Some SISO Transfer Function Facilities in CTRL-C

Lilja, Mats

1986

Document Version:
Publisher's PDF, also known as Version of record

CODEN: LUTFD2/(TFRT-7325)/1-019/(1986)

Some SISO Transfer Function Facilities in CTRL-C

Mats Lilja

Department of Automatic Control
Lund Institute of Technology
August 1986

Department of Automatic Control
Lund Institute of Technology
P.O. Box 118
S-221 00 Lund Sweden

Document name
Report

Date of issue
August 6, 1986

Document Number
CODEN: LUTFD2/(TFRT-7325)/1-019/(1986)

Author(s)
Mats Lilja

Supervisor

Sponsoring organisation

Title and subtitle
Some SISO Transfer Function Facilities in CTRL-C

Abstract

In the matrix manipulation language CTRL-C there exist some functions for analysis of linear systems. These operates however only on state space representations. This report presents some functions that use (SISO) transfer function representations instead. Nyquist curve plotting of systems with time delay, pole placement design and root locus plotting are covered.

Key words

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title

ISBN

Language
English

Number of pages
19

Recipient's notes

Security classification

Some SISO transfer function facilities in CTRL-C

Background

The matrix manipulation language CTRL-C contains some functions for handling polynomials. Polynomials are represented as row vectors and polynomial multiplication is implemented as convolution of vectors. There are however presently no control system tools that use a transfer function representation. In this report some CTRL-C-functions are listed, which are useful for SISO systems given on transfer function form. This includes functions for calculation and plotting of frequency response, solving of DAB (Diophantine-Aryabhatta-Bezout) equations and plotting of root loci.

Frequency response plotting

The predefined library SYSLIB in CTRL-C contains some functions for calculating the frequency response of a system given on state space form (A, B, C, D). The function NYQU calculates the real and imaginary parts of $G(i\omega) = C(i\omega I - A)^{-1}B + D$ for a given vector of frequencies ω and the function BODE gives the magnitude and phase of $G(i\omega)$. The slightly more general function FREQ gives $G(s) = C(sI - A)^{-1}B + D$ as a vector of complex numbers where s is a vector of arbitrary complex numbers. If the system instead is represented by its transfer function, it is possible to convert it to state space representation and then use the already existing functions mentioned above. However it is more tempting to take a shortcut and directly calculate the frequency response from the polynomials of the transfer function. A time delay could also be included in the transfer function. The resulting values of $G(i\omega)$ are stored either in the 'Nyquist' way (real and imaginary parts) or in the 'Bode' way (magnitude and phase). The 'Bode' representation is often to be preferred, at least when the time delay is large. The 'Nyquist' representation looses in this case information about the phase. One drawback with the 'Bode' alternative is however that the phase has to be checked and corrected when passing odd multiples of π (in case the principal branch of the logarithm is preferred). This checking may cause unnecessary long computation times at least if you just want to plot the Nyquist curve.

The data are stored in a 100×3 matrix with frequencies in the first column and the corresponding values of the transfer function in the second and third columns (in either of the representations above). Different plot functions, which operate on such matrices, are availible. The transfer function is assumed to be of the form

$$G(s) = \frac{B(s)}{A(s)} e^{-\tau s}$$

and the calculation of the frequency response (in 'Nyquist' form) is performed by the following function call

```
[>  fr1 = frny(a,b,tau,lgw1,lgw2);
```

where a and b are row vectors representing the polynomials A and B. The time delay is tau and the lower and upper limit for the 100 logarithmically spread frequencies are 10**lgw1 and 10**lgw2 respectively. The resulting frequency response can now be plotted by the function call

```
[>  npl1(fr1)
```

which plots the Nyquist curve in a fresh diagram. For each tenth frequency, the corresponding point on the Nyquist curve is marked. These frequencies are easily displayed:

```
[>  fr1(10:10:100,1)
```

