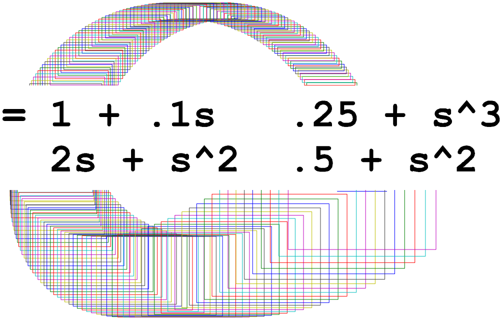
Version 2.0

[PolyX]

The Polynomial Toolbox for MATLAB

On-Line Manual



```
P = 1 + .1s    .25 + s^3
    2s + s^2   .5 + s^2
```

March, 1999

PolyX, Ltd

E-mail info@polyx.com
Support support@polyx.com
Sales sales@polyx.com
Web www.polyx.com
Tel. +420-2-66052314
Fax +420-2-6884554

Jarni 4
Prague 6, 16000
Czech Republic

Polynomial Toolbox Manual

Printing history: March 1999. First printing.

Contents

Basic operations on polynomial matrices

- Addition, subtraction and multiplication
- Determinants and inverses
- Rank, bases and null spaces
- Roots and stability
- Special constant matrices related to polynomials

Divisors and multiples

- Scalar divisor and multiple
- Division
- Division with remainder
- Greatest common divisor
- Coprimeness
- Least common multiple
- Matrix divisors and multiples
- Matrix division
- Matrix division with remainder
- Greatest common left divisor
- Least common right multiple
- Dual concepts

Reduced and canonical forms

- Row degrees
- Row and column reduced matrices
- Row reduced form
- Triangular and staircase form
- Another triangular form
- Hermite form
- Echelon form
- Smith form
- Invariant polynomials

Control system design

Introduction

Basic control routines

Introduction
Stabilization
Youla-Kucera parametrization
Example
Example
Pole placement
Example
Deadbeat controller
Example
Example

H-2 optimization

Introduction
Scalar case
Example
MIMO case
Examples

H-infinity optimization

Introduction
SISO mixed sensitivity optimization
Example
Descriptor solution of the standard *H*-infinity problem
Suboptimal solutions
Example
Optimal solutions

Quick Start

Initialization

Every Polynomial Toolbox session starts with the initialization command `pinit`:

```
pinit

Polynomial Toolbox initialized. To get started, type one of

these: helpwin or poldesk. For product information, visit

www.polyx.com or www.polyx.cz.
```

This function creates global polynomial properties and assigns them to their default values. If you place this command in your `startup.m` file then the Polynomial Toolbox is automatically initialized. If you include lines such as

```
path(path,'c:\Matlab\toolbox\polynomial')
pinit
```

in the `startup.m` file in the folder in which you start MATLAB then the toolbox is automatically included in the search path and initialized each time you start MATLAB.

Help

To see a list of all the commands and functions that are available type

```
help polynomial
```

To get help on any of the toolbox commands, such as `axxab`, type

```
help axxab
```

To get help on overloaded commands, that is, commands that also exist for other objects than polynomial matrices, such as `rank`, type

```
help pol/rank
```

How to define a polynomial matrix

Polynomial matrices may be looked at in two different ways.

You may directly enter a polynomial matrix by typing its entries. Upon initialization the Polynomial Toolbox defines several indeterminate variables for polynomials and polynomial matrices. One of them is *s.* Thus, you can simply type

```
P = [  1+s       s^2
      2*s^3   1+2*s+s^2 ]
```

to define the polynomial matrix

$$P(s) = \begin{bmatrix} 1+s & s^2 \\ 2s^3 & 1+2s+s^2 \end{bmatrix}$$

MATLAB returns

```
P =

       1 + s      s^2
       2s^3       1 + 2s + s^2
```

The Polynomial Toolbox displays polynomial matrices by default in this style.

We may also render the polynomial matrix *P* as

$$P(s) = \begin{bmatrix} 1+s & s^2 \\ 2s^3 & 1+2s+s^2 \end{bmatrix} = \underbrace{\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}}_{P_0} + \underbrace{\begin{bmatrix} 1 & 0 \\ 0 & 2 \end{bmatrix}}_{P_1} s + \underbrace{\begin{bmatrix} 0 & 1 \\ 0 & 1 \end{bmatrix}}_{P_2} s^2 + \underbrace{\begin{bmatrix} 0 & 0 \\ 2 & 0 \end{bmatrix}}_{P_3} s^3$$

Polynomial matrices may be defined this way by typing

```
P0 = [ 1 0; 0 1 ];

P1 = [ 1 0; 0 2 ];

P2 = [ 0 1; 0 1 ];

P3 = [ 0 0; 2 0 ];

P = pol([P0 P1 P2 P3],3)
```

MATLAB again returns

```
P =

     1 + s      s^2

     2s^3       1 + 2s + s^2
```

The display format may be changed to “coefficient matrix style” by the command

```
pformat coef
```

Typing the name of the matrix

```
P
```

now results in

```
Polynomial matrix in s: 2-by-2,  degree: 3

P =

  Matrix coefficient at s^0 :

     1     0

     0     1

  Matrix coefficient at s^1 :

     1     0
     0     2

  Matrix coefficient at s^2 :

     0     1

     0     1

  Matrix coefficient at s^3 :

     0     0

     2     0
```

Simple operations with polynomial matrices

In version 2 of the Polynomial Toolbox polynomial matrices are objects for which all standard operations are defined.

Addition, subtraction and multiplication

Define the polynomial matrices

```
P = [ 1+s 2; 3 4], Q = [ s^2 s; s^3 0]

P =

     1 + s     2

     3         4

Q =
          s^2     s

          s^3     0
```

The sum and product of *P* and *Q* follow easily:

```
S = P+Q

S =

          1 + s + s^2     2 + s

          3 + s^3         4

R = P*Q

R =

          s^2 + 3s^3      s + s^2

          3s^2 + 4s^3     3s
```

Concatenation and working with submatrices

All standard MATLAB operations to concatenate matrices or selecting submatrices also apply to polynomial matrices. Typing

```
T = [P Q]
```

results in

```
T =
        1 + s      2      s^2      s
        3          4      s^3      0
```

The last row of *T* may be selected by typing

```
t = T(2,:)

t =

        3      4      s^3      0
```

Submatrices may be assigned values by commands such as

```
T(:,1:2) = eye(2)

T =

        1      0      s^2      s

        0      1      s^3      0
```

Coefficients and coefficient matrices

It is easy to extract the coefficient matrices of a polynomial matrix. Given *T*, the coefficient matrix of s^2 may be retrieved as

```
T{2}

ans =

        0      0      1      0
```

```
     0     0     0     0
```

The coefficients of the (1,3) entry of *T* follow as

```
T{:}(1,3)

ans =

     0     0     1     0
```

Conjugation and transposition

Given

```
T

T =

     1     0     s^2     s

     0     1     s^3     0
```

the standard MATLAB conjugation operation results in

```
T'

ans =

     1       0

     0       1

     s^2    -s^3
```

```
         -s        0
```

Transposition follows by typing

```
T.'
 ans =
         1         0
         0         1
         s^2       s^3
         s         0
```

The command `transpose(T)` is synonymous with `T.'`

Advanced operations and functions

The Polynomial Toolbox knows many advanced operations and functions. After defining

```
P = [ 1+s   2

           3   4+s^2 ];
```

try a few by typing for instance `det(P)`, `roots(P)`, or `smith(P)`.

The commands may be grouped in the categories listed in Table 1. A full list of all available commands is available in the companion volume **Commands**. The same list appears after typing

```
help polynomial
```

in MATLAB.

Table 1. Command categories

Global structure	Numerical routines
Polynomial matrix properties	Canonical and reduced forms
Convertors	Control routines
Overlodaded operations	Equation solvers
Overloaded functions	Factorizations
Basic functions (other than overloaded)	Simulink
Advanced functions (other than overloaded)	Visualization
Special matrices	Graphic user interface
Matrix pencil routines	Demonstrations and help

Tutorial

Introduction

In this chapter we review in a tutorial style many of the functions and operations defined for polynomials and polynomial matrices. Functions and operations for LTI systems defined by polynomial matrix fractions are discussed in the chapter *Polynomial matrix fractions and LTI systems.* The chapter *Control system design* covers the applications of polynomial matrices in control system design.

More detailed information is available in the chapter *Reference*, and in the manual pages included in the companion volume **Commands**.

Entering polynomial matrices

Polynomials and polynomial matrices are most easily entered using one of the indeterminate variables *s*, *p*, *z*, *q*, *z*^–1 or *d* that are recognized by the Polynomial Toolbox, combined with the usual MATLAB conventions for entering matrices. Thus, typing

```
P = [   1+s      2*s^2

       2+s^3      4   ]
```

defines the matrix

$$P(s) = \begin{bmatrix} 1+s & 2s^2 \\ 2+s^3 & 4 \end{bmatrix}$$

and returns

```
P =

      1 + s        2s^2

      2 + s^3      4
```

In a system and control theoretic context, the indeterminates *s* and *p* are usually associated with continuous time, and are closely related to the differential operator $x(t) \rightarrow \dot{x}(t)$.

The indeterminates *z* and *q* are associated with the discrete-time one-step shift operator $x_t \rightarrow x_{t+1}$, and the indeterminates *d* and *z*^–1 with the delay operator $x_t \rightarrow x_{t-1}$. The indeterminate *z*^–1 may be entered with arbitary nonnegative integral powers. For typing convenience *z*^–1 may be abbreviated to `zi`. Thus,

```
P = 1+z^-1+z^-2

P =

      1 + z^-1 + z^-2
```

and

```
P = 1+zi+zi^2
```

```
P =

     1 + z^-1 + z^-2
```

return the same result.

Note that if any of the default indeterminates is redefined as a variable then it is no longer available as an indeterminate. Typing

```
s = 1;

P = 1+s^2
```

results in

```
P =

     2
```

To free the variable simply type

```
clear s;

P = 1+s^2
```

```
P =

     1 + s^2
```

The `pol` command

Polynomials and polynomial matrices may also be entered in terms of their coefficients or coefficient matrices. For this purpose the `pol` command is available. Typing

```
P0 = [1 2;3 4];
P1 = [3 4;5 1];
P2 = [1 0;0 1];
P = pol([P0 P1 P2],2,'s')
```

for instance, defines the polynomial matrix

```
P =

     1 + 3s + s^2      2 + 4s

     3 + 5s            4 + s + s^2
```

according to

$$P(s) = P_0 + P_1 s + P_2 s^2$$

The Polynomial Matrix Editor

More complicated polynomial matrices may be entered and edited with the help of the Polynomial Matrix Editor.

Changing the default indeterminate variable

After the Polynomial Toolbox has been started up the default indeterminate variable is *s.* This implies, among other things, that the command

```
P = pol([P0 P1 P2],2)
```

returns a polynomial matrix

```
P =

    1 + 3s + s^2      2 + 4s

    3 + 5s            4 + s + s^2
```

in the indeterminate *s*. The indeterminate variable may be changed with the help of the `gensym` command: typing

```
gensym z; pol([P0 P1 P2],2)
```

for instance, results in

```
ans =

    1 + 3z + z^2      2 + 4z

    3 + 5z            4 + z + z^2
```

The indeterminate variable *v* is automatically replaced by the default indeterminate variable. Thus, after the default indeterminate variable has been set to *z* by the command

```
gensym z
```

then

```
V = 1+v^2+3*v^3
```

returns

```
V =

     1 + z^2 + 3z^3
```

Concatenation and working with submatrices

Standard MATLAB conventions may be used to concatenate polynomial and standard matrices:

```
[P 1+s 3]
```

results in

```
ans =

     1 + s^2     1 + s     3
```

Submatrices may be selected such as in

```
ans(2:3)
```

```
ans =

     1 + s     3
```

All usual MATLAB subscripting and colon notations are available.

Coefficients and coefficient matrices

It is easy to extract the coefficient matrices of a polynomial matrix. Given

```
T = [ 1+s  2  s^2  s
       3   4  s^3  0 ];
```

the coefficient matrix of s^2 may be retrieved as

```
T{2}
ans =
     0     0     1     0
     0     0     0     0
```

The coefficients of the (1,3) entry of *T* follow as

```
T{:}(1,3)
ans =
   0     0     1     0
```

Basic operations on polynomial matrices

Addition, subtraction and multiplication

Define the polynomial matrices

```
P = [ 1+s 2; 3 4], Q = [ s^2 s; s^3 0]
```

```
P =

     1 + s     2

     3         4

Q =

     s^2     s

     s^3     0
```

The sum and product of *P* and *Q* follow easily:

```
S = P+Q
```

```
S =

     1 + s + s^2     2 + s

     3 + s^3         4
```

```
R = P*Q
```

```
R =
          s^2 + 3s^3      s + s^2

          3s^2 + 4s^3     3s
```

The command

```
R+3

ans =

          3 + s^2 + 3s^3      3 + s + s^2

          3 + 3s^2 + 4s^3     3 + 3s
```

obviously is interpreted as the instruction to add three times the unit matrix to *R.* The command

```
3*R

ans =

          3s^2 + 9s^3      3s + 3s^2

          9s^2 + 12s^3     9s
```

yields the expected result.

