculation arrays)
P,Q,Deltap,Deltar,Denom : Real; (P,Q: coeff of quadratic factor)
(Deltap,Deltar: iteration increment)
(Denom: Denom used to compute )

IterationCount,1 : Integer; (IterationCount: iteration counter)
(i: misc. loop counter)

finished : Boolean; (finished: flag to tell when done)

PROCEDURE Solve_Quadratic (var b,c,real1,imag1,real2,imag2:real);

(Const Epsilon = 0.00001; (acceptable computation error)
Var
S,radical : real;

B--------------------------------------------- (procedure solves quadratic equation)
S:= -b/2.0;
radical := (b*b)/4.0 - c;
if radical > 0 then
    real1 := S+sqrt(radical);
    real2 := S-sqrt(radical);
    imag1 := 0.0;
    imag2 := 0.0;

E---------------------------------------------

else

B--------------------------------------------- (img roots)
real1 := S;
real2 := S;
imag1 := sqrt(-radical);
imag2 := -sqrt(-radical);

E---------------------------------------------

End;

B--------------------------------------------- (PROCEDURE ROOTFINDER)

(Const Epsilon = 0.00001; (acceptable computation error)
Var
A,B,C : PolyArrays; (calculation arrays)
P,Q,Deltap,Deltar,Denom : Real; (P,Q: coeff of quadratic factor)
(Deltap,Deltar: iteration increment)
(Denom: Denom used to compute )

IterationCount,1 : Integer; (IterationCount: iteration counter)
(i: misc. loop counter)

finished : Boolean; (finished: flag to tell when done)

PROCEDURE Solve_Quadratic (var b,c,real1,imag1,real2,imag2:real);

(Const Epsilon = 0.00001; (acceptable computation error)
Var
S,radical : real;

B--------------------------------------------- (procedure solves quadratic equation)
S:= -b/2.0;
radical := (b*b)/4.0 - c;
if radical > 0 then
    real1 := S+sqrt(radical);
    real2 := S-sqrt(radical);
    imag1 := 0.0;
    imag2 := 0.0;

E---------------------------------------------

else

B--------------------------------------------- (img roots)
real1 := S;
real2 := S;
imag1 := sqrt(-radical);
imag2 := -sqrt(-radical);

E---------------------------------------------

End;

B--------------------------------------------- (PROCEDURE ROOTFINDER)

(Const Epsilon = 0.00001; (acceptable computation error)
Var
A,B,C : PolyArrays; (calculation arrays)
P,Q,Deltap,Deltar,Denom : Real; (P,Q: coeff of quadratic factor)
(Deltap,Deltar: iteration increment)
(Denom: Denom used to compute )

IterationCount,1 : Integer; (IterationCount: iteration counter)
(i: misc. loop counter)

finished : Boolean; (finished: flag to tell when done)

PROCEDURE Solve_Quadratic (var b,c,real1,imag1,real2,imag2:real);

(Const Epsilon = 0.00001; (acceptable computation error)
Var
S,radical : real;

B--------------------------------------------- (procedure solves quadratic equation)
S:= -b/2.0;
radical := (b*b)/4.0 - c;
if radical > 0 then
    real1 := S+sqrt(radical);
    real2 := S-sqrt(radical);
    imag1 := 0.0;
    imag2 := 0.0;

E---------------------------------------------

else

B--------------------------------------------- (img roots)
real1 := S;
real2 := S;
imag1 := sqrt(-radical);
imag2 := -sqrt(-radical);

E---------------------------------------------

End;

B--------------------------------------------- (PROCEDURE ROOTFINDER)

(Const Epsilon = 0.00001; (acceptable computation error)
Var
A,B,C : PolyArrays; (calculation arrays)
P,Q,Deltap,Deltar,Denom : Real; (P,Q: coeff of quadratic factor)
(Deltap,Deltar: iteration increment)
(Denom: Denom used to compute )

IterationCount,1 : Integer; (IterationCount: iteration counter)
(i: misc. loop counter)

finished : Boolean; (finished: flag to tell when done)
For i:=3 to N+3 do Ali := Coeff[iN+3]/Coeff[i+1]; (normalize to unity, highest order coeff)

For i=1 to maxorder do Bi[i] := 0.0; (initialization)
C:=0; IterationCount:=1; finished := false;
P:=P1; Q:=Q1; (best guesses for initial starting values - normally 0)

This section solves simple polynomials of order zero or one.