If fr2 is defined then the call

```
[>  npl(fr2)
```

plots the Nyquist curve corresponding to fr2 in the same diagram as fr1. The command

```
[>  npl2(fr1,fr2)
```

clears the screen and plots the Nyquist curves corresponding to fr1 and fr2 in a fresh diagram. The functions APL, APL1 and APL2 work in a similar way but giving the amplitude curve instead and the functions PPL, PPL1 and PPL2 gives the phase curves (with the sorting mentioned above). Sometimes it is of interest in stability theory to plot the Popov curve

$$(\text{Re } G(i\omega), \omega \,\text{Im } G(i\omega))$$

This is done by using the functions POP, POP1 and POP2. As a special feature there is a possibility to compute the frequency response for arbitrary transfer functions. This is done by using the 'text macro' facility in CTRL-C. The commands

```
[>  g = 'exp(-sqrt(s));';

[>  fr = frfu(g,-2,2);
```

```
[>  npl1(fr)
```

plots the Nyquist curve of $G(s) = e^{-\sqrt{s}}$ for frequencies ω ranging from 0.01 to 100 rad/s. By using the convention to let the complex variable name in the transfer function be s, it is possible to pass over the transfer function as the string **g** to the function FRFU where it is evaluated for $s = i\omega$ and stored in the output matrix:

```
for j=1:nw,..
  s = sqrt(-1)*w(j);..
  ]g[;..
  fr(j,:)=[w(j) real(ans) imag(ans)];..
end;
```

The evaluation of the text string **g** is done by the reversed square brackets, **]g[** and the result is stored in the default variable **ans**.

Solving DAB equations

Many control problems involves solving of DAB equations

$$AX + BY = C$$

where A, B and C are known polynomials and X and Y are unknown polynomials. The polynomials A and X are assumed to be monic. Let

$$\begin{aligned} A(s) &= s^n + a_1 s^{n-1} + \cdots + a_n \\ B(s) &= b_1 s^{n-1} + b_2 s^{n-2} + \cdots + b_n \\ C(s) &= s^l + c_1 s^{l-1} + \cdots + c_l \end{aligned}$$

where $n > m$. The unique solution for which $\deg Y = \deg A - 1$ is chosen ($\deg A + \deg X = \deg C$). The coefficents c_j, $j = 1, 2, \ldots, l$ are found by solving the equations

$$\begin{pmatrix} 1 & 0 & \cdots & 0 & 0 & \cdots \\ a_1 & 1 & \cdots & 0 & 0 & \cdots \\ \vdots & \vdots & & \vdots & \vdots & \\ a_{l-2n+2} & a_{l-2n+1} & \cdots & b_1 & 0 & \cdots \\ a_{l-2n+3} & a_{l-2n+2} & \cdots & b_2 & b_1 & \cdots \\ \vdots & \vdots & & \vdots & \vdots & \\ 0 & a_n & \cdots & 0 & 0 & \cdots \end{pmatrix} \begin{pmatrix} 1 \\ x_1 \\ \vdots \\ x_{l-n} \\ y_0 \\ y_1 \\ \vdots \\ y_{n-1} \end{pmatrix} = \begin{pmatrix} 1 \\ c_1 \\ c_2 \\ \vdots \\ c_l \end{pmatrix}$$

(for the special case $l < 3n - 2$). The matrix in the left member is nonsingular iff A and B have no common root. The function DIOP solves the DAB equation above by solving this system of equations. This is done by the command

```
[>  [x,y] = diop(a,b,c)
```

Consider the pole-placement problem: Choose polynomials R, S and T s.t. the control law

$$R(p)u = -S(p)y + T(p)r$$

($p := \frac{d}{dt}$) applied to the system

$$A(p)y = B(p)u$$

gives the closed loop system

$$A_m(p)y = B_m(p)r$$

This is equivalent to solving the DAB equation

$$AR_1 + B^-S = A_mA_o$$

where $B = B^-B^+$. The regulator is then given by

$$R = R_1B^+$$
$$S = S$$
$$T = t_0A_oB'_m$$
$$B_m = B'_mB^-$$

The constant t_0 is chosen s.t. that the closed loop system has a stationary gain of 1. The function RSTC uses DIOP to perform this:

```
[>  [r,s,t] = rstc(a,bplu,bmin,am,bmp,ao,1)
```

with obvious notation. In case integral action is wanted in the regulator, the function call should be modified to

```
[>  [r,s,t] = rstc(a,bplu,bmin,am,bmp,ao,[1 0])
```

A simpler version of RSTC is CRST, which keeps the old B-polynomial:

```
[>  [r,s,t] = crst(a,b,am,ao,1)
```

A simple simulation can be made by calling the function YUCL, which uses the CTRL-C-function SIMU. The input consists of a unit step in the reference signal at time $t = 0$ and a negative unit step in the load disturbance at half the simulation time. The result is stored in a 100×3 matrix with the time t,the process output y and the controller output u in the three columns respectively. The signals can be plotted by either YUPL (new diagram) or YUP (old diagram). YUP uses the global variables USCA and YSCA to remember the scalings for the old u- and y-plots. The command

```
[> clo = yucl(a,b,r,s,t,40);

[> yupl(clo);
```

simulates the closed loop system in 40 seconds, with a step in the reference at $t = 0$ and a negative step in the load disturbance at $t = 20$, and then plots u and y.