Determinants and inverses

The determinant of a square polynomial matrix is defined exactly as its constant matrix counterpart. In fact, its computation is not much more difficult:

```
P = [1 s s^2; 1+s s 1-s; 0 –1 –s]
```

```
P =

     1         s     s^2

     1 + s     s     1 - s

     0        -1    -s

det(P)

ans =

     1 - s - s^2
```

If its determinant happens to be constant then the polynomial matrix is called unimodular:

```
U =

     2 - s - 2s^2      2 - 2s^2     1 + s

     1 - s - s^2       1 - s^2      s

    -1 - s            -s            1

det(U)

Constant polynomial matrix: 1-by-1

ans =
     1
```

If a matrix is suspected of unimodularity then one can make it sure by a special tester

```
isunimod(U)
```

```
ans =

     1
```

Also the adjoint matrix is defined as for constant matrices. The adjoint is a polynomial matrix and may be computed by typing

```
adj(P)
```

```
ans =

     1 - s - s^2     0      s - s^2 - s^3

     s + s^2        -s     -1 + s + s^2 + s^3

    -1 - s           1     -s^2
```

Quite to the contrary, the inverse of a square polynomial matrix is usually rational. Based on the well-known formula

$$P^{-1} = \frac{1}{\det P}\,\mathrm{adj} P\,,$$

the inverse is computed and returned in terms of the adjoint and the determinant.

```
[adjP,p] = inv(P)
adjP =
    -1 + s + s^2     0    -s + s^2 + s^3
    -s - s^2         s     1 - s - s^2 - s^3
     1 + s          -1     s^2
p =
    -1 + s + s^2
```

As expected,

```
P*adj(P)/det(P)
```

equals the identity matrix

```
Constant polynomial matrix: 3-by-3
ans =
     1     0     0
     0     1     0
     0     0     1
```

On the other hand, unimodular matrices do have a polynomial inverse:

```
[adjU,u] = inv(U)

adjU =

     1    -2 - s + s^2    -1 + s + s^2 - s^3

    -1     3 + s - s^2     1 - 2s - s^2 + s^3

     1    -2 + s^2         s - s^3

Constant polynomial matrix: 1-by-1

u =

     1
```

Indeed,

```
U*adj(U)

Constant polynomial matrix: 3-by-3

ans =

     1     0     0

     0     1     0

     0     0     1
```

If the matrix is nonsquare but has full rank then a usual partial replacement for the inverse is provided by the generalized Moore-Penrose pseudoinverse, which is computed by the `pinv` function.

```
Q = P(:,1:2)

Q =

     1          s

     1 + s      s

     0         -1

[Qpinv,d] = pinv(Q)

Qpinv =

     1 - s^3        1 + s + s^3      2s + s^2

     s^2 + s^3    -s^2              -2 - 2s - s^2

d =

     2 + 2s + s^2 + s^4
```

Once again,

```
Qpinv*Q/d

Constant polynomial matrix: 2-by-2
```

```
ans =

     1     0

     0     1
```

Rank, bases and null spaces

A polynomial matrix $P(s)$ has *full column rank* (or full *normal column rank*) if it has full column rank everywhere in the complex plane except at a finite number of points. Similar definitions hold for *full row rank* and *full rank.*

Recall that

```
P

P =

     1         s     s^2

     1 + s     s     1 - s

     0        -1    -s
```

The rank test

```
isfullrank(P)

ans =

     1
```

confirms that P has full rank.

The *normal rank* of a polynomial matrix $P(s)$ equals

$$\max_{s \in C} \operatorname{rank} P(s)$$

Similar definitions apply to the notions of normal column rank and normal row rank. The rank is calculated by

```
rank(P)
```

```
ans =

     3
```

As for constant matrices, rank evaluation may be quite sensitive and an *ad hoc* change of tolerance (which may be included as an optional input parameter) may be helpful for difficult examples.

A polynomial matrix is *nonsingular* if it has full normal rank.

```
issingular(P)
```

```
ans =

     0
```

There are two important subspaces (more precisely, submodules) associated with a polynomial matrix $A(s)$: its null space and its range (or span). The (right) null space is defined as the set of all polynomial vectors $x(s)$ such that $A(s)x(s) = 0$. It is computed by

```
A = P(1:2,:)

A =

     1          s      s^2

     1 + s      s      1 - s

N = null(A)

N =

     s - s^2 - s^3

    -1 + s + s^2 + s^3

    -s^2
```

Here the null space dimension is 1 and its basis has degree 3.

The range of $A(s)$ is the set of all polynomial vectors $y(s)$ such that $y(s)=A(s)x(s)$ for some polynomial vector $x(s)$. In the Polynomial Toolbox, the minimal basis of the range is returned by the command

```
minbasis(A)

Constant polynomial matrix: 2-by-2

ans =

     1     0
```

```
        2      1
```

Roots and stability

The *roots* or *zeros* of a polynomial matrix $P(s)$ are those points s_i in the complex plane where $P(s)$ loses rank.

```
roots(P)

ans =

   -1.6180

    0.6180
```

The roots can be both finite and infinite. The infinite roots are normally suppressed. To reveal them, type

```
roots(P,'all')

ans =

   -1.6180

    0.6180

       Inf
```

Unimodular matrices have no finite roots:

```
roots(U)

ans =
     []
```

but typically have infinite roots:

```
roots(U,'all')

ans =

   Inf

   Inf

   Inf
```

If $P(s)$ is square then its roots are the roots of its determinant $\det P(s)$, including multiplicity:

```
roots(det(P))

ans =

   -1.6180

    0.6180
```

The finite roots may be visualized as in Fig. 1 by typing

```
zpplot(P)
```

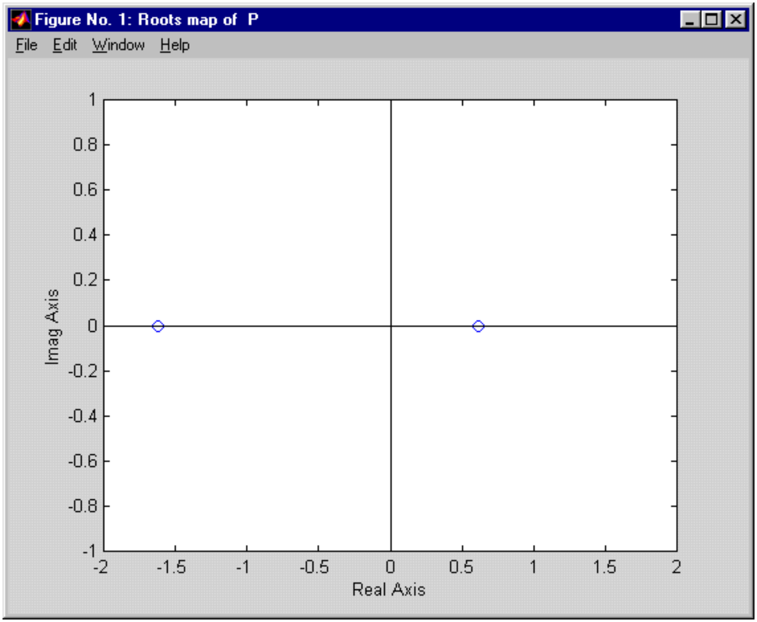


Fig. 1. Locations of the roots of a polynomial matrix

A polynomial matrix is stable if all its roots fall within a relevant stability region. Three standard stability regions are considered in the Polynomial Toolbox:

- the open left half plane for polynomial matrices in s and p, which means Hurwitz stability used for continuous-time systems

- the open unit disc for polynomial matrices in z and q, which corresponds to Schur stability for discrete-time systems in forward-shift operators, and
- the exterior of the unit disc for polynomial matrices in z^{-1} and d, which corresponds to inverse Schur stability for discrete-time systems in delay operators.

The macro `isstable` checks stability chosen according to the variable symbol. Thus,

```
isstable(s-2)

ans =

   0
```

and

```
isstable(z-2)

ans =

     0
```

but

```
isstable(z^-1-2)

ans =

     1
```

Special constant matrices related to polynomials

There are several interesting constant matrices that are composed from the coefficients of polynomials (or the matrix coefficients of polynomial matrices) and are frequently encountered in mathematical and engineering textbooks. Given a polynomial

$$p(v) = p_0 + p_1 v + p_2 v^2 + \cdots + p_d v^n$$

of degree n we may for instance define the corresponding $n \times n$ *Hurwitz matrix*

$$H_p = \begin{bmatrix} p_{n-1} & p_{n-3} & p_{n-5} & \cdots & \cdots & \\ p_n & p_{n-2} & p_{n-4} & \cdots & \cdots & \\ 0 & p_{n-1} & p_{n-3} & p_{n-5} & \cdots & \\ 0 & p_n & p_{n-2} & p_{n-4} & \cdots & \\ 0 & 0 & 0 & p_n & \ddots & \\ 0 & 0 & 0 & 0 & \cdots & p_0 \end{bmatrix},$$

a $k \times (n+k)$ *Sylvester matrix* (for some $k \geq 1$)

$$S_p = \begin{bmatrix} p_0 & p_1 & \cdots & p_n & 0 & \cdots & \cdots & 0 \\ 0 & p_0 & p_1 & \cdots & p_n & 0 & \cdots & 0 \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & \cdots & \cdots & 0 & p_0 & p_1 & \cdots & p_n \end{bmatrix}$$

or an $n \times n$ *companion matrix*

$$C(p)=\begin{bmatrix} 0 & 1 & 0 & \cdots & \cdots \\ 0 & 0 & 1 & 0 & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & \cdots & \cdots & 0 & 1 \\ -\dfrac{p_0}{p_n} & -\dfrac{p_1}{p_n} & \cdots & \cdots & -\dfrac{p_{n-1}}{p_n} \end{bmatrix}.$$

Using the Polynomial Toolbox, we take

```
p = pol([-1 1 2 3 4 5],5)

p =

    -1 + s + 2s^2 + 3s^3 + 4s^4 + 5s^5
```

and simply type

```
hurwitz(p)

ans =

     4     2    -1     0     0

     5     3     1     0     0

     0     4     2    -1     0

     0     5     3     1     0

     0     0     4     2    -1
```

or

```
sylv(p,3)

ans =

    -1     1     2     3     4     5     0     0     0
     0    -1     1     2     3     4     5     0     0
     0     0    -1     1     2     3     4     5     0
     0     0     0    -1     1     2     3     4     5
```

or

```
compan(p)

ans =

         0    1.0000         0         0         0
         0         0    1.0000         0         0
         0         0         0    1.0000         0
         0         0         0         0    1.0000
    0.2000   -0.2000   -0.4000   -0.6000   -0.8000
```

For a polynomial matrices the block matrix versions are defined and computed in a fairly obvious manner.

Divisors and multiples

Scalar divisor and multiple

To understand when division of polynomials and polynomial matrices is possible, consider three polynomials $a(s)$, $b(s)$ and $c(s)$ such that $a(s) = b(s)c(s)$. We say that $b(s)$ is a *divisor* (or *factor*) of $a(s)$ or $a(s)$ is a *multiple* of $b(s)$, and write $a(s) \mid b(s)$. This is sometimes also stated as $b(s)$ divides $a(s)$. For example, take

```
b = 1-s; c = 1+s; a = b*c

a =

     1 - s^2
```

Division

As $b(s)$ is a divisor of $a(s)$, the division

```
a/b
```

can be done and results in

```
ans =

     1 + s
```

Of course, the division by $b(s)$ fails if $b(s)$ is not a divisor

```
c/b
```

```
Constant polynomial matrix: 1-by-1

ans =

     NaN
```

Division with remainder

On the other hand, any $n(s)$ can be divided by any nonzero $d(s)$ using another operation called *division with a remainder.* This division results in a quotient $q(s)$ and a remainder $r(s)$:

$$n(s) = q(s)d(s) + r(s)$$

Typically, it is required that $\deg r(s) < \deg d(s)$, which makes the result unique. Thus, dividing

```
n = (1-s)^2;
```

by

```
d = 1+s;
```

yields

```
[q,r] = ldiv(n,d)

q =

    -3 + s

Constant polynomial matrix: 1-by-1
```

```
r =

     4
```

Division with remainder is sometimes called Euclidean division.

Greatest common divisor If a polynomial $g(s)$ divides both $a(s)$ and $b(s)$ then $g(s)$is called a *common divisor* of $a(s)$ and $b(s)$. If, furthermore, $g(s)$ is a multiple of every common divisor of $a(s)$ and $b(s)$ then $g(s)$is *a greatest common divisor* of $a(s)$ and $b(s)$.