If N=0 then finished:= true; (quit if polynomial is order 0)

If N=1 then (solve polynomial of form: x + a = 0)

If N=2 then (solve polynomial of form: x^2 + ax + b = 0)

While (not finished) and (IterationCount < 40) do

For i:=3 to N+3 do (solves iteratively for quadratic coefficients)

B[3]:= Ali - Pi/2 - Q/B[i-2];
C[3]:= - B[3] - C[i-1]/2;

Denom := C[n+1] * C[n+1] - C[i+2] * B[i+2]) * C[n];

If Denom <> 0.0 then

DeltaP := (B[n+2] * C[n+1] - B[n+1] * C[n]) / Denom;
DeltaQ := (C[n+1] * B[n+1] - C[i+2] * B[i+2]) / Denom;
P:= P+DeltaP; Q := Q+DeltaQ;

If (abs(DeltaP) + abs(DeltaQ)) < epsilon then

Solve_Quadratic(P, Q, RealPartRoot[n], ImagPartRoot[n], RealPartRoot[n-1], ImagPartRoot[n-1]);
n:=n-2;

case n of
  0: finished := true;  
  1: Begin
    RealPartRootIn := realRoot(n+1)/realRoot(n+2);
    ImagPartRootIn := 0.0;
    finished := true;
  End;
  2: Begin
    Solve_Quadratic(B(n+2),B(n+3),RealPartRootIn,ImagPartRootIn,RealPartRootIn-1,ImagPartRootIn-1);
    end;
  else
    For i:= 3 to N+3 do A[i] := B[i];  
    B is reduced order polynomial. Assign to A and iterate again.
    end;
  End
  else
    IterationCount := IterationCount+1;
  End{While)
  else
    P:=P+1;  Q:=Q+1;  IterationCount:=1;  beep(350,150);
  end;
End{Procedure)
The routine Expand_Poly takes real and complex conjugate factors of a polynomial and expands the factors into that polynomial. The routine uses a shift-multiply-add algorithm for combining the factors two at a time. Complex conjugate factors are pre-multiplied in the procedure. Conjug_Mult can yield a real second-order sub-polynomial and then combined with the remaining factors.

Procedure Expand_Poly(RealPt, ImagPt: PolyArray; var Poly: PolyArray; Order: integer);

Type
    Vec2 = Array[1..2] of real;

Var
    Temp : PolyArray;
    Hold : Vec2;
    I : Integer;
    Monomial : Boolean;

Procedure Conjug_Mult(R1, R2: real; var Poly1, Poly2: real);

Begin
    Poly1 := 1.0 + R1 * R2;       (local procedure to expand a quadratic factor)
    Poly2 := 2.0 * R1;           (into a second-degree polynomial)
end;

Begin {Expand_Poly}

For I:=1 to MaxOrder do

Begin
    Temp[I] := 0.0; Poly[I] := 0.0;    (initialize the working arrays)
end;

If Order = 0 then Poly[1] := 1.0;
While Order > 0 do

Begin
    If ImagPt[Order] = 0.0 then

    Temp[I] := -RealPt[Order];
    Temp[2I] := 1.0;
    Poly[I] := Temp[I];
End
Poly[21] := Temp[21];
Order := Order - 1;
else
begin
Conjug.Mult(-RealPt[Order], ImagPt[Order], Temp[1], Temp[2]);
Poly[21] := Temp[21];
Temp[31] := 1.0;
Poly[31] := 1.0;
Order := Order - 2;
end;
end;
end;

While Order > 0 do
begin
if ImagPt[Order] = 0.0 then
begin
if Monomial then
begin
Hold[1] := -RealPt[Order];
Hold[2] := 0.0;
end
else
begin
Hold[1] := 0.0;
Hold[2] := RealPt[Order];
end;
end
else
begin
Conjug.Mult(-RealPt[Order], ImagPt[Order], Hold[1], Hold[2]);
end;
end;
if Not(Monomial) then
begin
For I := MaxOrder downto 2 do Poly[I] := Poly[I - 1];
Poly[I] := 0.0;
Order := Order - 1;
end;
end;
end;
end;
end; (while)
end; (while)
end; (procedure)
Procedure HftKEGEQ;

Var
  i,j       : integer;  {general purpose loop indices}
  Fwd_count_I,
  Fwd_count_P,
  Fbk_count_I,
  Fbk_count_P : integer;  {pole and zero counters}
  RTemp, ITemp,
  TempPoly1,
  TempPoly2,
  Fwd_I1,Fwd_R1,
  Fwd_IP,Fwd_RP,
  Fbk_I1,Fbk_R1,
  Fbk_IP,Fbk_RP : PolyArray;  {Gef factored form holding arrays}
  Fwd_gain,
  Fbk_gain       : Real;
  count,count2  : integer;  {temporary counters}
  FeedbackPathExists: Boolean;

Begin
  Fwd_count_I:= 0;  Fwd_count_P:=0;  {initialize temporary holding variables}
  Fbk_count_I:= 0;  Fbk_count_P:=0;
  Fwd_gain := 1;  Fbk_gain := 1;
  FeedbackPathExists := false;