Root locus plots

A typical problem that arises in control is to display the roots of the equation

$$f(s,k) := A(s) + kB(s) = 0$$

as a function of the real, positive variable k. The common way to do this is to just plot the roots for equidistant values of k. This gives however a very sparse look of the root locus near multiple roots. A way to avoid this is to detect closeness to multiple roots and decrease the increment in k. This can be done by applying the implicit function theorem to f (let prime denote derivation w.r.t. s):

$$\frac{ds}{dk} = -\frac{\frac{\partial f}{\partial k}}{\frac{\partial f}{\partial s}} = -\frac{B(s)}{A'(s) + kB'(s)}$$

for those values of s and k for which the partial derivative of f w.r.t. s does not vanish (this derivative vanishes precisely when $A(s) + kB(s) = 0$ has a multiple root). If a maximum step length in s, say Δs, is specified, the step length in k can be chosen as

$$|\Delta k| = \min_i \left| \frac{A'(s_i) + kB'(s_i)}{B(s_i)} \right| |\Delta s|$$

where the minimum is taken w.r.t. the roots for the present value of k. This gives a decent step length near multiple roots but, of course, the maximum step length Δs cannot be guaranteed since this is just a linear extrapolation. The command

```
[> rl = rloc([1 3 3 2 0],[1 3 2 1],0,2,0.1);
```

plots the root locus for

$$A(s) + kB(s) = s^4 + 3s^3 + 3s^2 + 2s + k(s^3 + 3s^2 + 2s + 1) = 0$$

for $0 \leq k < 2$ and with a prespecified maximum distance $\Delta s \leq 0.1$ (fig.1). The zeros of B (the finite end points of the root locus) are marked as 'octagons'

and multiple roots are, if any, plotted as 'star bursts' (CTRL-C terminology).

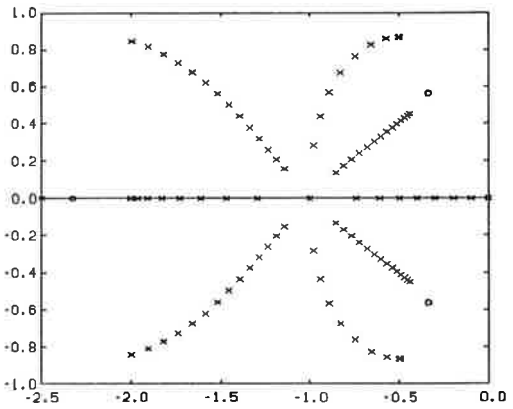

Fig.1 Root locus for $s^4 + 3s^3 + 3s^2 + 2s + k(s^3 + 3s^2 + 2s + 1) = 0$

The multiple roots are found by considering the identity

$$\begin{pmatrix} A(s) & B(s) \\ A'(s) & B'(s) \end{pmatrix} \begin{pmatrix} 1 \\ k \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

The non-trivial solutions to this equation are found by solving $A(s)B'(s) - A'(s)B(s) = 0$. Selecting the roots s_i, for which $k = -\frac{A(s_i)}{B(s_i)}$ is real, gives all multiple roots of $A(s) + kB(s) = 0$.

An example

Consider the system with transfer function

$$G(s) = \frac{-s+2}{(s+1)(s^2+2s+10)}$$

For convenience we store the coefficients in **a** and **b**:

```
[>  b = [-1 2]; a = conv([1 1],[1 2 10]);
```

The Nyquist curve for the systems is obtained for frequencies $10^{-2} < \omega < 10^2$ by the commands

```
[>  fr1 = frny(a,b,0,-2,2);

[>  npl1(fr1)
```