Coprimeness If the only common divisors of $a(s)$ and $b(s)$ are constants then the polynomials $a(s)$ and $b(s)$ are *coprime* (or *relatively prime*). To compute a greatest common divisor of

```
a = s+s^2;
```

and

```
b = s-s^2;
```

type

```
gld(a,b)
```

```
ans =

     s
```

Similarly, the polynomials

```
c = 1+s; d = 1-s;
```

are coprime as

```
gld(c,d)
```

is a constant

```
Constant polynomial matrix: 1-by-1

ans =

     1
```

Coprimeness may also be tested directly

```
isprime([c,d])

ans =

     1
```

As the two polynomials *c* and *d* are coprime, there exist other two polynomials *e* and *f* that satisfy the linear polynomial equation

$$ce + df = 1$$

The polynomials *e* and *f* may be computed according to

```
[e,f] = axbyc(c,d,1)

Constant polynomial matrix: 1-by-1
```

```
e =

     0.5

Constant polynomial matrix: 1-by-1

f =

     0.5
```

We check the result by typing

```
c*e+d*f

Constant polynomial matrix: 1-by-1

ans =

     1
```

Least common multiple

If a polynomial $m(s)$ is a multiple of both $a(s)$ and $b(s)$ then $m(s)$ is called a common multiple of $a(s)$ and $b(s)$. If, furthermore, $m(s)$ is a divisor of every common multiple of $a(s)$ and $b(s)$ then it is a *least common multiple* of $a(s)$ and $b(s)$:

```
m = llm(a,b)

m =

     s - s^3
```

The concepts just mentioned are combined in the well-known fact that the product of a greatest common divisor and a least common multiple equals the product of the two original polynomials:

```
isequal(a*b,gld(a,b)*llm(a,b))

ans =

     1
```

Matrix divisors and multiples

Next consider polynomial matrices $A(s)$, $B(s)$and $C(s)$ of compatible sizes such that $A(s) = B(s)C(s)$. We say that $B(s)$ is a left divisor of $A(s)$, or $A(s)$ is a right multiple of $B(s)$. Take for instance

```
B = [1 s; 1+s 1-s], C = [2*s 1; 0 1], A = B*C

B =

     1          s

     1 + s      1 - s

C =

     2s     1

     0      1

A =

     2s              1 + s
```

```
          2s + 2s^2      2
```

Matrix division As $B(s)$ is a left divisor of $A(s)$, the matrix left division

```
B\A
```

can be done and results in the matrix

```
ans =

          2s      1

          0       1
```

Of course, the left division by $B(s)$ fails if $B(s)$ is not a left divisor

```
B\C

Constant polynomial matrix: 2-by-2

ans =

          NaN     NaN

          NaN     NaN
```

This includes the case that it is a divisor but from the other ("wrong") side

```
A/B

Constant polynomial matrix: 2-by-2
```

```
ans =

     NaN     NaN

     NaN     NaN
```

Matrix division with remainder

On the other hand, a polynomial matrix $N(s)$ can be divided by any compatible nonsingular square matrix $D(s)$ *with a remainder,* resulting in a matrix quotient $Q(s)$ and a matrix remainder $R(s)$ such that

$$N(s) = D(s)Q(s) + R(s)$$

If it is required that the rational matrix $D^{-1}(s)R(s)$ is strictly proper then the division is unique. Thus, dividing a random matrix

```
N = prand(4,2,3,'ent','int')

N =

   0           -3 + 11s                        -1 + s

   5 - 4s^3     1 - 7s + 4s^2 + 8s^3 - 3s^4     4 + 6s
```

from the left by a random square matrix

```
D = prand(3,2,'ent','int')

D =

    -7                   3 - 2s + 3s^2
```

```
        4 + 4s + 6s^2     0
```

results in

```
[Q,R] = ldiv(N,D)
Q =
     0.44 - 0.67s    -0.11 + 1.7s - 0.5s^2     0
     0               -1.2                      0
R =
     3.1 - 4.7s     -0.28 + 20s    -1 + s
     3.2 + 0.89s     1.4 - 13s      4 + 6s
```

Indeed,

```
deg(det(D))
ans =
     4
```

while

```
deg(adj(D)*R,'ent')
ans =
     3       3       3

     3       3       3
```

so that each entry of the rational matrix $D^{-1}(s)R(s) = (\mathrm{adj}D)R\,/\,\det D$ is strictly proper.

Greatest common left divisor

If a polynomial matrix $G(s)$ is a left divisor of both $A(s)$ and $B(s)$ then $G(s)$ is called a *common left divisor* of $A(s)$ and $B(s)$. If, furthermore, $G(s)$ is a right multiple of every common left divisor of $A(s)$ and $B(s)$ then it is a *greatest common left divisor* of $A(s)$ and $B(s)$. If the only common left divisors of $A(s)$ and $B(s)$ are unimodular matrices then the polynomial matrices $A(s)$ and $B(s)$ are *left coprime*. To compute a greatest common left divisor of

```
A = [1+s-s^2, 2*s + s^2;1-s^2, 1+2*s+s^2]

A =

     1 + s - s^2     2s + s^2

     1 - s^2         1 + 2s + s^2
```

and

```
B = [2*s, 1+s^2; 1+s, s+s^2]

B =

     2s         1 + s^2
```

```
         1 + s       s + s^2
```

type

```
gld(A,B)
```

```
ans =

          0          1

         1 + s       0
```

Similarly, the polynomial matrices

```
C = [1+s, 1; 1-s 0], D = [1-s, 1; 1 1]
```

```
C =

          1 + s      1

          1 - s      0

D =

          1 - s      1

          1          1
```

are left coprime as

```
gld(D,C)
```

```
Constant polynomial matrix: 2-by-2

ans =

     1     0

     0     1
```

is obviously unimodular. You may also directly check

```
isprime([C,D])

ans =

   1
```

As the two polynomial matrices *C* and *D* are left coprime there exist other two other polynomial matrices *E* and *F* that satisfy

$$CE + DF = I$$

They may be computed according to

```
[E,F] = axbyc(C,D,eye(2))

Constant polynomial matrix: 2-by-2

E =

     0     0
     1    -1

Constant polynomial matrix: 2-by-2

F =

     0     0

    0     1
```

Least common right multiple

If a polynomial matrix $M(s)$ is a right multiple of both $A(s)$ and $B(s)$ then $M(s)$ is called a *common right multiple* of $A(s)$ and $B(s)$. If, furthermore, $M(s)$ is a left divisor of every common right multiple of $A(s)$ and $B(s)$ then $M(s)$ is *a least common right multiple* of $A(s)$ and $B(s)$.

```
M = lrm(A,B)

M =

     1 + 4s + 3s^2    -2 - 9s - 2s^2 + s^3

     2 + 4s + 2s^2    -5 - 7s - s^2 + s^3
```

which is verified by

```
A\M, B\M

ans =

     1 + s    -2 - s
     1 + s     -3
ans =
     2 + s     -5
     1         -2 + s
```

Dual concepts The dual concepts of right divisors, left multiples, common right divisors, greatest common right divisors, common left multiples, and least common left multiples are similarly defined and computed.

Reduced and canonical forms

Suppose that we have a polynomial matrix

```
P = [1+s^2, -2; s-1 1]
P =
     1 + s^2    -2
    -1 + s       1
```

of degree

```
deg(P)
ans =
     2
```

with leading coefficient matrix

```
Pl = P{2}
Pl =
     1     0
     0     0
```

Row degrees

Besides the (overall) degree of *P* we may also consider its *row degrees*

```
deg(P,'row')
ans =
     2
     1
```

and *column degrees*

```
deg(P,'col')
ans =
     2     0
```

Associated with the row degrees is the *leading row coefficient matrix*

```
lcoef(P,'row')

ans =

     1     0

     1     0
```

and associated with the column degrees is the *leading column coefficient matrix*

```
lcoef(P,'col')

ans =

     1    -2

     0     1
```

Row and column reduced matrices

A polynomial matrix is *row reduced*if its leading row coefficient matrix has full row rank. Similarly, it is *column reduced* if its leading column coefficient matrix has full column rank. The matrix *P* is definitely not row reduced

```
isfullrank(lcoef(P,'row'))

ans =

     0
```

but it is column reduced

```
isfullrank(lcoef(P,'col'))
```

```
ans =

     1
```

Row reduced form

Any polynomial matrix with full row rank may be transformed into *row reduced form* by pre-multiplying it by a suitable unimodular matrix. To compute a row reduced form of *P*, call

```
P_row_reduced = rowred(P)
```

```
P_row_reduced =

     1 + s    -2 - s
    -1 + s     1
```

Indeed, the row rank of

```
lcoef(P_row_reduced,'row')
```

```
ans =

     1    -1
     1     0
```

is full.

Triangular and staircase form

There are several special forms of a polynomial matrix that can be achieved by pre- and/or post-multiplying it by suitable unimodular matrix. These operations preserve many important properties and indeed serve to make these visible.

Thus, a lower-left triangular form $T(s)$ of $A(s)$ resulting from column operations $T(s) = A(s)U(s)$ can be computed by the macro `tri`:

```
A = [s^2 0 1; 0 s^2 1+s]

A =

     s^2      0        1

     0        s^2      1 + s

T = tri(A)

T =

    -1          0          0

    -1 - s    -1.2s^2      0
```

The corresponding unimodular matrix is returned by

```
[T,U] = tri(A); U

U =

     0     0.29              0.5
```

```
        0    -0.87 + 0.29s     0.5 + 0.5s

       -1    -0.29s^2         -0.5s^2
```

If $A(s)$ has not full row rank then $T(s)$ is in staircase form. Similarly, an upper-right triangular (row staircase) form is achieved by row (unimodular) operations. It results from the call `tri(A,'row')`.

Another triangular form

If $B(s)$is a square polynomial matrix with nonsingular constant term then another upper-triangular form may be obtained by the overloaded macro `lu`:

```
B = [ 1 1 s; s+1 0 s; 1-s s 2+s]

B =

     1         1     s

     1 + s     0     s

     1 - s     s     2 + s

[V,T] = lu(B)

V =

     1         0.33s                  0.11s

     1 + s     1 + 1.3s + 0.33s^2     0.44s + 0.11s^2

     1 - s     1 - 1.7s - 0.33s^2     1 - 0.56s - 0.11s^2
```

```
T =

       1      1 + 0.33s      0.78s + 0.11s^3
       0     -1             -0.67s - s^2 + 0.33s^3
       0      0              2 + 2s + 2s^2 - s^3
```

Hermite form

The triangular forms described above are by no means unique. A canonical triangular form is called the *Hermite* form. An $n \times m$ polynomial matrix $A(s)$ of rank r is in *column Hermite form* if it has the following properties:

- it is lower triangular
- the diagonal entries are all monic
- each diagonal entry has higher degree than any entry on its left
- in particular, if the diagonal element is constant then all off-diagonal elements in the same row are zero
- if $n > r$ then the last $n - r$ columns are zero

The nomenclature in the literature is not consistent. Some authors (in particular Kailath, 1980) refer to this as the row Hermite form. The polynomial matrix A is in *row Hermite form* if it is the transpose of a matrix in column Hermite form. The command

```
H = hermite(A)
```

returns the column Hermite form

```
H =

      1         0       0

      1 + s     s^2     0
```

while the call

```
[H,U] = hermite(A); U

U =

      0    -0.25             0.5

      0     0.75 - 0.25s     0.5 + 0.5s

      1     0.25s^2         -0.5s^2
```

provides a unimodular reduction matrix U such that $H(s) = A(s)U(s)$.

Echelon form

Yet another canonical form is called the *(Popov) echelon* form. A polynomial matrix E is in *column echelon form* (or *Popov form*) if it has the following properties:

- it is column reduced with its column degrees arranged in ascending order
- for each column there is a so-called *pivot index* i such that the degree of the i-th entry in this column equals the column degree, and the i-th entry is the lowest entry in this column with this degree

- the pivot indexes are arranged to be increasing
- each pivot entry is monic and has the highest degree in its row

A square matrix in column echelon form is both column and row reduced

Given a square and column-reduced polynomial matrix $D(s)$ the command

```
[E,U] = echelon(D)
```

computes the column echelon form $E(s)$ of $D(s)$. The unimodular matrix $U(s)$ satisfies $E(s) = D(s)U(s)$.

By way of example, consider the polynomial matrix

```
D = [ -3*s s+2; 1-s 1];
```

To find its column echelon form and the associated unimodular matrix, type

```
[E,U] = echelon(D)
```

MATLAB returns

```
E =

     2 + s    -6

     1        -4 + s

Constant polynomial matrix: 2-by-2
```

```
U =

     0    -1

     1    -3
```

Smith form

The ultimate, most structured canonical form for a polynomial matrix is its *Smith form*. A polynomial matrix $A(s)$ of rank r may be reduced to its Smith form $S(s) = U(s)A(s)V(s)$ by pre- and post- multiplication by unimodular matrices $U(s)$ and $V(s)$, respectively. The Smith form looks like this:

$$S(s) = \begin{bmatrix} S_r(s) & 0 \\ 0 & 0 \end{bmatrix}$$

with the diagonal submatrix

$$S_r(s) = \mathrm{diag}(a_1(s), a_2(s), \cdots, a_r(s))$$

The entries $a_1(s), a_2(s), \cdots, a_r(s)$ are monic polynomials such that $a_1(s)$ divides $a_{i+1}(s)$ for $i = 1, 2, \ldots, r-1$. The Smith form is particularly useful for theoretical considerations as it reveals many important properties of the matrix. Its practical use, however is limited because it is quite sensitive to small parameter perturbations. The computation of the Smith form becomes numerically troublesome as soon as the matrix size and degree become larger. The Polynomial Toolbox offers a choice of three different algorithms to achieve the Smith form, all programmed in macro `smith.` For larger examples, a manual change of tolerance may be necessary. To compute the Smith form of a simple matrix

```
A=[1+s, 0, s+s^2; 0, s+2, 2*s+s^2]
```

```
A =

         1 + s     0         s + s^2

         0         2 + s     2s + s^2
```

simply call

```
smith(A)
```

```
ans =

         1     0                0

         0     2 + 3s + s^2     0
```

Invariant polynomials

The polynomials $a_1(s), a_2(s), \cdots, a_r(s)$ that appear in the Smith form are uniquely determined and are called the *invariant polynomials* of $A(s)$. They may be retrieved by typing

```
diag(smith(A))
```

```
ans =

     1

  2 + 3s + s^2
```

Polynomial matrix equations

Diophantine equations

The simplest type of linear scalar polynomial equation — called Diophantine equation after by the Alexandrian mathematician Diophantos (A.D. 275) — is

$$a(s)x(s)+b(s)y(s)=c(s)$$

The polynomials $a(s)$, $b(s)$ and $c(s)$ are given while the polynomials $x(s)$ and $y(s)$ are unknown. The equation is solvable if and only if the greatest common divisor of $a(s)$ and $b(s)$ divides $c(s)$. This implies that with $a(s)$ and $b(s)$ coprime the equation is solvable for any right hand side polynomial, including $c(s)=1$.