  For j:=1 to NBlocks do
    (repeat for every block in the loop)

    Begin
      with Block(j) do
        begin
          If Feedback then
            (if the block is in feedback path do this)
            begin
              For i:=1 to NZeros do
                (gather poles & zeros for all feedback blocks — equivalent to multiplying
                block polynomials together)
                Fbk_count_Z:=Fbk_count_Z + 1;
                Fbk_RZ[Fbk_count_Z]:= RealPartZerot[i];
                Fbk_IP[Fbk_count_Z]:= ImagPartZerot[i];
            end;
            For i:=1 to NPoles do
              (gather poles and zeros for all forward path blocks — same multiplicative
              steps for both)
              Fbk_count_P:=Fbk_count_P + 1;
              Fbk_IP[Fbk_count_P]:= RealPartPole[i];
              Fbk_IP[Fbk_count_P]:= ImagPartPole[i];
          end;
          Fbk_gain := Fbk_gain * K;
          FeedbackPathExists := true;
        end
        else
          begin
            else
              begin
                (gather poles and zeros for all forward path blocks — same multiplicative
                steps for both)
                Fbk_gain := Fbk_gain * K;
                FeedbackPathExists := true;
              end
          end
    end
End.
For i!=l to NZeros do 

begin
Fwd_count_L:=Fwd_count_L + 1;
Fwd_RZ[Fwd_count_L]:= RealPartZero[i];
Fwd_IL[Fwd_count_L]:= ImagPartZero[i];
end;

For i=1 to NPoles do 

begin
Fwd_count_P:=Fwd_count_P + 1;
Fwd_RP[Fwd_count_P]:= RealPartPole[i];
Fwd_IP[Fwd_count_P]:= ImagPartPole[i];
end;

Fwd_gain := Fwd_gain * K;
end; (else)

end; (or)

(Make the G equivalent block from poles and zeros gathered previously)

with G_eq do

begin
For i:=1 to MaxOrder do (initialize G_eq factors)

begin
RealPartZero[i]:=0.0; ImagPartZero[i]:=0.0;
RealPartPole[i]:=0.0; ImagPartPole[i]:=0.0;
NunCoeff[i]:=0.0; DenCoeff[i]:=0.0;
end;

end;

NZeros:=0;

(****** G equivalent Zeros are product of Forward path zeros and ******) 

(Feedback path poles.)

For i:= 1 to Fwd_count_L do 

begin
NZeros:=NZeros + 1;
RealPartZero[NZeros] := Fwd_RZ[i];
ImagPartZero[NZeros] := Fwd_IL[i];
end;

For i:= 1 to Fbk_count_P do 

begin
NZeros:=NZeros + 1;
RealPartZero[NZeros] := Fbk_RP[i];
ImagPartZero[NZeros] := Fbk_IP[i];
end;

If NZeros = 0 then NunCoeff[1] := 1.0
else
Expand_poly(RealPartZero, ImagPartZero, NunCoeff, NZeros);

161
count:=0; count2:=0;

For i:=1 to Fwd_count_I do
  begin
    count:=count+1;
    {collect forward path zeros}
    RTemp[count] := Fwd_RZ[i];
    ITemp[count] := Fwd_ILZ[i];
  end;

For i:=1 to Fbk_count_I do
  begin
    count:=count+1;
    {combine with feedback path zeros}
    RTemp[count] := Fbk_RZ[i];
    ITemp[count] := Fbk_ILZ[i];
  end;

Expand_Poly(RTemp, Itemp, TempPoly, count);

For i:=1 to count + 1 do
  begin
    {multiply zeros polynomial by gains}
    TempPoly[i] := TempPoly[i-1] * Fwd_Gain * Fbk_Gain;
  end;

For i:=1 to Fwd_count_P do
  begin
    count2:=count2+1;
    {collect forward block poles}
    RTemp[count2] := Fwd_RPi[i];
    ITemp[count2] := Fwd_IP[i];
  end;

For i:=1 to Fbk_count_P do
  begin
    {combine with feedback block poles}
    RTemp[count2] := Fbk_RPi[i];
    ITemp[count2] := Fbk_IP[i];
  end;

if not(FeedbackPathExists) then
  begin
    {if there are no feedback blocks, then}
    for i:=1 to Fwd_count_P do
      {G_eq poles are fwd block poles}
      RealPartPole[i] := Fwd_RPi[i];
      ImagPartPole[i] := Fwd_IP[i];
  end;

MPoles := Fwd_count_P;
Expand_Poly(RealPartPole, ImagPartPole, DenCoeff, MPoles);

else
  begin
    {otherwise add two temporary polynomials}
    Expand_Poly(RTemp, Itemp, TempPoly2, count2);
    if count > count2 then MPoles := count;
    end;