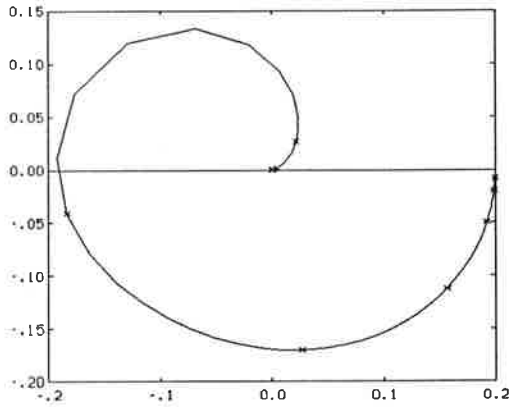

Fig.2 Nyquist curve for the open loop system

A pole placement design for which the closed loop poles are placed at $-4, -2 \pm 2i$ is done by entering the following commands

```
[>  i = sqrt(-1);

[>  am = real(poly([-3 -2+2*i -2-2*i]))';

[>  ao = poly(-5*ones(1,2))';

[>  [r,s,t] = crst(a,b,am,ao,1)
```

The two observer poles are placed in -10. Since no pole-zero cancellation is recommendable in this case, the simpler pole-placer **CRST** is used. The step response for the closed loop system can now be plotted:

```
[>  cl1 = yucl(a,b,r,s,t,20);

[>  yupl(cl1)
```

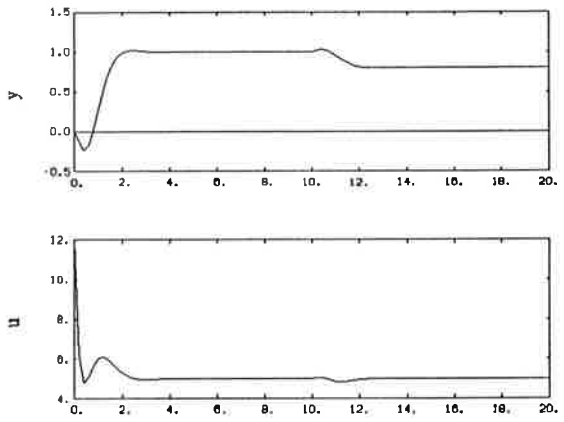

Fig.3 Simulation of closed loop system (without integration)

The step response for the first 20 seconds will now appear in the upper diagram and the lower diagram will show the output from the controller. Notice that a load disturbance is introduced at $t = 10$. A new design, for which the regulator dynamics contains an integrator, is easily obtained:

```
[>  ao = poly(-5*ones(1,3))';

[>  [r,s,t] = crst(a,b,am,ao,[1 0]);
```

where the degree of the observer polynomial had to be increased by one (all observer poles still in -10). To check the correctness of this design one can for example calculate the poles of the closed loop system:

```
[>  clp='ar=conv(a,r);bs=conv(b,s);..
arbs=ar+[0*ones(1,max(size(ar)-size(bs))) bs];..
bt=conv(b,t);clpo=root(arbs),';

[>  ]clp[
```

where the CTRL-C 'text macro' facility is demonstrated. The string variable `clp` can be evaluated by enclosing it with reversed square brackets (`]clp[`), which from now on is a convenient way to display the closed loop poles. To get some idea of how robust this design is you can plot the Nyquist curve for the loop transfer function:

```
[>  frloop = frny(conv(a,r),conv(b,s),0,0,2);
```

```
[>  npl1(frloop)
```

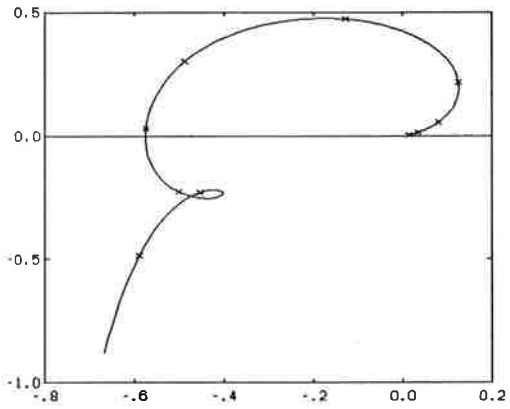

Fig.4 Nyquist curve of loop transfer function

Another possibility to check the robustness is to plot a root locus w.r.t. the loop gain (solving $A(s)R(s) + kB(s)S(s) = 0$):

```
[>  rl = rloc(conv(a,r),conv(b,s),0,2,0.5);
```