The Diophantine equation possesses infinitely many solutions whenever it is solvable. If $x'(s)$, $y'(s)$ is any (particular) solution then the general solution of the Diophantine equation is

$$\begin{aligned} x(s) &= x'(s)+\bar{b}(s)t(s) \\ y(s) &= y'(s)-\bar{a}(s)t(s) \end{aligned}$$

Here $t(s)$ is an arbitrary polynomial (the parameter) and $\bar{a}(s)$, $\bar{b}(s)$ are coprime polynomials such that

$$\frac{\bar{b}(s)}{\bar{a}(s)}=\frac{b(s)}{a(s)}$$

If the polynomials $a(s)$ and $b(s)$ themselves are coprime then one can naturally take $\bar{a}(s)=a(s)$ and $\bar{b}(s)=b(s)$.

Among all the solutions of Diophantine equation there exists a unique solution pair $x(s)$, $y(s)$ characterized by

$$\deg x(s) < \deg \bar{b}(s).$$

There is another (generally different) solution pair characterized by

$$\deg y(s) < \deg \bar{a}(s).$$

The two special solution pairs coincide only if

$$\deg a(s) + \deg b(s) \geq \deg c(s).$$

The Polynomial Toolbox offers two basic solvers that may be used for scalar and matrix Diophantine equations. They are suitably named `axbyc` and `xaybc`. For example, consider the simple polynomials

```
a = 1+s+s^2; b = 1-s; c = 3+3*s;
```

When typing

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

Constant polynomial matrix: 1-by-1

x =

     2

y =
        1 + 2s
```

MATLAB returns the solution pair $\left(x(s), y(s)\right) = \left(2, 1+2s\right)$ with minimal overall degree. The alternative call

```
   [x,y,f,g] = axbyc(a,b,c)
```

```
Constant polynomial matrix: 1-by-1

   x =

        2

   y =

        1 + 2s

   f =

       -1 + s

   g =

        1 + s + s^2
```

retrieves the complete general solution in the form $\left(x(s)+f(s)t(s), y(s)+g(s)t(s)\right)$ with an arbitrary polynomial parameter $t(s)$.

To investigate the case of different minimal degree solutions, consider a right hand side of higher degree

```
c = 15+15*s^4;
```

As before, the call

```
[x1,y1] = axbyc(a,b,c)

x1 =

     8 - 13s + 15s^2

y1 =

     7 + 12s + 2s^2
```

results in the solution of minimal overall degree (in this case $\deg x_1 = \deg y_1 = 2$).

A slightly different command

```
[x2,y2] = axbyc(a,b,c,'minx')

Constant polynomial matrix: 1-by-1

x2 =

     10

y2 =

     5 - 5s - 15s^2 - 15s^3
```

returns another solution with the minimal degree of the first unknown. Finally, typing

```
[x2,y2] = axbyc(a,b,c,'miny')

x2 =

          10 - 15s + 15s^2

y2 =

          5 + 10s
```

produces the solution of minimal degree in the second unknown.

Bézout equations

A Diophantine equation with 1 on its right hand side is called a *Bézout* equation. It may look like

$$a(s)x(s)+b(s)y(s)=1$$

with $a(s)$, $b(s)$ given and $x(s)$, $y(s)$ unknown.

Matrix polynomial equations

In the matrix case, the polynomial equation becomes a polynomial matrix equation. The basic matrix polynomial (or polynomial matrix) equations are

$$A(s)X(s)=B(s)$$

and

$$X(s)A(s)=B(s)$$

or even

$$A(s)X(s)B(s)=C(s)$$

$A(s)$, $B(s)$ and, if applicable, $C(s)$ are given while $X(s)$ is unknown. The Polynomial Toolbox functions to solve these equations are conveniently named `axb`, `xab`, and `axbc`. Hence, given the polynomial matrices

```
A= [1 s 1+s; s-1 1 0]; B = [s 0; 0 1];
```

the call

```
X0 = axb(A,B)

X0 =

     1        -1

     1 - s     s

    -1 + s     1 – s
```

solves the first equation and returns its solution of minimal overall degree. Typing

```
[X0,K] = axb(A,B); K

 K =

     1 + s

     1 - s^2
```

```
-1 - s + s^2
```

also computes the right null-space of A so that all the solutions to $A(s)X(s) = B(s)$ may easily be parametrized as

$$X(s) = X_0(s) + K(s)T(s)$$

$T(s)$ is an arbitrary polynomial matrix of compatible size. The other equations are handled similarly.

One-sided equations

In systems and control several special forms of polynomial matrix equations are frequently encountered, in particular the *one-sided* equations

$$A(s)X(s) + B(s)Y(s) = C(s)$$

and

$$X(s)A(s) + Y(s)B(s) = C(s)$$

Two-sided equations

Also the *two-sided* equations

$$A(s)X(s) + Y(s)B(s) = C(s)$$

and

$$X(s)A(s) + B(s)Y(s) = C(s)$$

are common. $A(s)$,$B(s)$ and $C(s)$ are always given while $X(s)$ and $Y(s)$ are to be computed. $A(s)$ is typically square invertible.

The solutions of the one- and two-sided equations may be found with the help of the Polynomial Toolbox macros `axbyc`, `xaybc`, and `axybc` . Thus, for the matrices

```
A= [1 s; 1+s 0]; B = [s 1; 1 s]; C = [1 0; 0 1];
```

the call

```
[X,Y] = axbyc(A,B,C)
```

returns

```
  X =

      0.25 - 0.5s     0.5

      0.25 + 0.5s    -0.5

 Y =

      -0.25 - 0.5s     0.5

       0.75 + 0.5s    -0.5
```

Various other scalar and matrix polynomial equations may be solved by directly applying appropriate solvers programmed in the Polynomial Toolbox, such as the equation

$$A(s)X^*(s)+X(s)A^*(s)=B(s)$$

Table 2 lists all available polynomial matrix equation solvers.

Table 2. Equation solvers

Equation	**Name of the routine**
$AX = B$	axb
$AXB = C$	axbc
$AX + BY = C$	axbyc
$A^*X + X^*A = B$	axxab
$A^*X + Y^*A = B$	axyab
$AX + YB = C$	axybc
$XA^* + AX^* = B$	xaaxb
$XA = B$	xab
$XA + YB = C$	xaybc

Factorizations

Besides linear equations special quadratic equations in scalar and matrix polynomials are encountered in various engineering fields. One of them is the polynomial *spectral factorization*

$$A(s) = X^*(s) J X(s)$$

and the *spectral co-factorization*

$$A(s) = X(s) J X^*(s).$$

In either case, the given polynomial matrix $A(s)$ satisfies $A(s) = A^*(s)$ (we say it is para-Hermitian symmetric) and the unknown $X(s)$ is to be stable. The case of $A(s)$ positive definite on the stability boundary results in $J = I$.

Spectral factorization with $J = I$ is the main tool to design LQ and LQG controllers as well as Kalman filters. On the other hand, if $A(s)$ is indefinite on the stability boundary then $J = \mathrm{diag}\{+1, +1, \ldots, +1, -1, -1 \ldots, -1\}$. This is the famous J-spectral factorization problem, which is an important tool for robust control and filter design based on H_∞ norms. The Polynomial Toolbox provides two macros called `spf` and `spcof` to handle spectral factorization and co-factorization, respectively.

By way of illustration consider the para-Hermitian matrix

```
A =

     34 - 56s^2          -13 - 22s + 60s^2     36 + 67s

    -13 + 22s + 60s^2     46 - 1e+002s^2      -42 - 26s +
38s^2

     36 - 67s            -42 + 26s + 38s^2     59 - 42s^2
```

Its spectral factorization follows by typing

```
[X,J] = spf(A)

X =

     2.1 + 0.42s     5.2 + 0.39s    -2 + 0.35s

    -5.5 + 4s        4.3 + 0.64s    -7.4 - 5.5s

     0.16 + 6.3s    -0.31 - 10s     -0.86 + 3.5s

J =

     1     0     0

     0     1     0

     0     0     1
```

while the spectral co-factorization of A is computed via

```
[Xcof,J] = spcof(A)

Xcof =

     2.7 + 0.42s     4.8 + 4s          2 + 6.3s

    -1.6 + 0.39s     0.93 + 0.64s     -6.5 - 10s

     4.3 + 0.35s     2.7 - 5.5s        5.8 + 3.5s

J =

     1     0     0

     0     1     0

     0     0     1
```

The resulting *J* reveals that the given matrix *A* is positive-definite on the imaginary axis. On the other hand, the following matrix is indefinite

```
B =

     5           -6 - 18s        -8

    -6 + 18s     -41 + 81s^2     -22 - 18s

    -8           -22 + 18s       -13
```

Its spectral factorization follows as

```
[Xf,J] = spf(B)

Xf =

     3.3     2.9 - 0.92s    -1.6

     1.8     6.6 + 0.44s     3.4

     1.6     2.5 + 9s       -2

J =

     1     0     0

     0    -1     0

     0     0    -1
```

or

```
[Xcof,J] = spcof(B)

Xcof =

     3.3            -1.8             1.6

    -1.9 + 0.92s    -4.4 + 0.44s    -5 - 9s

    -1.6            -3.4            -2

J =
     1     0     0

     0    -1     0

     0     0    -1
```

Matrix pencil routines

Matrix pencils are polynomial matrices of degree 1. They arise in the study of continuous- and discrete-time linear time-invariant state space systems given by

$$\begin{aligned} \dot{x}(t) &= Ax(t) + Bu(t) \qquad & x(t+1) &= Ax(t) + Bu(t) \\ y(t) &= Cx(t) + Du(t) \qquad & y(t) &= Cx(t) + Du(t) \end{aligned}$$

and continuous- and discrete-time descriptor systems given by

$$\begin{aligned} E\dot{x}(t) &= Ax(t) + Bu(t) \qquad & Ex(t+1) &= Ax(t) + Bu(t) \\ y(t) &= Cx(t) + Du(t) \qquad & y(t) &= Cx(t) + Du(t) \end{aligned}$$

The transfer matrix of the descriptor system is

$$H(s) = C(sE - A)^{-1}B + D$$

in the continuous-time case, and

$$H(s) = C(zE - A)^{-1}B + D$$

in the discrete-time case. The polynomial matrices

$$sE - A, \quad zE - A$$

that occur in these expressions are matrix pencils. In the state space case they reduce to the simpler forms $sI - A$ and $zI - A$.

Transformation to Kronecker canonical form

A nonsingular square real matrix pencil $P(s)$ may be transformed to its *Kronecker canonical form*

$$C(s) = QP(s)Z = \begin{bmatrix} a + sI & 0 \\ 0 & I + se \end{bmatrix}$$

Q and Z are constant orthogonal matrices, a is a constant matrix whose eigenvalues are the negatives of the roots of the pencil, and e is a nilpotent constant matrix. (That is, there exists a nonnegative integer k such that $e^i = 0$ for $i \geq k$. The integer k is called the *nilpotency* of e.)

This transformation is very useful for the analysis of descriptor systems because it separates the finite and infinite roots of the pencil and, hence, the corresponding modes of the system.

By way of example we consider the descriptor system

$$\underbrace{\begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}}_{E} \dot{x} = \underbrace{\begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}}_{A} x + \underbrace{\begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}}_{B} u$$

$$y = \underbrace{\begin{bmatrix} 1 & -1 & 0 \end{bmatrix}}_{C} x + \underbrace{2}_{D} u$$

The system is defined by the matrices

```
A = [ -1 0 0; 0 1 0; 0 0 1 ];
B = [ 1; 0; 1 ];
C = [ 1 -1 0 ];
D = 2;
E = [ 1 0 0; 0 0 1; 0 0 0 ];
```

We compute the canonical form of the pencil $sE - A$ by typing

```
c = pencan(s*E-A)
c =
     1 + s      0      0
     0          1     -s
     0          0      1
```

Inspection shows that *a* = 1 has dimensions 1×1 and that *e* is the 2×2 nilpotent matrix

$$e = \begin{bmatrix} 0 & -1 \\ 0 & 0 \end{bmatrix}$$

The descriptor system has a finite pole at -1 and two infinite poles.

Transformation to Clements form

A para-Hermitian real pencil $P(s) = sE + A$, with E skew-symmetric and A symmetric, may — under the assumption that it has no roots on the imaginary axis — be transformed into its “Clements” form (Clements, 1993) according to

$$C(s) = UP(s)U^T = \begin{bmatrix} 0 & 0 & sE_1 + A_1 \\ 0 & A_2 & sE_3 + A_3 \\ -sE_1^T + A_1^T & -sE_3^T + A_3^T & sE_4 + A_3 \end{bmatrix}$$

The constant matrix U is orthogonal and the finite roots of the pencil $sE_1 + A_1$ all have negative real parts. This transformation is needed for the solution of various spectral factorization problems that arise in the solution of H_2 and H_∞ optimization problems for descriptor systems.

Let the para-Hermitian pencil P be defined as

$$P(s) = \begin{bmatrix} 100 & -0.01 & s & 0 \\ -0.01 & -0.01 & 0 & 1 \\ -s & 0 & -1 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

It Clements form follows by typing

```
P = [100 -0.01 s 0; -0.01 -0.01 0 1; -s 0 -1 0; 0 1 0 0];

C = pzer(clements(P))

C =
0          0                      0         -10 + s

0         -1                      0         -3.5e-005 -
0.0007s

0          0                      1         -0.014 +
0.00071s

-10 - s    -3.5e-005 + 0.0007s    -0.014 - 0.00071s     99
```

The `pzer` function is included in the command to clear small coefficients in the result.

Pencil Lyapunov equations

When working with matrix pencils the two-sided matrix pencil equation

$$A(s)X + YB(s) = C(s)$$

is sometimes encountered. *A* and *B* are square matrix pencils and *C* is a rectangular pencil with as many rows as *A* and as many columns as *B*. If *A* and *B* have no common roots (including roots at infinity) then the equation has a unique solution pair *X*, *Y* with both *X* and *Y* a constant matrix.

By way of example, let