162
else npoles := count2;  (poly is npoles)
173  For i:=1 to npoles + 1 do
174    If NegFeedback then DenCoeff[i] := TempPoly[i] + TempPoly[i]
175    else DenCoeff[i] := TempPoly[i] - TempPoly[i];
176
177  RootFinder(npoles, DenCoeff, RealPartPole, ImagPartPole, 0, 0);
178  end;
179
180  K := Fwd_gain;  (G_eq block gain is combined forward path gains)
181
182  FeedBack := false;  (set G_eq equivalent boolean flags)
183  Factored := true;
184  LeadNumCoeff := 1; LeadDenCoeff := 1;
185
186  end;  (with)
187  end;  (procedure)
Cross-Reference & Block Listing

Procedure ShowRoots(x_eq : Blocks);

Var
  I,N,PosCounter: Integer;
  RealRoot,
  ImagRoot,
  ClosedLoopPoly: PolyArray;
  key,keyold      : integer;
  not_erased      : boolean;

Begin

With G_eq do

  ClrScr; HioghVideo; (on-screen titles)
  Center('** Block Transfer Function Closed-Loop Roots **',1,2,80);
  writeln; writeln; writeln;
  writeln('ZEROS: ');

  For I:=1 to NZeros do (position for output)
  begin
    PosCounter := I mod 2;
    If PosCounter = 1 then writeln;
    LowVideo; (write zeros - note: zeros of D.L. system same as
    C.L. roots)
    writeln('RealPartZerodlMO,' +j +ImagPartZerodlMO);
    writeln(' ');
  end;

  writeln; writeln; HighVideo;
  writeln('POLES: ');

  (Make closed loop, unity-feedback system)
  For I:=1 to Norder do ClosedLoopPoly[I] := 0.0;
  For I:=1 to NZeros + 1 do ClosedLoopPoly[I] := NumCoeff(I) * X;
  For I:=1 to Npoles + 1 do ClosedLoopPoly[I] := ClosedLoopPoly[I] +
                      DenCoeff(I);

  If Npoles > NZeros then N := Npoles
  else N := NZeros;

  RootFinder(N,ClosedLoopPoly,RealRoot,ImagRoot,0,0); (find factors)

  For I:=1 to N do
  begin
    PosCounter := (I mod 2);
    If PosCounter = 1 then writeln;
  end;

End.
LoVideo;
(output roots)
write('st', 1, 1) = ', RealRoot(11:10:3, ' + j', ImagRoot(11:10:3);
write(' ');
end;(for)
end;(with)

HighVideo;

msg ('Press any key to continue or (Shift) [PrtSc] for hardcopy.', 1, 24);

(check keyboard buffer for value change. If number changes by 1 or 2
indicates that shift key depressed. If so, then remove “Press any key...”
prompt from screen so it won’t print to printer)

keyold :- BB(II[0000:1047);
not "erased := true;

key := BB(II[0000:1047);
not erased := false;

if ((key=keyold + 1) or (key=keyold + 2)) and (not erased) then
begin
GotoXY(l, 24); write' ': 80); 
not erased := false;
end;

 Until KeyPressed;
end;(Procedure ShowRoots)
Procedure Show_Factored_Roots;
Var
I,M,N,PosCounter: Integer;
NomFact := Real;
key: keyold := Integer;
inc erased := Boolean;

Begin
if NBlocks = 0 then     {print warning if no blocks currently in memory)
begin
crscr;TextColor(Red + Blink);
center('"WARNING"',1,10,80); TextColor(White);
center('"" There are NO blocks currently loaded ! ";',1,12,80);
end;

For m:= 1 to NBlocks do   {for each block in the loop)
begin
With Block(m) do

begin
ClrScr; HighVideo;
Center('" Block Transfer Function Roots ";',1,2,80);
writeln; writeln(s);TextColor(white);
NomFact:= LeadDenCoeff/LeadNumCoeff;
write(' BLOCK  |

Block Gain = ";K1 NomFact:14:4);
TextColor(green);
write(' ZEROS: ");
end;
end;
end;

For i:=1 to NZeros do   {write all zeros)
begin
PosCounter := (i mod 2); {write 2 across screen)
If PosCounter = 1 then writeln;
LowVideo;
write('s[k,I,] = ";RealPartZero(i):Re3, +j ',ImagPartZero[i]:B:3);
write(' ");
end;
end;
end;
end;
end;
end;