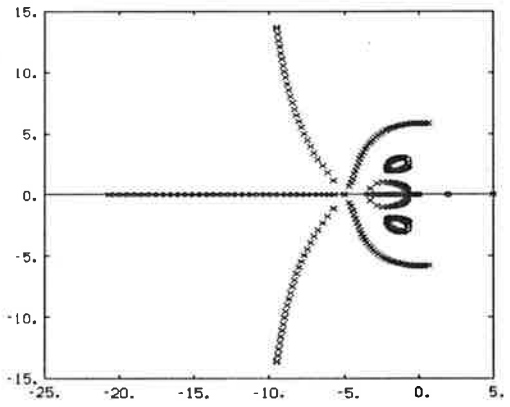

Fig.5 Root locus for $A(s)R(s) + kB(s)S(s) = 0$

The 'loop gain' k varies from 0 to 2 (a loop gain of one corresponds here to the actual design) and the maximum 'step length' in the s-plane is specified to be 0.5 (this may not be satisfied near multiple roots).

Summary of functions

Frequency response

FRNY	Computes the frequency response of a system with time delay
FRFU	Computes the frequency response for arbitrary transfer functions
NPL1	Plots one Nyquist curve in a new diagram
NPL2	Plots two Nyquist curves in a new diagram
NPL	Plots one Nyquist curve in an old diagram
APL1	Plots one amplitude curve in a new diagram
PPL1	Plots one phase curve in a new diagram
POP1	Plots one Popov curve in a new diagram

DAB equations

DIOP	Solves the DAB equation $AX + BY = C$
RSTC	Solves the pole placement problem
CRST	Simple version of RSTC
YUCL	Calculates the step response of the closed loop system
YUPL	Plots one step response in a new diagram
YUP	Plots one step response in an old diagram

Root locus plots

RLOC	Calculates and plots root locus

Summary of function calls

`fr = frny(a,b,tau,lgw1,lgw2)`

Computes the frequency response of the system with transfer function

$$G(s) = \frac{B(s)}{A(s)} e^{-\tau s}$$

for 100 logarithmically spread frequencies. `lgw1` and `lgw2` are the base 10 logarithms of the lower and upper limit for the frequencies. The output matrix `fr` contains the frequencies in the first column and real and imaginary part of $G(i\omega)$ in the second and third columns.

`fr = frfu(g,lgw1,lgw2)`

Works similar to FRNY but the transfer function is specified as a string (e.g. `g = '1/(1+sqrt(s))'`) using the name convention `s` for the complex variable.

`npl1(fr)`

Plots the Nyquist curve (in a fresh diagram) from the data in the matrix `fr`.

`npl2(fr1,fr2)`

Similar to `NPL1` but for two Nyquist curves.

`npl(fr1,fr2)`

Similar to `NPL1` but without erasing the old diagram.

`apl1(fr)`

Plots the amplitude curve from data in `fr`.

`ppl1(fr)`

Plots the phase curve from data in `fr`.

`pop1(fr)`

Plots the Popov curve from data in `fr`.

`[x,y] = diop(a,b,c)`

Solves the DAB equation $AX + BY = C$. The solution corresponding to minimal degree of Y is chosen.

`[r,s,t] = rstc(a,bplus,bminus,am,bmprim,ao,ar)`

Solves the pole placement problem

$$\begin{aligned} A(s)R_1(s) + B^-(s)S(s) &= A_m(s)A_o(s) \\ T(s) &= t_0 B'_m(s)A_o(s) \\ R(s) &= B^+(s)A_r(s)R_1(s) \end{aligned}$$

where the minimal-degree-of-S solution is chosen.

`[r,s,t] = crst(a,b,am,ao,ar)`

Simplified version of `RSTC`, with $B^+(s) = B(s)$, $B^-(s) = 1$ and $B'_m(s) = 1$.

`clo = yucl(a,b,r,s,t,time)`

Simulates the closed loop system

$$\begin{aligned} A(p)y &= B(p)(u + d) \\ R(p)u &= T(p)r - S(p)y \end{aligned}$$

with a unit step in r at time $t = 0$ and a negative unit step in d at $t =$ `time`$/2$ up to time `time`. The 100×3 matrix `clo` contains t, y and u in its three columns.

`yupl(clo)`

Plots the process output y in the upper diagram and the control signal u in the lower diagram with the data from `clo`. The screen is erased before plotting.

`yup(clo2)`

Same as `yupl` but the screen is not erased and the old scales are used. Two global variables must be defined – `uscale` and `yscale` (by using the `glob` command). Their start values must be a 3×2 matrices (take simply `0*ones(3,2)`).