```
A = s+1;

B = [ s 0

      1 2];
```

```
C = [ 3 4 ];
```

Then

```
[X,Y] = plyap(A,B,C)
```

gives the solution

```
X =

     1     0

Y =

    -1     2
```

Numerical routines

The Polynomial Toolbox provides several auxiliary routines for ordinary matrices that may come in handy for other applications. We review two of them.

qzord

The standard MATLAB routine `qz` allows the simultaneous transformation

$$QAZ = a, \quad QBZ = b$$

of two square matrices A and B into upper triangular matrices a and b. Q and Z are unitary. The ratios a_{ii} / b_{ii}, $i = 1, 2, \cdots$, of the diagonal entries of a and b are the generalized eigenvalues of the matrix pair (A, B).

The Polynomial Toolbox routine `qzord` additionally orders the diagonal ratios a_{ii} / b_{ii} according to the options `'partial'` (default) and `'full'` . With the option `'partial'` the finite eigenvalues come first and are followed by the infinite eigenvalues. With the option `'full'` the generalized eigenvalues are ordered according to increasing real parts with the infinite generalized eigenvalues last.

Consult the manual page for `qzord` for examples.

schurst

The Polynomial Toolbox routine `schurst` extends the standard MATLAB routine `schur`. The routine `schur` transforms a constant matrix into upper triangular form. The Toolbox routine `schurst` additionally orders the diagonal entries so that the diagonal entries with negative real part precede those with nonnegative real part. See the manual page for `schurst` for an example.

Visualization

The Polynomial Toolbox has several graphical routines.

pplot, pplot3

The macros `pplot` and `pplot3` provide two- and three-dimensional visualizations of polynomial matrices. They are described in the chapter ***Reference***, and also on their manual pages.

zpplot

The macro `zpplot` plots the zeros of polynomial matrices and the zeros and poles of polynomial matrix fractions. For examples see its manual page.

khplot, ptoplot

The routines `khplot` and `ptoplot` are graphical routines for parametric robustness analysis. They are described in the chapter ***Robust control with parametric uncertainties***, and of course on their manual pages.

The Polynomial Matrix Editor

Introduction

The Polynomial Matrix Editor (PME) is recommended for creating and editing polynomial and standard MATLAB matrices of medium to large size, say from about 4×4 to 30×35. Matrices of smaller size can easily be handled in the MATLAB command window with the help of monomial functions, overloaded concatenation, and various applications of subscripting and subassigning. On the other hand, opening a matrix larger than 30×35 in the PME results in a window that is difficult to read.

Quick start

Type `pme` to open the main window called Polynomial Matrix Editor. This window displays all polynomial matrices (POL objects) and all standard MATLAB matrices (2-dimensional DOUBLE arrays) that exist in the main MATLAB workspace. It also allows you to create a new polynomial or standard MATLAB matrix. In the Polynomial Matrix Editor window you can

- create a new polynomial matrix, by typing its name and size in the first (editable) line and then clicking the Open button

- modify an existing polynomial matrix while retaining its size and other properties: To do this just find the matrix name in the list and then double click the particular row

- modify an existing polynomial matrix to a large extent (for instance by changing also its name, size, variable symbol, etc.): To do this, first find the matrix name in the list and then click on the corresponding row to move it up to the editable row. Next type in the new required properties and finally click Open

Each of these actions opens another window called Matrix Pad that serves for editing the particular matrix.

In the Matrix Pad window the matrix entries are displayed as boxes. If an entry is too long so that it cannot be completely displayed then the corresponding box takes a slightly different color (usually more pinkish.)

To edit an entry just click on its box. The box becomes editable and large enough to display its entire content. You can type into the box anything that complies with the MATLAB syntax and that results in a scalar polynomial or constant. Of course you can use existing MATLAB variables, functions, etc. The program is even more intelligent and handles some notation going beyond the MATLAB syntax. For example, you can drop the `*` (times) operator between a coefficient and the related polynomial symbol (provided that the coefficient comes first). Thus, you can type `2s` as well as `2*s`.

To complete editing the entry push the Enter key or close the box by using the mouse. If you have entered an expression that cannot be processed then an error message is reported and the original box content is recovered.

To bring the newly created or modified matrix into the MATLAB workspace finally click Save or Save As.

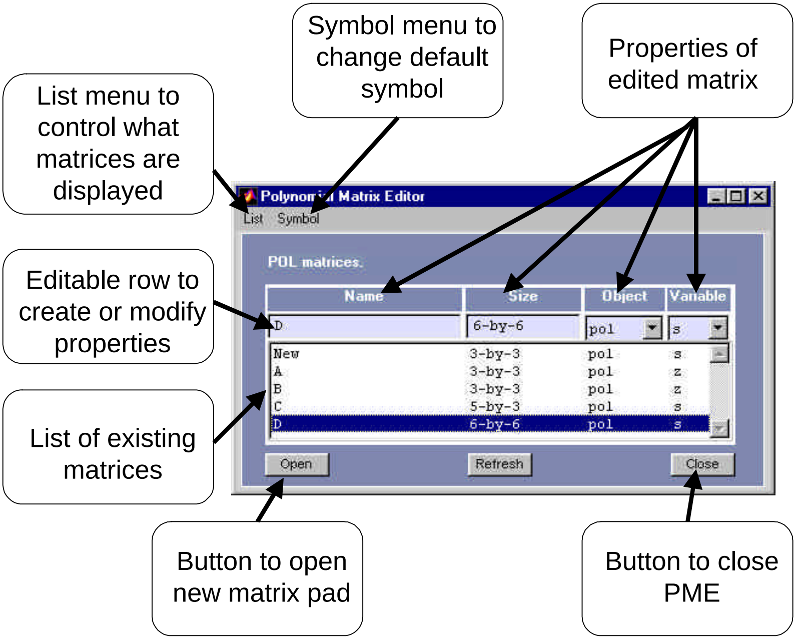
Main window

The main Polynomial Matrix Editor window is shown in Fig. 2.

Main window buttons

The main window contains the following buttons:

- Open Clicking the Open button opens a new Matrix Pad for the matrix specified by the first (editable) line. At most four matrix pads can be open at the same time.
- Refresh Clicking the Refresh button updates the list of matrices to reflect changes in the workspace since opening the PME or since it was last refreshed.
- Close Clicking the Close button terminates the current PME session.



List menu to control what matrices are displayed

Symbol menu to change default symbol

Properties of edited matrix

Editable row to create or modify properties

List of existing matrices

Button to open new matrix pad

Button to close PME

POL matrices.

Name	Size	Object	Variable
D	6-by-6	pol	s
New	3-by-3	pol	s
A	3-by-3	pol	z
B	3-by-3	pol	z
C	5-by-3	pol	s
D	6-by-6	pol	s

Fig. 2. Main window of the Polynomial Matrix Editor

Main window menus

The main PME window offers the following menus:

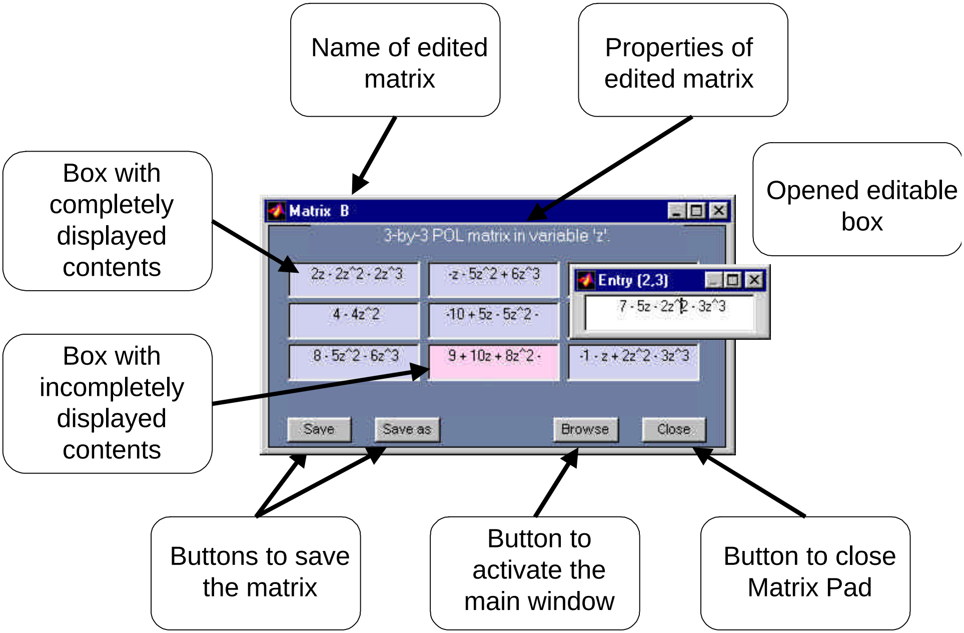
- Menu List Using the menu List you can set the type of matrices listed in the main window. They may be polynomial matrices (POL objects) – default – or standard MATLAB matrices or both at the same time.
- Menu Symbol Using the menu Symbol you can choose the symbol (such as *s*,*z*,...) that is by default offered for the 'New' matrix item of the list.

Matrix Pad window

To edit a matrix use one of the ways described above to open a Matrix Pad window for it. This window is show in Fig. 3.

The Matrix Pad window consists of boxes for the entries of the polynomial matrix. If some entry is too long and cannot be completely displayed in the particular box then its box takes a slightly different color.

To edit an entry you can click on its box. The box pops up, becomes editable and large enough to display its whole content. You can type into the box anything that complies with the MATLAB syntax and results in a scalar polynomial or constant. Of course, you can use all existing MATLAB variables, functions, etc. The editor is even a little more intelligent: it handles some notation going beyond the MATLAB syntax. For example, you can drop the `*` (times) operator between a coefficient and the related polynomial symbol (provided that the coefficient comes first). Thus, you can type `2s` as well as `2*s`. Experiment a little to learn more.



Matrix B

3-by-3 POL matrix in variable 'z'.

2z - 2z^2 - 2z^3
-z - 5z^2 + 6z^3
4 - 4z^2
-10 + 5z - 5z^2 -
8 - 5z^2 - 6z^3
9 + 10z + 8z^2 -
-1 - z + 2z^2 - 3z^3

Entry (2,3)

7 - 5z - 2z^2 - 3z^3

Name of edited matrix

Properties of edited matrix

Opened editable box

Box with completely displayed contents

Box with incompletely displayed contents

Buttons to save the matrix

Button to activate the main window

Button to close Matrix Pad

Fig. 3. Matrix Pad with an opened editable box

When editing is completed push the Enter key on your keyboard or close the box by mouse. If you have entered an expression that cannot be processed then an error message is appears and the original content of the box is retrieved.

When the matrix is ready bring it into the MATLAB workspace by clicking either the Save or the Save As button.

Matrix Pad buttons

The following buttons control the Matrix Pad window:

- Save button: Clicking Save copies the Matrix Pad contents into the MATLAB workspace.
- Save As button: Clicking Save As copies the matrix from Matrix Pad into the MATLAB workspace under another name.
- Browse button: Clicking Browse moves the cursor to the main window (the same as directly clicking there).

Close Button: Clicking Close closes the Matrix Pad window.

The Matrix Pad window consists of boxes for the entries of the polynomial matrix. If some entry is too long and cannot be completely displayed then its box takes a slightly different color.

To edit an entry you can click on its box. The box pops up, becomes editable and large enough to display its whole content. You can type into the box anything that complies with the MATLAB syntax and results in a scalar polynomial or constant. Of course, you can use all existing MATLAB variables, functions, etc. The editor is even a little more intelligent: it handles some notation going beyond the MATLAB syntax.

For example, you can drop the `*` (times) operator between a coefficient and the related polynomial symbol (provided that the coefficient comes first). Thus, you can type `2s` as well as `2*s`. Experiment a little to learn more.

When editing is completed push the Enter key on your keyboard or close the box by mouse. If you have entered an expression that cannot be processed then an error message is appears and the original content of the box is retrieved.

When the matrix is ready bring it into the MATLAB workspace by clicking either the Save or the Save As button.

Polynomial matrix fractions and LTI systems

Introduction

Linear time-invariant systems are a very important class of models for control. Even though the “real world” is without doubt thoroughly nonlinear, linear models provide an extraordinarily useful tool for the study of dynamical systems.

A very well know model for linear time-invariant systems is of course the familiar state space description, which for continuous-time systems takes the form

$$\begin{aligned} \dot{x}(t) &= Ax(t) + Bu(t) \\ y(t) &= Cx(t) + Du(t) \end{aligned}$$

with *u* the input, *y* the output, and *x* the state variable, and where the time *t* is a continuous variable. All these signals may be vector-valued, and *A, B, C* and *D* are constant matrices of appropriate dimensions.

Discrete-time state space models are similarly described by equations of the form

$$\begin{aligned} x(t+1) &= Ax(t) + Bu(t) \\ y(t) &= Cx(t) + Du(t) \end{aligned}$$

where now time *t* assumes values on a discrete set, typically the integers.

State space models often arise from first principle modelling. Even more naturally first principle modelling results in sets of first-order differential equations together with sets of algebraic equations in the system variables. This leads to *descriptor representations* of the form

$$\begin{aligned} E\dot{x}(t) &= Ax(t) + Bu(t) \\ y(t) &= Cx(t) + Du(t) \end{aligned}$$

where the matrix E is not necessarily invertible. Discrete-time descriptor models similarly have the form

$$\begin{aligned} Ex(t+1) &= Ax(t) + Bu(t) \\ y(t) &= Cx(t) + Du(t) \end{aligned}$$

Polynomial matrix fractions

For many applications the internal state or pseudo state x is not of interest, and only the external input and output variables u and y are relevant. It is not difficult to see that elimination of the internal variables by repeated differentiation and substitution in the continuous-time case leads to sets of differential equations in the output y and the input u that can be arranged in the form

$$Q\left(\frac{d}{dt}\right)y(t) = P\left(\frac{d}{dt}\right)u(t)$$

Q and P are polynomial matrices. For consistency Q needs to be square nonsingular.

Assuming zero initial conditions on all variables Laplace transformation of the differential equations results in the set of algebraic equations

$$Q(s)\hat{y}(s) = P(s)\hat{u}(s)$$

The circumflex denotes Laplace transformation. Solving for the output we have

$$\hat{y}(s) = Q^{-1}(s)P(s)\hat{u}(s)$$

This is where the nonsingularity of Q is needed. We see that the transfer matrix H of the system is the *polynomial matrix fraction*

$$H = Q^{-1}P$$

In the familiar single-input single-output case the matrix fraction reduces to a fraction of scalar polynomials P and Q. In the multi-input multi-output case there are many advantages in retaining the polynomial matrix fraction, without reducing it to a matrix of scalar fractions. The Polynomial Toolbox provides a comprehensive set of tools to work with such fractions.

State space systems and left polynomial matrix fractions

The Polynomial Toolbox command