For i:=1 to NPoles do   {repeat for poles)
begin
PosCounter := (i mod 2); {write 2 across screen)
If PosCounter = 1 then writeln;
LowVideo;
write('s[k,I,] = ";RealPartPole(i):Re3, +j ',ImagPartPole[i]:B:3);
write(' ");
end;
end;
end;
57     HighVideo;
58     msg ('Press any key to continue or [Shift] [PrtSc] for hardcopy.',1,24);
59     (Test keyboard buffer for shift key depressed (shift key changes buffer
60     contents by 1 or 2). If shift key is pressed, then a blank line is
61     written over the message 'Press any key...' so it will not print)
62     keyold := 00000:10473;  (check initial value of buffer)
63     not_erased := true;
64     Repeat
65     key := 00000:10473;  (repeatedly check buffer for changes)
66     if ((key=keyold + 1) or (key=keyold + 2)) and (not_erased) then
67     (if shift key pressed and message not previously erased then:
68     begin
69     begin
70     BlockXY(1,24); write(' :BO);  (overwrite message)
71     not_erased := false;
72     end;  
73     Until KeyPressed;  (if other key pressed then continue to next event)
74     end;  
75     end;  
76     end;  
77     With G_eq do  (repeat entire process for G-equivalent block)
78     Begin
79     ClrScr; HighVideo;
80     Center('### Loop Equivalent Block Transfer Function ###';,1,2,30);
81     writeln; writeln(s);TextColor(white);
82     writeln('G equivalent Block Block Gain = ','K;i:4;4);  
83    TextColor(green);
84     writeln('ZEROS: ');
85     For h=l to NZeros do
86     begin
87     PosCounter := (I mod 2) ;  
88     If PosCounter = 1 then writeln;
89     LowVideo;
90     write('
',j,realPartZeroUhB*8:3);  
91     writeln; writeln; writeln('POLES: ');
92     For I:=l to NPoles do
93     begin
94     PosCounter := (I mod 2) ;  
95     If PosCounter = 1 then writeln;
96     LowVideo;
97     write('s',j,realPartPole UhB*8:3);  

HighVideo;
msg ('Press any key to continue or [Shift] [PrtSc] for hardcopy.',1,24);
keyold := mem(0000h:1047); not_erased := true;

Repeat

key := mem(0000h:1047);
if ((key=keyold + 1) or (key=keyold + 2)) and (not_erased) then

begin
GotoXY(1,24); write('BO');
not_erased := false;

end;

Until KeyPressed;

end; (Procedure ShowRoots)
Procedure ShowPoly;

Var
  VertPos,
  HorizPos,
  PosCounter,
  I,J : Integer;
  Exponent : String[2];
  NormFact : Real;
  Character : Char;
  key=keyold : Integer;
  not_erased : Boolean;

-Begin

if NBlocks = 0 then (print warning if no blocks in memory)
  begin
    clrscr;TextColor(Red + Blink); center('WARNING',1,10,80); TextColor(White);
    center('*** There are NO blocks currently loaded ! ***',1,12,80);
  end;

For i= 1 to NBlocks do (print each block in loop)
  begin
    With Block[i] do
    Begin
      ClrScr; HighVideo; (screen title)
      Center('*** Coefficients of Block Polynomial ***',1,7,80);
      writeln; writeln(s); TextColor(Green);
      writeln('BLOCK ',i);
      NormFact := LeadDenCoeff/LeadNumCoeff; TextColor(White);
      writeln(' Block Gain = ', NormFact:14:4);
      (compute screen positions for coefficient display)
      VertPos= 6; LowVideo;
      For j= NZeros+1 downto 1 do
        begin
          PosCounter := j mod 3+1; (display 3 coeff across screen)
          HorizPos := PosCounter * 15;
          If PosCounter = 1 then VertPos := VertPos + 2;
          If i <> 1 then (write "+ s" with proper exponent)
            begin
              msq('s',HorizPos,VertPos);
              str(i-1:2,Exponent);
            end
        end
ensq (Exponent ^izPos+^VertPos-l);
end;
GaMV(HonzPa5-9,VertPas))
with LeadNiunCopff » Nu«Coef«n):8i3)i
-­end;
HighVideo;Hriteln; writelnlsl; Kriteln; (do same for denominator)
writelnCDENOHlNATGR:
VertPos := 16; LowVideo;
For I:= NPoles+1 downto 1 do
 begin
 J:=NPoles+I - 1;
 PosCounter := (J mod 4) + 1; HorizPos := PosCounter * 15;
 If PosCounter = 1 then VertPos := VertPos t 2;
 end; (if)
GotoXY(HorizPos-9,VertPos);
 writelnLLeadDenCoef» DenCoeff[I]:B:3);
end; (for)
msg ('Press any key to continue or [Shift] [PrtScJ for hardcopy.',l,24);
(check keyboard buffer for shift key (shift keys change buffer
contents by 1 or 2). If shift key depressed, then the
"Press any key..." message is erased from the screen so it
won't print.)
keyold := memEOOOO;10471; (check
initial
condition of buffer)
not_era5Bd := true; (initialize boolean)
Repeat
key := memOO00.'KMTlj (repeatedly check bufer)
if key=keyold + 1) or (key=keyold + 2)) and (not erased) then
begin
GotoXY(l,24); writeln(':80); (write blank line across message)
 not_era5Bd := false; (sets erased boolean)
end;
Until KeyPressed; (any other key will continue to next event)
With G_eq do (repeat entire process with G_equivalent block)
Begin
ClrScr; HighVideo;
Center(' *** Coefficients of Block Polynomial ***',1,2,80);
writeln; writeln(is); TextColor(Green);
 writeln('G-equivalent ');
NormFact := LeadDenCoeff/LeadNumCoeff; TextColor(White);
 writeln(' Block Gain = ', K+ NormFact14:4);
For I:= NZeros+1 downto 1 do
begin
J:=NZeros+1 - I;
PosCounter := (J mod 3)+1; HorizPos := PosCounter * 15;
If PosCounter = 1 then VertPos := VertPos + 2;
If I <> 1 then
begin
msg('s *',HorizPos,VertPos);
str(1-1:2,Exponent);
msg(Exponent,HorizPos+1,VertPos-1);
end;
GotoXY(HorizPos-9,VertPos);
end;