`rl = rloc(a,b,k1,k2,dsmax)`

Calculates and plots the root locus for

$$A(s) + kB(s)$$

The start value for k is **k1** and the maximal k-value is **k2**. An attempt is done to keep the modulus of the increment Δs less than **dsmax** by choosing the increment Δk according to

$$\Delta k = \min_i \left| \frac{A'(s) + kB'(s)}{B(s)} \Delta s_{max} \right|$$

(linear extrapolation in k). The open loop zeros are marked by "octagons" and the open loop poles by "fancy diagonal crosses". Multiple poles are computed as the zeros of $A'(s)B(s) - A(s)B'(s)$. The k:s are stored in the first column of **rl** and the corresponding roots in the other columns. The open loop zeros, open loop poles and multiple poles are not stored.

Appendix: Listing of the functions

```
[fr]=frny(Apoly,Bpoly,Tdelay,lgw1,lgw2);
// Computes the frequency response for the transfer function
//
//                          Bpoly(s)  -Tdelay s
//                 G(s) = --------- e
//                          Apoly(s)
//
// for frequencies (w) logarithmically spaced between 10**(lgw1)
// and 10**(lgw2). The values are stored in columns as
//
//               [  w   Re G(iw)   Im G(iw) ]
//
  na = size(Apoly)*[0;1];
  nb = size(Bpoly)*[0;1];
  i = sqrt(-1);
  ln10 = log(10);
  w = exp((lgw1:((lgw2-lgw1)/99):lgw2)*ln10);
  iw = i*w;
  nw = size(iw)*[0;1];
  one = ones(1,nw);
  avec = Apoly(1)*one;
  for j=2:na,..
    avec = iw*.avec + Apoly(j)*one;..
  end;
```

```
  bvec = Bpoly(1)*one;
  for j=2:nb,..
    bvec = iw*.bvec + Bpoly(j)*one;..
  end;
  G = bvec/.avec;
  Gd = exp(-Tdelay*iw)*.G;
  Re = real(Gd);
  Im = imag(Gd);
  fr = [w' Re' Im'];
////////////////////////////////////////////////
[fr] = frfunc(func,lgw1,lgw2);
// Computes the frequency response for a system with transfer
// function specified by the string 'func' as in the following
// example:
//
// [>  func = 'exp(-sqrt(s))';
//
// [>  fr = frfunc(func,-2,2);
//
// The resulting 100 x 3 matrix 'fr' will in this case look like
//
//           [  w   Re (exp(-sqrt s))   Im (exp(-sqrt s))  ]
//
// where 1e-2 <= w <= 1e2.
//
  i = sqrt(-1);
  w = logspace(lgw1,lgw2);
  iw = i*w;
  for j=1:100,..
    s = iw(j); f = [func ';']; ]f[ ; G(1,j) = ans;..
  end;
  Re = real(G);
  Im = imag(G);
  fr = [w' Re' Im'];
////////////////////////////////////////////////////
[]=npl1(frf);
  eras;
  plot('scale');
  nw = size(frf)*[1;0];
  i10 = min([10,nw])*(1:round(max([10,nw])/10 - 0.5));
  plot(frf(:,2),frf(:,3));
  ppp = plot('peek');  plot(ppp,'scale');
  plot(frf(i10,2),frf(i10,3),'point=1');
  plot('scale');
```

```
////////////////////////////////////////////////////
[]=npl2(frf1,frf2);
  eras;
  plot('scale');
  nw1 = size(frf1)*[1;0];
  i101 = min([10,nw1])*(1:round(max([10,nw1])/10 - 0.5));
  nw2 = size(frf2)*[1;0];
  i102 = min([10,nw2])*(1:round(max([10,nw2])/10 - 0.5));
  plot(frf1(:,2),frf1(:,3),frf2(:,2),frf2(:,3));
  ppp = plot('peek');  plot(ppp,'scale');
  plot(frf1(i101,2),frf1(i101,3),'point=1',...
       frf2(i102,2),frf2(i102,3),'point=9');
  plot('scale');

////////////////////////////////////////////////////
[]=npl(frf);
  ppp = plot('peek');
  plot(ppp,'scale');
  nw = size(frf)*[1;0];
  i10 = 10*(1:round(nw/10 - 0.5));
  plot(frf(:,2),frf(:,3));
  plot(frf(i10,2),frf(i10,3),'point=2');