```
[P,Q] = ss2lmf(A,B,C,D)
```

converts the state space representation

$$\dot{x}(t) = Ax(t) + Bu(t)$$
$$y(t) = Cx(t) + Du(t)$$

to the input-output representation

$$Q(\frac{d}{dt})y(t) = P(\frac{d}{dt})u(t)$$

Q and P are left coprime and the transfer matrix $Q^{-1}(s)P(s)$ equals the transfer matrix $C(sI - A)^{-1}B + D$ of the state space system. Q is row reduced and its row degrees are the observability indices of the state space system.

The state space representation and the input-output representation are equivalent — in the sense that they define the same sets of input-output pairs (u, y) — only if the state space system is controllable.

Example We consider the state space system

$$\dot{x} = -x$$
$$y = x + u$$

Obviously, the system is observable but not controllable. We enter its data as

```
A = 1; B = 0; C = 1; D = 1;
```

ss2lmf Conversion to a left coprime fractional representation yields

```
[Pl,Ql] = ss2lmf(A,B,C,D)
```

```
Constant polynomial matrix: 1-by-1

Pl =

     1

Constant polynomial matrix: 1-by-1

Ql =

   1
```

The polynomials `Pl` and `Ql` are obviously coprime. The input-output model corresponding to the left fraction $Q_l^{-1}P_l$ is

$$y = u$$

This model is not equivalent to the state space system because its output shows no trace of the uncontrollable but observable mode corresponding to the pole –1. We may obtain the correct model by first computing a left coprime fraction for $C(sI-A)^{-1}$ and then using this to find a non-coprime left fractional representation for the complete system:

```
[pl,Ql] = ss2lmf(A,1,C,0)

Constant polynomial matrix: 1-by-1

pl =

     1
```

```
Ql =

    -1 + s

Pl = pl*B+Dl*D

Pl =

    -1 + s
```

We thus have the correct input-output representation

$$\frac{dy(t)}{dt} - y(t) = \frac{du(t)}{dt} - u(t)$$

Left and right polynomial matrix fractions

Given a left polynomial fraction $Q_l^{-1}P_l$ there always exists a right fraction $P_rQ_r^{-1}$ so that

$$Q_l^{-1}P_l = P_rQ_r^{-1}$$

The polynomial matrices P_r and Q_r are highly nonunique because if T is any square nonsingular polynomial matrix of the same size as Q_r then

$$(P_rT)(Q_rT)^{-1} = P_rQ_r^{-1}$$

lmf2rmf, rmf2lmf

The Polynomial Toolbox routine

```
[Pr,Qr] = lmf2rmf(Pl,Ql)
```

converts the left fraction $Q_l^{-1}P_l$ to the right fraction $P_rQ_r^{-1}$ such that P_r and Q_r are coprime. Similarly, the routine

```
[Pl,Ql] = rmf2lmf(Pr,Qr)
```

converts a (not necessarily coprime) right fraction to a coprime left fraction.

Example

Conversion of the left fraction $Q_l^{-1}P_l$ of the previous example to the right fraction $P_rQ_r^{-1}$ results in

```
[Rr,Qr] = lmf2rmf(Pl,Ql)

Constant polynomial matrix: 1-by-1

Rr =

     1

Constant polynomial matrix: 1-by-1

Qr =

     1
```

Right polynomial matrix fractions and system models

ss2rmf The command

```
[P,Q] = ss2rmf(A,B,C,D)
```

implements an algorithm to obtain the right coprime fractional representation $H = PQ^{-1}$ of the transfer matrix of a state space system. Q is column reduced and its column degrees are the controllability indices of the state space system.

The right fraction representation

$$H = PQ^{-1}$$

of a transfer matrix does not correspond to an input-output model. Rather, by writing

$$\hat{y}(s) = P(s)Q^{-1}(s)\hat{u}(s)$$

and defining the "internal" signal x whose Laplace transform is $\hat{x}(s) = Q^{-1}(s)\hat{u}(s)$ we see that the right fraction corresponds to the system described by the differential equations

$$Q(\frac{d}{dt})x(t) = u(t)$$

$$y(t) = P(\frac{d}{dt})u(t)$$

The state space system is equivalent to this system only if the state space system is observable.

Example Conversion to right polynomial fraction form of the state system of the previous examples leads to

```
[Pr,Qr] = ss2rmf(A,B,C,D)

Constant polynomial matrix: 1-by-1

Pr =

     1

Constant polynomial matrix: 1-by-1

Qr =

     1
```

The transfer function $H(s)=1$ is of course correctly represented by this result but the system defined by $x=u$, $y=x$ is not equivalent to the state space system. There is no system of the form $Q(d/dt)x=u$, $y=P(d/dt)x$ that is equivalent to a system with uncontrollable but observable modes.

Polynomial matrix fraction to state space conversion

lmf2ss

Naturally the Polynomial Toolbox has routines to obtain state space representations from polynomial matrix fraction models. The command

```
[A,B,C,D] = lmf2ss(Pl,Ql)
```

constructs the observer form realization (Kailath, 1980) corresponding to the left fraction $Q_l^{-1}P_l$. The realization is always observable and controllable iff `Pl` and `Ql` are left coprime. The input-output system $Q_l(d/dt)y(t) = P_l(d/dt)u(t)$ and the state space system are equivalent.

rmf2ss

Similarly, the command

```
[A,B,C,D] = rmf2ss(Pr,Qr)
```

results in the controller form realization. It is always controllable and observable iff `Pr` and `Qr` are right coprime. The state space system is equivalent to the system defined by the differential equations $Q_r(d/dt)x(t) = u(t)$, $y(t) = P_r(d/dt)x(t)$.

Nonproper fractions and generalized state space models

Nonproper left or right polynomial matrix fractions do not have a state space representation in the strict sense. This situation is accommodated in the Polynomial Toolbox by allowing the “direct feedthrough term” in the output equation to contain differentiators. Thus, state space systems have the general form

$$\frac{d}{dt}x(t) = Ax(t) + Bu(t)$$

$$y(t) = Cx(t) + D(\frac{d}{dt})u(t)$$

with *D* a polynomial matrix.

Examples The command

```
[A,B,C,D] = lmf2ss(1+s,1+s)
```

produces the observable but obviously uncontrollable realization

```
A =

    -1

B =

     0

C =

     1

D =

     1
```

A nonproper left fraction such as

```
[A,B,C,D] = lmf2ss(1+s+s^2,2+s)
```

yields

```
A =

    -2

B =

     3

C =

     1

D =

    -1 + s
```

The corresponding state space system is

$$\frac{dx(t)}{dt} = -2x(t) + 3u(t)$$

$$y(t) = x(t) - u(t) + \frac{du(t)}{dt}$$

The non-coprime right fraction defined by

```
[A,B,C,D] = rmf2ss(1+s,1+s)
```

yields the unobservable but controllable system

```
A =

    -1

B =

     1

C =

     0

D =

     1
```

Continuous and discrete time systems

So far the discussion has been limited to continuous-time systems but the various commands apply to discrete-time systems as well.

State space to polynomial matrix fraction

The continuous-time state space system

$$\dot{x}(t) = Ax(t) + Bu(t), \; y(t) = Cx(t) + Du(t)$$

has the transfer matrix

$$C(sI-A)^{-1}B+D$$

Given the matrices *A*, *B*, *C* and *D* the routines `ss2lmf` and `ss2rmf` return the polynomial matrices that define the required polynomial matrix fraction as polynomial matrices in *s* provided that the global indeterminate variable is set to *s*. If the global variable is *p* then the notation is again consistent with a continuous-time environment.

Since the discrete-time system

$$x(t+1)=Ax(t)+Bu(t), \quad y(t)=Cx(t)+Du(t)$$

has the transfer matrix

$$C(zI-A)^{-1}B+D$$

the results returned by `ss2lmf` and `ss2rmf` fit in with a discrete-time environment if the global variable is *z*. If the global variable is *q*¸ *d* or z^{-1} then the fraction is automatically converted to a fraction in the correct variable.

Example Consider the discrete-time system defined by

```
A = 1; B = 1; C = 1; D = 0;
```

Then

```
gensym('z'); [P,Q] = ss2lmf(A,B,C,D)
```

results in

```
Constant polynomial matrix: 1-by-1

P =

     1

Q =

    -1 + z
```

while

```
gensym('z^-1'); [P,Q] = ss2lmf(A,B,C,D)
```

correctly returns

```
P =

     z^-1

Q =

     1 - z^-1
```

Polynomial matrix fraction to state space

The commands that convert polynomial fractions to a state space representation infer the nature of time from the argument of the input matrices. Thus,

```
[A,B,C,D] = lmf2ss(1,1+s)
```

yields

```
A =

    -1

B =

     1

C =

     1

D =

     0
```

but

```
[A,B,C,D] = lmf2ss(1,1+d)
```

gives

```
A =

    -1

B =

    -1

C =
     1

D =

     1
```

Also nonproper fractions are handled:

```
[A,B,C,D] = lmf2ss(1+z^2,2+z)
```

yields

```
A =

    -2

B =

     5

C =

     1

D =

  -2 + z
```

This defines the noncausal system

$$x(t+1) = -2x(t) + 5u(t)$$
$$y(t) = x(t) - 2u(t) + u(t+1)$$

On the other hand,

```
[A,B,C,D] = lmf2ss(1+d^2,2+d)
```

returns the causal system given by

```
A =

   -0.5000    1.0000

         0         0

B =

   -0.5000

    1.0000

C =

    0.5000         0

D =

  0.5000
```

Descriptor systems

Descriptor systems are described by equations of the form

$$\begin{aligned} E\dot{x}(t) &= Ax(t) + Bu(t) \\ y(t) &= Cx(t) + Du(t) \end{aligned}$$

in continuous time, and

$$\begin{aligned} Ex(t+1) &= Ax(t) + Bu(t) \\ y(t) &= Cx(t) + Du(t) \end{aligned}$$

in discrete time. In both cases the matrix E may be singular.

dss2lmf, dss2rmf, lmf2dss, rmf2dss

The commands `dss2lmf`, `dss2rmf`, `lmf2dss` and `rmf2dss` are available to convert descriptor systems to left or right fractions, and conversely.

Descriptor systems may correspond to nonproper fractions, even if the “direct feedthrough term” is non-derivative. Descriptor systems therefore always have nonderivative direct feedthrough terms.

ss2dss, dss2ss

Moreover, the commands `ss2dss` and `dss2ss` allows to convert state space systems, with or without derivative feedthrough term, to and from descriptor systems.

Examples

We may convert the nonproper left fraction given by

```
[A,B,C,D,E] = lmf2dss(1+s^2,1+s)
```

to the descriptor system specified by

```
A =

    -1     0     0

     0     1     0

     0     0     1

B =

     2

     0

     1

C =

     1    -1     0

D =

    -1

E =

     1     0     0

     0     0     1
     0     0     0
```

Internally the transformation follows by first converting to (generalized) state space form

```
[A,B,C,D] = lmf2ss(1+s^2,1+s)

A =

    -1

B =

     2

C =

     1

D =

    -1 + s
```

and from this to descriptor form

```
[A,B,C,D,E] = ss2dss(A,B,C,D)

A =

    -1     0     0
```

```
     0     1     0
     0     0     1
B =
     2
     0
     1
C =
     1    -1     0
D =
    -1
E =
     1     0     0
     0     0     1
     0     0     0
```

Conversely

```
[P,Q] = dss2lmf(A,B,C,D,E)
```

restores the matrix fraction

```
P =

      1 + s^2

Q =

      1 + s
```

Of course all this also works in discrete time. We see that the discrete-time descriptor system given by

```
A = [  0     1;   -1     0];
B = [  1;    1];
C = [  1     0 ];
D =    0;
E = [ 1     0;   0     1];
```

corresponds to the nonproper fraction

```
gensym 'd'; [P,Q] = dss2lmf(A,B,C,D,E)
P =

      d + d^2
```

```
Q =

     1 + d^2
```

or the proper fraction (in *z*)