HighVideo; writeln; writeln;
writeln('DENOMINATOR: ');
VertPos := 16; LowVideo;
For I:= NPoles+1 downto 1 do
begin
J:=NPoles+1 - I;
PosCounter := (J mod 4)+1; HorizPos := PosCounter * 15;
If PosCounter = 1 then VertPos := VertPos + 2;
If I <> 1 then
begin
msg('s *',HorizPos,VertPos);
str(1-1:2,Exponent);
msg(Exponent,HorizPos+1,VertPos-1);
end; (if)
GotoXY(HorizPos-9,VertPos);
write(LeadDenCoeff * DenCoeff[11:8:3]);
end; (for)

msg ('Press any key to continue or [Shift][PrtSc] for hardcopy.',1,24);
keyold := men[0000:10471]; not_erased := true;
Repeat
key := men[0000:10473];
if ((key=keyold + 1) or (key=keyold + 2)) and (not_erased) then
begin
GotoXY(1,24); writeln (' :00');
not_erased := false;
end;
Until KeyFressed;
end; (with)

end; (Procedure ShowPoly)
(* User_Polynomial_Roots is a procedure to allow the user to input any *)
(* arbitrary polynomial and automatically output the roots of that *)
(* polynomial. It uses the standard input routines and RootFinder to *)
(* compute the roots of the polynomial: *)

Procedure User_Polynomial_Roots;
Var
PolyOrder,
PosCounter,
VertPos,
HorizPos,
I,K,code : Integer;
PolyCoeff,
RealPart,
ImagPart : PolyArray;
Exponent : String[2];

Begin
ClrScr; HighVideo; (TITLE)
Center('*** ROOTS OF A USER INPUT POLYNOMIAL ***','1,1,80);
Fillchar(s,100,'*'); s:= copy(s,1,80);
 writeln(s); writeln;
 writeln('Polynomial Coefficient Input: ');
 writeln;
 writeln;

msg('What is the Order of the Polynomial? ',10,0); (prompt user for poly)
input('N''','true,Fl,F10);
val(Answer,PolyOrder,code);
 writeln;

VertPos:=12; (position prompts on screen)
For I:= PolyOrder+1 downto 1 do
 begin
  K:=PolyOrder+1 - I;
  PosCounter := (K mod 6) + 1; HorizPos := PosCounter * 10;
  If PosCounter = 1 then VertPos := VertPos + 2;
  input(CNV^HorizPos-S^VertPos^true^l^l0);
  val(I answer 1 Pol yCoeff(1),code);
 end;
ClrScr; Center(''' Computing Roots — Please Wait '''','1,1,80); (wait msg)
RootFinder(PolyOrder,PolyCoeff,RealPart,ImagPart,0,0); (solve for roots)
ClrScr; HighVideo; Beep(150,150); (alert user when finished)
Center('### Roots of the Polynomial ###',1,2,80);  (root display)

writeln; writeln(); writeln;

For i=1 to PolyOrder do       (compute screen positions)

begin
PosCounter := (i mod 2) ;
If PosCounter = 1 then

begin

writeln; writeln;
end;

(display the roots)

LowVideo;

write('s(\',i,\') = ',RealPart(1i*Br3, '+' ,ImagPart(1i*Br3));

write(' ');     