  plot('scale');
////////////////////////////////////////////////////
[]=apl1(frf);
  eras;
  plot('scale');
  w = frf(:,1);
  ln10 = log(10);
  lgw = log(w)/ln10;
  lga = 0.5*log(frf(:,2)*.frf(:,2)+frf(:,3)*.frf(:,3))/ln10;
  plot(lgw,lga);
////////////////////////////////////////////////////
[]=ppl1(frf);
  eras;
  plot('scale');
  i = sqrt(-1);
  w = frf(:,1);
  n = size(w);
  ln10 = log(10);
  lgw = log(w)/ln10;
  phase = imag(log(frf(:,2)+i*frf(:,3)))*180/pi;
  k = 1;
  while k<n;..
```

```
    k=k+1;..
    if phase(k)-phase(k-1)>180;..
      phase(k:n) = phase(k:n) - 360*ones(k:n,1)';..
    end;..
    if phase(k)-phase(k-1)<-180;..
      phase(k:n) = phase(k:n) + 360*ones(k:n,1)';..
    end;..
  end;
  plot(lgw,phase);
////////////////////////////////////////////////////
[]=pop1(frf);
  eras;
  plot('scale');
  w = frf(:,1);
  nw = size(w)*[1;0];
  i10 = 10*(1:round(nw/10 - 0.5));
  plot(frf(:,2),w*.frf(:,3));
  ppp = plot('peek');  plot(ppp,'scale');
  plot(frf(i10,2),w(i10)*.frf(i10,3),'point=1');
  plot('scale');
////////////////////////////////////////////////////
[x,y] = Diophantine(a,b,c);
// Solves the Diophantine equation
//
//              AX + BY = C
//
// where A,B,C,X and Y are polynomials with deg Y = deg A - 1
//
  na = max(size(a));
  nb = max(size(b));
  nc = max(size(c));
  ny = na - 1;
  test = ny + nb - nc - 1;
  if test > 0 then,..
    c = [0*ones(1,test) c];..
    nc = nc + test;..
  end;
  nx = nc - ny;
  ex = eye(nx);
  ey = eye(ny);
  for j=1:nx, mx(j,:)=conv(a,ex(j,:));
  for j=1:ny, my(j,:)=conv(b,ey(j,:));
  m = [mx ; [0*eye(ny,nx-nb+1) my]];
  xy = c/m;
```

```
  x = xy(1:nx);
  y = xy(nx+1:nc);
////////////////////////////////////////////////////
[r,s,t]=rstc(a,bplus,bminus,am,bm1,ao,ar);
// Solves the pole placement problem
//
//              + -
//             B B                  T         S
//       y  =  ---- u  ,    u  =  --- y  -  --- y
//              A                  R   r     R
//
//              -
//             B B
//                m1
//       y  =  -----  y   ,  T  =  t A
//              A      r           0 o
//               m
//
// where t0 is a normalization constant such that
//
//     lim  y(t) = y
//    t->oo         r
//
// when the reference signal yr is a step (continuous time).
// The polynomial Ar consists of factors to be forced into R
// (e.g. Ar(s) = s  introduces integration in the controller)
//
 a1 = conv(a,ar);
 c = conv(am,ao);
 bm = conv(bminus,bm1);
 nam = size(am)*[0;1];
 nbm = size(bm)*[0;1];
 t0 = am(nam)/bm(nbm);
 t = t0*conv(ao,bm1);
 [r1,s]=diophant(a1,bminus,c);
 r = conv(conv(r1,ar),bplus);
 s = s/r(1);
 t = t/r(1);
 r = r/r(1);
////////////////////////////////////////////////////
[r,s,t]=crst(a,b,am,ao,ar);
 a1 = conv(a,ar);
 c = conv(am,ao);
 nam = size(am)*[0;1];
```

```
 nb = size(b)*[0;1];
 t0 = am(nam)/b(nb);
 t = t0*ao;
 [r1,s]=diophant(a1,b,c);
 r = conv(r1,ar);
 s = s/r(1);
 t = t/r(1);
 r = r/r(1);
////////////////////////////////////////////////////
[tyu] = yucl(apol,bpol,rpol,spol,tpol,time);
// Computes the step response from reference signal and
// load disturbance to controller output and measured
// output
//
  [a,b,c,d] = tf2ss(bpol,apol);
  [ff,gg,hh,kk] = tf2ss([-spol ; tpol],rpol);