```
gensym 'z'; [P,Q] = dss2lmf(A,B,C,D,E)
```

```
P =

     1 + z

Q =

     1 + z^2
```

Polynomial matrix fractions, rational models, and transfer function models

lmf2rat, rmf2rat, rat2lmf, rat2rmf	In rational form the transfer matrix of a system with dimensions $n \times m$ is represented by two $n \times m$ polynomial matrices `num` and `den`. The entries of `num` are the numerators of the entries of the transfer matrix, and those of `den` the denominators. The commands `lmf2rat`, `rmf2rat`, `rat2lmf` and `rat2rmf` provide conversion of polynomial matrix fractions to and from this format.
lmf2tf, rmf2tf, tf2lmf, tf2rmf	The Control Toolbox of MATLAB calls this type of model the transfer function model. It represents the numerator and denominator matrices `num` and `den` in Control Toolbox format, that is, as cell arrays of polynomials in the standard MATLAB format. These cell arrays are easily converted to and from Polynomial Toolbox

format by the commands `pol2mat` and `mat2pol`. Direct conversion of polynomial fractions to and from numerator and denominator polynomial matrices is provided by the commands `lmf2tf`, `rmf2tf`, `tf2lmf`, and `tf2rmf`.

Examples

We consider the 1×2 transfer matrix

$$H(s)=\begin{bmatrix} \dfrac{1}{1+s} & \dfrac{2+s}{s^2} \end{bmatrix}$$

The numerator and denominator matrices are (in Polynomial Toolbox format)

```
num = [ 1  2+s ];  den = [ 1+s  s^2 ];
```

The corresponding left and right coprime polynomial matrix fractions follow as

```
[Nl,Dl] = rat2lmf(num,den)

Nl =

     s^2     2 + 3s + s^2

Dl =

     s^2 + s^3

[Nr,Dr] = rat2rmf(num,den)

Nr =

     1     2 + s
```

```
Dr =

     1 + s      0

     0          s^2
```

From the latter we retrieve the transfer function model representation in Control Toolbox format according to

```
[Num,Den] = rmf2tf(Nr,Dr)

Num =

    [1x2 double]    [1x2 double]

Den =

    [1x3 double]    [1x3 double]
```

We display the results in Polynomial Toolbox format by typing

```
mat2pol(Num), mat2pol(Den)

ans =

     1      2 + s

ans =

     1 + s      s^2
```

This result follows directly by the command

```
[NUM,DEN] = rmf2rat(Nr,Dr)

NUM =

     1      2 + s

DEN =

   1 + s      s^2
```

Polynomial matrix fractions and zero-pole-gain models

The Control Toolbox of MATLAB allows a variation of the transfer function model by representing an $n\times m$ transfer matrix by two $n\times m$ cell arrays *Z* and *P* and an $n\times m$ array *K*. The entries of *Z* contain the zeros of the corresponding elements of the transfer matrix, those of *P* the poles, and those of *K* the gains. Thus, the transfer matrix

$$H(s)=\begin{bmatrix}\dfrac{1}{1+s} & \dfrac{2+s}{s^2}\end{bmatrix}$$

is represented by the zero-pole-gain model

```
Z =

     []     [-2]
```

```
P =

    [-1.0000]    [2x1 double]

K =

     1     1
```

where we have

```
P{1,2}

ans =

     0

     0
```

lmf2zpk, rmf2zpk, zpk2lmf, zpk2rmf

Polynomial fractions may be converted to and from zpk models by the Polynomial Toolbox commands `lmf2zpk`, `rmf2zpk`, `zpk2lmf` and `zpk2rmf`.

Examples

Consider the left fraction defined by

```
Nl = [ s^2 (2+s)*(1+s)];

Dl = s^2*(1+s);
```

The zpk model follows as

```
  [Z,P,K] = lmf2zpk(Nl,Dl)

  Z =

       []    [-2.0000]

  P =

      [-1]    [2x1 double]

  K =

       1     1
```

We convert this zpk model to a right polynomial matrix fraction by the command

```
[Nr,Dr] = zpk2rmf(Z,P,K)

Nr =

     1     2 + s

Dr =

     1 + s     0

     0         s^2
```

Polynomial matrix fractions and Control Toolbox LTI objects

lti2lmf, lti2rmf

The Control Toolbox of MATLAB recognizes LTI objects in four different formats: state space, descriptor system, transfer function and zero-pole-gain models. An arbitrary object in any of these formats is converted to left or right polynomial matrix fraction form by the Polynomial Toolbox commands `lti2lmf` or `lti2rmf`.

ss, dss, tf, zpk

Conversely, overloaded versions of the commands `ss`, `dss`, `tf` and `zpk` are available to create Control Toolbox objects in the corresponding formats from left or right polynomial matrix fractions.

Examples

We start the exercise by converting a left fraction into a Control Toolbox state space system

ss

```
Nl = [ s^2 (2+s)*(1+s)];

Dl = s^2*(1+s);

sys1 = ss(Nl,Dl,'l')

a =

                   x1          x2          x3

      x1           -1           1           0

      x2            0           0           1

      x3            0           0           0
```

```
b =
                        u1          u2
           x1            1           1
           x2            0           3
           x3            0           2
c =
                        x1          x2          x3
           y1            1           0           0
d =
                        u1          u2
           y1            0           0
Continuous-time system.
```

The `ss` command requires the presence of the Control Toolbox. The option `'l'` is the default and can be omitted.

lti2rmf

We convert this state space model to a right fraction according to

```
[Nr,Dr] = lti2rmf(sys1)
```

```
Constant polynomial matrix: 1-by-2

Nr =

     0     1

Dr =

     1.8 + 2.7s + 0.9s^2     1 + s

    -0.9s^2                  0
```

tf From this right fraction we manufacture a Control Toolbox `tf` object by the command

```
sys2 = tf(Nr,Dr,'r')

Transfer function from input 1 to output:

     0.9014

-----------------

0.9014 s + 0.9014


Transfer function from input 2 to output:

0.9014 s + 1.803
```

```
----------------

   0.9014 s^2
```

From this we retrieve our left fraction as

```
[Nl2,Dl2] = lti2lmf(sys2)

Nl2 =

     s^2     2 + 3s + s^2

Dl2 =

     s^2 + s^3
```

zpk The matrix fraction representation may be changed to a zero-pole-gain representation by the overloaded command

```
sys3 = zpk(Nl,Dl)

Zero/pole/gain from input 1 to output:

  1

-----

(s+1)

Zero/pole/gain from input 2 to output:
```

```
(s+2)
-----
 s^2
```

lti2rmf The zpk model may be reverted to a right fraction according to

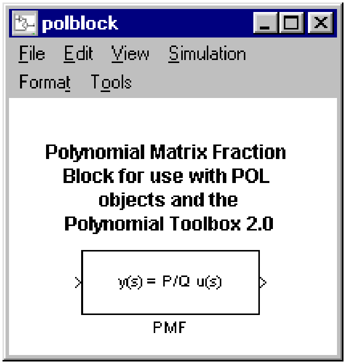
```
[Nr,Dr] = lti2rmf(sys3)
Nr =
     1     2 + s
Dr =
     1 + s     0
     0         s^2
```

SIMULINK models

The Polynomial Toolbox allows to include polynomial fraction models directly into SIMULINK models. SIMULINK is started by by typing

```
simulink
```

After opening Block sets and Toolboxes in the SIMULINK Library and double clicking POL the Polynomial Library window of Fig. 4 appears.



Polynomial Matrix Fraction
Block for use with POL
objects and the
Polynomial Toolbox 2.0

Fig. 4. The Polynomial Library Window in SIMULINK

The Polynomial Matrix Fraction block may be handled as any other Simulink built-in block to form a model. Consider the simulation of a discrete-time unit feedback system as in Fig. 5.

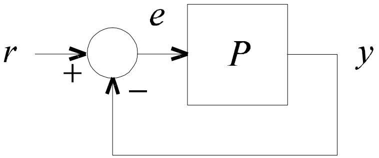


Fig. 5. Unit feedback system

The plant *P* is defined by the left polynomial matrix fraction

$$P(z) = D^{-1}(z)N(z)$$

with

```
D = z-1; N = 0.5;
```

We create the Simulink block diagram of Fig. 6 by dragging and interconnecting various standard blocks in the usual way. The parameters of the Polynomial Matrix Fraction block are defined as in Fig. 7. Finally, Fig. 8 shows the simulated step response.

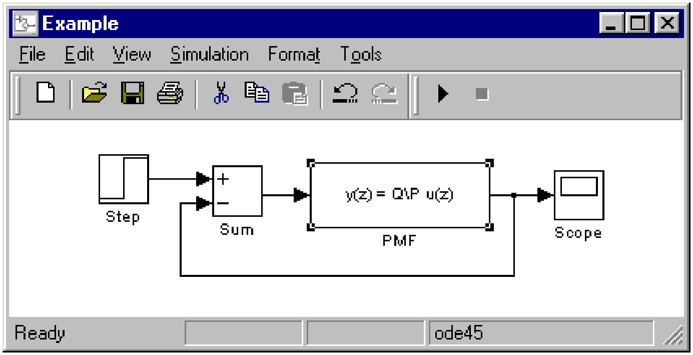
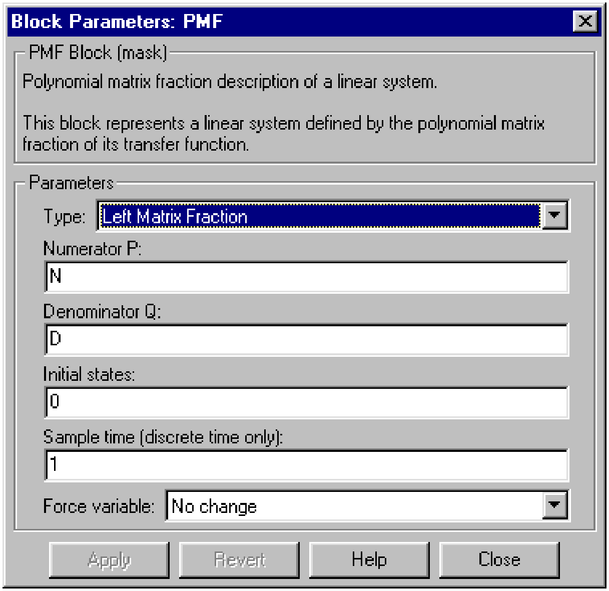


Fig. 6. SIMULINK model of the unit feedback system



PMF Block (mask)

Polynomial matrix fraction description of a linear system.

This block represents a linear system defined by the polynomial matrix fraction of its transfer function.

Parameters

Type: Left Matrix Fraction

Numerator P: N

Denominator Q: D

Initial states: 0

Sample time (discrete time only): 1

Force variable: No change

Fig. 7. Parameters of the Polynomial Matrix Fraction block

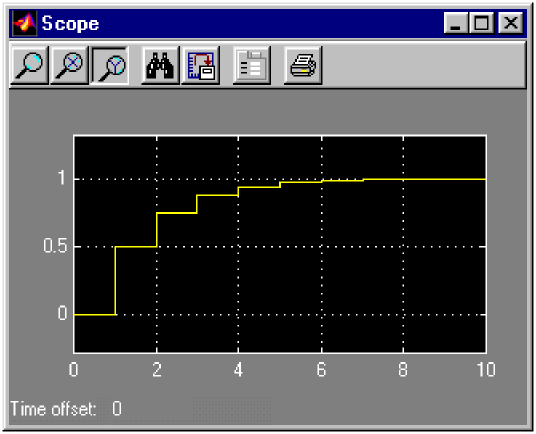


Fig. 8. Step response of the unit feedback system

Useful functions

The Polynomial Toolbox provides various functions to evaluate polynomial matrix fraction properties. Consider the left fraction $P(s) = D^{-1}(s)N(s)$ defined by

```
D = [1+s 2; 0 3];
N = [1; 1];
```

isproper

The properness and strict properness of the fraction is tested by typing

```
isproper(N,D)

ans =

     1
```

or

```
isproper(N,D,'strictly')

ans =

     0
```

h2norm, hinfnorm, norm

The H_2-norm of the fraction may be found with the help of the function

```
h2norm(N,D)

Warning: The matrix fraction is not strictly proper. H2
norm is infinite.

ans =

   Inf
```

The function call

```
norm(N,D,2)
```

```
Warning: The matrix fraction is not strictly proper. H2
norm is infinite.

ans =

   Inf
```

yields the same result. The H_∞-norm follows with the function `hinfnorm` and also by typing

```
norm(N,D,inf)

ans =

    0.4714
```

To compute the norms of right fractions the option `'r'` is available.

longldiv, longrdiv

The routines `longldiv` and `longrdiv` may be used to compute the Markov parameters of a continuous- or discrete-time system represented in left or right polynomial matrix fraction form. In the discrete-time case the Markov parameters constitute the impulse response. The command

```
[Q,R] = longldiv(N,D,5)
```

returns

```
Zero polynomial matrix: 2-by-1,  degree: -Inf

Q =
     0

     0

R(:,:,1) =

         0

    0.3333

R(:,:,2) =

    0.3333

         0

R(:,:,3) =

   -0.3333

         0

R(:,:,4) =

    0.3333

         0

R(:,:,5) =

   -0.3333
```

```
         0

R(:,:,6) =

    0.3333

         0
```

The matrices $R(:,:,j)$, $j = 2, 3, \cdots, 6$, are the first five Markov coefficients. Consult the manual page of `longldiv` for a detailed description.

gram The Gramian of a stable LTI system with transfer H is defined as

$$G = \frac{1}{2p} \int_{-\infty}^{\infty} H^T(-jw) H(jw) dw$$

in the continuous-time case, and as

$$G = \frac{1}{2p} \int_{-p}^{p} H^T(e^{-jw}) H(e^{jw})\, dw$$

in the discrete-time case. The function `gram` allows the computation of the Gramian directly from the left- or right polynomial matrix fraction of the system. We have for instance