HighVideo;

msg ('Press any key to continue or (Shift) (PrtSc) for hardcopy.','1,24);

repeat until keypressed;

end;
Cross-Reference & Block Listing

File: GRAPMENULINC

Date: 9/10/86

Page 1

{*******************************************************************************}
{** Graph Menu provides a window on screen and offers the user options to **}
{** make a title, print the graph, print the numbers from the graph, or **}
{** quit and return to the menu. **}
{*******************************************************************************}

Procedure Graph_Menu(TitleWindowName:Str20;var DumpGraphData,quit : boolean);

var
Line1, Line2, Line3 : string[40]; {graph prompt lines}

Procedure TitlePrompt; {Prompts for user supplied graph title}

begin
TextColor(White);
Center('*** Make Graph Title ***',1,2,80);

P111:= '1010040001-001000';
P121:= '1012000002-001000';
P131:= '1014000003-001000';

TextColor(yellow);
msg('Line 1:',1,10);
msg('Line 2:',1,12);
msg('Line 3:',1,14);

textcolor(green);
msg('Type in the title you wish for your graph.',6,20);

Input_handler('NO103',escape);

Line1:= copy(var[1],1,40);
Line2:= copy(var[2],1,40);
Line3:= copy(var[3],1,40);

end;

Procedure ShowTitle; {makes title block and writes title to block}

begin

copiescreen;
SetLineStyle(0);
DefineWindow(3,11,20,40,60);
DefineWorld(2,0,16,40,0);
SelectWorld(3); SelectWindow(3);
DefineHeader(3,TitleWindowName); {puts header on box}
SetBackground(0);
SetBadBrOn; DrawBorder;

DrawText(1,9,1,Line1);    {draw title supplied by user}
DrawText(1,11,1,Line2);
DrawText(1,13,1,Line3);
SetBreakOff; SetMessageOff;

repeat
read(kbd_ch);

{allow user to move title box anywhere on screen}

end.
File: GRMENU.INC  Cross-Reference & Block Listing  Date: 9/10/86  Page 2  

72: MoveHor(-4,true); (up arrow)  
75: MoveHor(-1,true); (left arrow)  
77: MoveHor(+1,true); (right arrow)  
80: MoveHor(+4,true); (down arrow)  

75: E-------------------------------------------end;  
76: E-------------------------------------------until ord(ch)= 13; (freeze box and continue with <return> key)  
77: E-------------------------------------------end;  
78: E--------------------------------------------end;  
79: E--------------------------------------------end;  
80: E-------------------------------------------begin  
81: DumpGraphData := False;  
82: copiescreen; SetLineStyle(0); (save underlying screen and display menu box)  
83: DefineWindow(4,0,20,35,90);  
84: DefineBox(4,0,20,0);  
85: DefineHeader(4, 'Graph Options Menu');  
86: SetMenuBox; SetBackground(0);DrawBorder;  
87: DrawText(M,4,1,0,'(P) Print Graph'); (display menu options)  
88: DrawText(M,7,1,0,'(T) Make Title Block');  
89: DrawText(M,10,1,0,'(D) Print Table of Numbers');  
90: DrawText(M,13,1,0,'used to Generate Graph');  
91: DrawText(M,17,1,0,'(O) Return to Main Menu');  
92: B-------------------------------------------repeat  
93: B--------------------------------------------case ch of  
94: B--------------------------------------------'P': begin  
95: B--------------------------------------------  
96: B--------------------------------------------  
97: B--------------------------------------------end;  
98: B--------------------------------------------'T': begin  
99: B--------------------------------------------  
100: B--------------------------------------------  
101: B--------------------------------------------  
102: B--------------------------------------------end;  
103: B--------------------------------------------'N': begin  
104: B--------------------------------------------  
105: B--------------------------------------------  
106: B--------------------------------------------end;  
107: if ch = 'Q' then Quit := true  
108: else Quit := false;  
109: B-------------------------------------------end;  
110: B-------------------------------------------end;  
111: B-------------------------------------------end;
Procedure PrintGraphData; (dumps rootlocus data to printer)
  var
  LineCount, i, j, k, l: integer;
  FirstPage: boolean;
  List: text;

Procedure NewPage;
begin
  if not(FirstPage) then
    writeln(list, chr(121));  {formfeed for all but first page}
  writeln(list, ' POLE LOCATIONS');  {write titles}
  if StepA then writeln(list, 'A = ', amin:10:3)
  else writeln(list, 'B = ', bmin:10:3);
  if StepA then writeln(list, ' B Pole')
  else writeln(list, ' A Pole');
  writeln(list, ' REAL', ' IMAGINARY');
  for i := 1 to 60 do writeln(list, ' ');  {draw a line across page}
  writeln(list);  
  FirstPage := False;
end;