  f=ff';g=hh';h=gg';k=kk';
  if (size(g)-size(k))*[0;1]=0,..
    gr = g(:,2);..
    g = g(:,1);..
  else,..
    gr = 0;..
    g = 0;..
  end;
  kr = kk(2);
  k = kk(1);
  if abs(1-d*k) < 1e-15 then disp('Ill conditioned problem');return;
  dk1 = 1/(1-d*k);
  aa = [a+b*c*k*dk1 b*h*dk1 ; g*c*dk1 f+g*h*d*dk1];
  bb = [dk1*b dk1*kr*b ; dk1*d*g gr+dk1*d*kr*g];
  cc = [k*dk1*c h*dk1 ; dk1*c d*dk1*h];
  dd = [k*dk1*d kr*dk1 ; dk1*d d*dk1*kr];
  t = (0:0.01:1)*time;
  ref = ones(0:100);
  dist = [0*ones(0:49) -ones(50:100)];
  uy = simu(aa,bb,cc,dd,[ dist ; ref ],t);
  u = uy(1,:);
  y = uy(2,:);
  tyu = [t' y' u'];
////////////////////////////////////////////////////
[]=yupl(yu);
  eras;
  plot('scale');
  window('211');
```

```
  plot(yu(:,1),yu(:,2));
  ylabel('y','1');
  yscale = plot('peek');
  plot('scale');
  window('212');
  plot(yu(:,1),yu(:,3));
  uscale = plot('peek');
  ylabel('u','1');
////////////////////////////////////////////////////
[]=yup(yu);
  window('211');
  plot(yscale,'scale');
  plot(yu(:,1),yu(:,2));
  window('212');
  plot(uscale,'scale');
  plot(yu(:,1),yu(:,3));
  plot('scale');
////////////////////////////////////////////////////
[Locus] = rloc(Apol,Bpol,K1,K2,dsmax);
////////////////////////////////////////////////////
// Special root locus routine, which calculates   //
// the increment in K such that abs(ds) < dsmax.  //
// K1 and K2 are lower and upper limit for K resp.//
// The roots are stored and the corresponding     //
// K-values are stored in Locus. The root locus,  //
// the zeros and the open loop poles are plotted. //
// Other CTRLC-functions that must be included    //
// are EVAL and DERI.                             //
////////////////////////////////////////////////////
  d = (size(apol) - size(bpol))*[0;1];
  a = apol;
  b = [0*(1:d) bpol];
  da = deri(a);
  db = deri(b);
  k = k1;
  epsi = 1e-10;
  i = 1;
  while k<=k2 ;..
    kvec(i) = k;..
    ri = root(a+k*b)';..
    rvec(i,:) = ri;..
    fs = eval(da+k*db,ri);..
    fk = eval(b,ri);..
```

```
    dk = dsmax*min(abs(fs/.fk))+epsi;..
    k =  k + dk;..
    i =  i + 1;..
  end;
  locus = [kvec rvec];
  poles = root(a);
  rl = rvec(:);
  dab = conv(da,b);
  adb = conv(a,db);
  if max(size(b))>1, mpol = dab - adb; else, mpol = dab;
  multroot = root(mpol);
  kmult = -eval(a,multroot')'/.eval(b,multroot')';
  mroot = 0;
  for j=1:max(size(kmult)),..
    if abs(imag(kmult(j)))<1e-9, mroot=[mroot;multroot(j)];..
  end;
  eras;
  if max(size(b))>1,..
    zeros = root(b);..
    plot(real(rl),imag(rl),'point=1',..
    real(poles),imag(poles),'point=7',..
    real(zeros),imag(zeros),'point=9');..
  else,..
    plot(real(rl),imag(rl),'point=1',..
    real(poles),imag(poles),'point=7');..
  end;
  if max(size(mroot))<2, return;
  mroot = mroot(2:max(size(mroot)));
  plot(plot('peek'),'scale');
  plot(real(mroot),imag(mroot),'point=8');
  plot('scale');
////////////////////////////////////////////////////
[value] = eval(Pol,svec);
  n = size(pol)*[0;1]-1;
  ns = size(svec)*[0;1]; one = ones(1:ns);
  value = pol(1)*one;
  for i=2:n+1; value = value*.svec + pol(i)*one;
////////////////////////////////////////////////////
[Prim] = deri(Pol);
  n = size(pol)*[0;1]-1; D = n:-1:0;
  prim = D*.pol;
  prim = prim(1:max([1,n]));
```