begin
  FirstPage := true;
  LeaveGraphic;  {display user prompts on screen}
  clrscr;
  Center('TURN ON FAINTER AND ALIGN PAPER', 1, 10, 80);
  TextColor(green);
  writeln('press <F> to print, <F> to quit print', 1, 13);
  TextColor(WHITE);
  writeln('File option prints numbers to a file named "THOPARAH.NUH", 1,17);
  writeln('on the current drive. Browse this file off-line using DOS "type" command', 1,18);
repeat
  Readln(kbd, ch);  {accept user input}
  if (ch = 'F') or (ch = 'f') or (ch = 'P') or (ch = 'p') then
    begin
      if (ch = 'F') or (ch = 'f') then
        begin
          assign(list, 'TwoParam.NUH');  {open a file for writing}
          rewrite(list);
          writeln(list);  
        end
      else
        begin
          assign(list, 'List.');  {open the printer for printing}
          writeln(list);  
        end;
    end;
  writeln('NewPage');
  LineCount := 0;
  a := amin; b := bmin;  {compute 2-parameter values for printing}
  for j := 1 to 5 do
    begin
      "..."
for k := 1 to 50 do  { smoothly move through the other parameter)
    begin
        (range in 50 increments)
        for i:= 1 to Order + 1 do  { compute each coefficient value)
            begin
                infix_to_polish(InfixArray[i],polish);
                Compute_Polish(polish,a,b,EvalArray[i]);
            end;
        end;
    RootFinder(Order,EvalArray,RealPart,ImagPart,0,0);  { find eq roots)
    if stepA then writeln(list,' ',b:10:3)
    else writeln(list,' ',a:10:3);
    LineCount := Linecount + 1;
    For i:=1 to Order do  { write Roots to page printer)
        begin
            writeln(list,' ',RealPart[i]:10:3,
            ImagPart[i]:10:3);
            Linecount := LineCount + 1;
            If LineCount mod 55) = 0 then NewPage;
        end;
    if stepA then  { increment one parameter)
        b := b + increm;
    else
        a := a + increm;
    end; (for)
    if stepA then  { increment stepping parameter)
        begin
            a := a + DeltaStep;
            b := bmin;
        end (next step)
    else
        begin
            b := b + DeltaStep;
            a := amin;
        end;
    end; (for)
    until ch in ['F','f','D','d','Q','q','P','p'];
    close(list);
    EnterGraphic;  { brings back graph to screen)
    swapscreen;
end;

(* *************** END of PrintGraphData Procedure *********************** *)
LIST OF REFERENCES


BIBLIOGRAPHY


INITIAL DISTRIBUTION LIST

<table>
<thead>
<tr>
<th>No. Copies</th>
<th>Name and Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Defense Technical Information Center&lt;br&gt;Cameron Station&lt;br&gt;Arlington, Virginia 22304-6145</td>
</tr>
<tr>
<td>2</td>
<td>Library, Code 0142&lt;br&gt;Naval Postgraduate School&lt;br&gt;Monterey, California 93943-5002</td>
</tr>
<tr>
<td>2</td>
<td>Department Chairman, Code 62&lt;br&gt;Naval Postgraduate School&lt;br&gt;Monterey, California 93943</td>
</tr>
<tr>
<td>15</td>
<td>Professor G. J. Thaler, Code 62TR&lt;br&gt;Naval Postgraduate School&lt;br&gt;Monterey, California 93943</td>
</tr>
<tr>
<td>1</td>
<td>Professor A. Gerba, Code 62GZ&lt;br&gt;Naval Postgraduate School&lt;br&gt;Monterey, California 93943</td>
</tr>
<tr>
<td>1</td>
<td>Professor D. E. Kirk, Code 62KI&lt;br&gt;Naval Postgraduate School&lt;br&gt;Monterey, California 93943</td>
</tr>
<tr>
<td>1</td>
<td>Professor R. D. Strum, Code 62ST&lt;br&gt;Naval Postgraduate School&lt;br&gt;Monterey, California 93943</td>
</tr>
<tr>
<td>1</td>
<td>Professor J. H. Duffin, Code 62DN&lt;br&gt;Naval Postgraduate School&lt;br&gt;Monterey, California 93943</td>
</tr>
<tr>
<td>1</td>
<td>Professor H. A. Titus, Code 62TS&lt;br&gt;Naval Postgraduate School&lt;br&gt;Monterey, California 93943</td>
</tr>
<tr>
<td>1</td>
<td>LT. Roy L. Wood&lt;br&gt;Route 1, Driftwood Cove #20&lt;br&gt;Jefferson, Texas 75657</td>
</tr>
<tr>
<td>1</td>
<td>Park, Pal Man&lt;br&gt;SMC 2400&lt;br&gt;Naval Postgraduate School&lt;br&gt;Monterey, California 93943</td>
</tr>
</tbody>
</table>
12. Mr. T. M. Slaughter
2028 Green Street
San Francisco, California 94123