```
gram(1,1+s)

ans =
        0.5000
```

Given a state space system $\dot{x} = Ax$, $y = Cx$ with

```
A = [-3 -4;1 0]; C = [1 1];
```

we may compute its observability Gramian according to

```
gram(C,s*eye(2)-A,'r')

ans =

        0.2083    0.1250

        0.1250    0.2083
```

Controllability Gramians may similarly be computed.

Control system design

Introduction

In the context of polynomial methods, control system design amounts to the selection of a polynomial matrix fraction description for a dynamic output feedback compensator to satisfy given specifications. This is one of the steps in industrial design that needs to be complemented with other steps such as exploratory analysis, identification, analysis, simulation, evaluation and assessment.

The Polynomial Toolbox provides several routines to solve typical design tasks. Their modifications as well as polynomial solutions to many other design problems can easily be built with the help of the basic tools of the Polynomial Toolbox.

In this chapter we successively discuss several basic control design routines, H_2 optimization, and H_∞ optimization.

Basic control routines

Introduction

The Polynomial Toolbox offers basic functions to

- stabilize the plant and, moreover, to parametrize all stabilizing controllers
- place closed-loop poles by dynamic output feedback

- design deadbeat controllers for discrete-time systems

Table 3 lists the corresponding routines.

Table 3. Basic control design routines

`stab`	Stabilization and Youla-Kucera parametrization
`pplace`	Pole placement
`debe`	Deadbeat design

Stabilization

A simple random stabilization can be achieved as follows. Given a linear time-invariant plant with transfer matrix

$$P(v) = D^{-1}(v)N(v)$$

where v can be any of the variables s, p, z, q, z^{-1} or d, the command

```
[Nc,Dc] = stab(N,D)
```

computes a stabilizing controller with transfer matrix

$$C(v) = N_C(v)D_C^{-1}(v)$$

The resulting closed-loop poles are randomly placed in the stability region, whose shape of course depends on the choice of the variable.

Youla-Kucera parametrization

For the same plant, the command

```
[Nc,Dc,E,F] = stab(N,D)
```

is used to obtain the parametrization of all stabilizing controllers in the form

$$C(v) = \left(N_C(v)P(v) + E(v)T(v)\right)\left(D_c(v)P(v) - F(v)T(v)\right)^{-1}$$

$P(v)$ is an arbitrary but stable polynomial matrix parameter of compatible size, and $T(v)$ is another (not necessarily stable) arbitrary polynomial matrix parameter of compatible size. The parameters can be chosen at will but so that the resulting controller is proper (or causal). If any common factor in $C(v)$ is cancelled then the above formula is the standard Youla-Kucera parametrization of all stabilizing controllers and $\det P(v)$ is the resulting closed-loop characteristic polynomial.

Similarly, for a plant with transfer matrix

$$P(v) = N(v)D^{-1}(v)$$

the command

```
[Nc,Dc] = stab(N,D,'r')
```

computes a stabilizing controller with transfer matrix

$$C(v) = D_C^{-1}(v)N_C(v)$$

The command

```
[Nc,Dc,E,F,degT] = stab(N,D,'r')
```

gives rise to the parametrization

$$C(v) = \left(P(v)D_c(v) - T(v)F(v)\right)^{-1}\left(P(v)N_C(v) + T(v)E(v)\right)$$

where $P(v)$ is an arbitrary but stable polynomial matrix parameter of compatible size and $T(v)$ is another (not necessarily stable) arbitrary polynomial matrix parameter of compatible size. The parameters can be chosen at will but such that the resulting controller is proper (or causal).

Example

Consider the simple continuous-time plant with defined by

```
d = 2-3*s+s^2

n = s+1
```

It has two unstable poles:

```
roots(d)

ans =

    2.0000

    1.0000
```

To obtain a stabilizing controller, type

```
[nc,dc] = stab(n,d)
```

```
nc =

     73 - 2.8s

dc =

     9.9 + s
```

Indeed, this controller gives rise to the closed-loop characteristic polynomial

```
cl = d*dc+n*nc

cl =

     93 + 43s + 4s^2 + s^3
```

and all closed-loop poles are in the left half plane:

```
roots(cl)

ans =

  -0.8297 + 6.1991i

  -0.8297 - 6.1991i

  -2.3816
```

Using the polynomial approach you can get not just one but all stabilizing controllers. Type

```
[nc,dc,e,f] = stab(n,d)

nc =

     1.4 + 6.8s

dc =

     0.98 + s

e =

     2 - 3s + s^2

f =

     1 + s
```

to get another (because of the random character of the macro) stabilizing controller, along with the parametrization of all stabilizing compensators

$$\frac{n_{\text{controller}}}{d_{\text{controller}}} = \frac{(1.396 + 6.83s)c(s) + (2 - 3s + s^2)t(s)}{(0.9801 + s)c(s) - (1 + s)t(s)}.$$

Taking $c(s) = 1$ and $t(s)$ arbitrary we get all the controllers that assign the closed-loop characteristic polynomial to be

```
m = d*dc+n*nc

m =
        3.4 + 7.3s + 4.8s^2 + s^3
```

The closed-loop poles are positioned at

```
roots(m)

ans =

   -2.1221

   -1.8176

   -0.8701
```

Similarly, for $c(s)$ fixed stable and $t(s)$ arbitrary we always obtain the closed-loop characteristic polynomial $m(s)c(s)$, unless we perform a cancellation in the controller. If $t(s)$ is chosen such that $m(s)$ cancels then the resulting closed-loop characteristic polynomial equals exactly $c(s)$.

Thus, for

```
c = (s+1)*(s+2)*(s+3);
```

and

```
t

t =

        2.525 + 3.616s + 1.17s^2
```

we obtain

```
nc1 = nc*c+e*t

nc1 =

     13.42 + 55.99s + 77.52s^2 + 42.48s^3 + 8s^4

dc1=dc*c-f*t  =

dc1 =

     3.356 + 10.64s + 12.09s^2 + 5.81s^3 + s^4
```

which are both divisible by *m*. Indeed,

```
nc2 = nc1/m

nc2 =

     4 + 8s

dc2 = dc1/m

dc2 =

     1 + s
```

Applying this controller we have

```
cl2 = d*dc2+n*nc2
```

```
cl2 =

     6 + 11s + 6s^2 + s^3

roots(cl2)

ans =

   -3.0000

   -2.0000

   -1.0000
```

which equals the desired $c(s)$.

Example Consider the three-input two-output discrete-time plant given by the transfer matrix $P(z^{-1}) = N(z^{-1})D^{-1}(z^{-1})$ with

```
N = [2 zi zi+1; 1-2*zi 0 zi]

N =

     2             z^-1     1 + z^-1

     1 - 2z^-1     0        z^-1
```

and

```
Dd = diag([2*zi+1 zi-1 1]);
```

```
D=[1 1 1; 0 1 zi; 0 0 1]*Dd*[1 0 0; zi+1 1 0; zi 1 1]

D =

     3z^-1 + z^-2     z^-1          1

    -1 + 2z^-2       -1 + 2z^-1     z^-1

     z^-1             1             1
```

The plant is stabilized by the controller $C(v) = D_C^{-1}(v) N_C(v)$ with

```
[Nc,Dc] = stab(N,D,'r')

Nc =

     16    -21 + 5.2z^-1

     21    -37 + 7.1z^-1

    -86     1.5e+002 - 29z^-1

Dc =

    -13 + 12z^-1        -2.4 - 6.4z^-1   -2.4 - 4z^-1 +
1.2z^-2

    -32 + 13z^-1         5.9 - 5.5z^-1   11 - 3.1z^-1 -
1.6z^-2

     1.3e+002 - 57z^-1  -22 + 29z^-1     -22 + 31z^-1
```

Indeed, the feedback matrix

```
Cl = Dc*D+Nc*N

Cl =

     13 + 11z^-1 + 2.5z^-2     0                  0

     0                         5.1 + 3.1z^-1      0

     0                         0                  19 + 17z^-
1
```

is stable as proved by typing

```
isstable(Cl)

ans =

     1
```

Pole placement Typically, the closed-loop poles should not only be stable but also located in prescribed positions within the stability region. The routine `pplace` takes care of this. Given the plant with transfer matrix

$$P(v) = D^{-1}(v)N(v)$$

and a vector of desired closed-loop pole locations `poles`, the command

```
[Nc,Dc] = pplace(N,D,poles)
```

computes a controller with transfer matrix

$$C(v) = N_C(v) D_C^{-1}(v)$$

which places the closed-loop poles at the locations `poles`. The multiplicity of the poles is increased if necessary. The resulting system may have real or complex coefficients depending on whether or not the desired poles are self-conjugate.

For the same plant and the same desired locations vector the command

```
[Nc,Dc,E,F,degT] = pplace(N,D,poles)
```

may be used to obtain the parametrization

$$C(v) = (N_C(v) + E(v)T(v))(D_c(v) - F(v)T(v))^{-1}$$

of all other controllers yielding the same dynamics. $T(v)$ is an arbitrary polynomial matrix parameter of compatible size and of degree bounded by `degT`.

The pole placement technique is particularly useful for single-input single-output plants. The macro does its job for multi-input multi-output systems as well but the user should be aware of the fact that assigning just pole locations does not need to be enough. In the multi-input multi-output case the desired behavior typically also depends on the closed-loop invariant polynomials rather than on the pole locations only. In fact, the assignment of invariant polynomials is very easy: all that is needed is to place the desired invariant polynomials p_i into a diagonal matrix $R(v)$ of the same size as $D(v)$ and call

```
[Nc,Dc] = pplace(N,D,R).
```

Dually, if the plant transfer matrix is given by

$$P(v) = N(v)D^{-1}(v)$$

then the command

```
[Nc,Dc] = pplace(N,D,poles,'r')
```

computes a controller transfer matrix in the form

$$C(v) = D_C^{-1}(v)N_C(v)$$

The command

```
[Nc,Dc,E,F,degT] = pplace(N,D,poles,'r')
```

gives rise to the parametrization

$$C(v) = \left(D_c(v) - T(v)F(v)\right)^{-1}\left(N_C(v) + T(v)E(v)\right)$$

$T(v)$ is an arbitrary polynomial matrix parameter of compatible size and degree of at most `degT`.

To prescribe not only he locations of the poles but also their distribution over the invariant polynomials p_i, put the poles into a diagonal matrix $R(v)$ of the same size as $D(v)$ and call

```
[Nc,Dc] = pplace(N,D,R,'r')
```

Example Consider a simple continuous-time plant described by

```
d = 2-3*s+s^2;

n = s+1;
```

The plant has two unstable poles

```
roots(d)

ans =

    2.0000

    1.0000
```

The poles may be shifted arbitrarily with a first order controller. Hence, the resulting feedback system has three poles. We place them at the locations $s_1 = -1$, $s_2 = -1 + j$ and $s_3 = -1 - j$. A controller that puts the poles at these locations results from

```
[nc,dc] = pplace(n,d,[-1,-1+j,-1-j])

nc =

     5s

dc =

     1 + s
```

and, hence, has the transfer function

$$\frac{n_c}{d_c} = \frac{5s}{1+s}.$$

Indeed,

```
r = d*dc+n*nc
r =
     2 + 4s + 3s^2 + s^3
```

and

```
roots(r)
ans =
  -1.0000 + 1.0000i
  -1.0000 - 1.0000i
  -1.0000
```

as desired. There are no other proper controllers placing poles exactly that way as

```
[nc,dc,f,g,degT] = pplace(n,d,[-1,-1+j,-1-j])
nc =
         5s

dc =

         1 + s

f =

         2 - 3s + s^2

g =

         1 + s

degT =

    -Inf
```

The parameter $T(s) = 0$ laves no degree of freedom.

Deadbeat controller

In discrete-time systems, there is one pole location of particular interest. The closed-loop system can be forced to have a finite time response from any initial condition by making the closed-loop characteristic polynomial equal to a suitable power of z or q. Equivalently, the characteristic polynomial equals 1 for systems described by a delay operator (z^{-1} or d).

The resulting performance is called *deadbeat control* and can be achieved as follows. Given a discrete-time plant with transfer matrix

$$P(z) = D^{-1}(z)N(z)$$

the command

```
[Nc,Dc] = debe(N,D)
```

computes a deadbeat controller with transfer matrix

$$C(z) = N_C(z) D_C^{-1}(z)$$

The resulting closed-loop response to any initial condition as well as to any finite length disturbance disappears in a *finite number of steps.*

Similarly, for a discrete-time plant with transfer matrix

$$P(z) = N(z) D^{-1}(z)$$

the command

```
[Nc,Dc] = debe(N,D,'r')
```

computes a deadbeat regulator with transfer matrix

$$C(z) = D_C^{-1}(z) N_C(z)$$

The macro works similarly for the other discrete-time operators *q, d* and z^{-1}.

If any other deadbeat regulators exist then they can be obtained with the parametrization

$$C(z) = (N_C(z) + E(z)T(z))(D_c(z) - F(z)T(z))^{-1}$$

which is computed by the command

```
[Nc,Dc,E,F,degT] = debe(N,D),
```

The alternative parametrization

$$C(z) = (D_c(z) - T(z)F(z))^{-1}(N_C(z) + T(z)E(z))$$

is computed by the command

```
[Nc,Dc,E,F,degT] = debe(N,D,'r').
```

$T(z)$ is an arbitrary polynomial matrix parameter of compatible size with degree limited by `degT.` Any such choice of $T(z)$ results in a proper controller yielding the desired dynamics.

If the design is being made in the backward shift operators d or z^{-1} then the degree of T is not limited. Any choice of T results in a causal controller that guarantees a finite response (with the number of steps depending on the degree, of course). In this case the output argument `degT` is useless and is returned empty.

Example

Consider a simple third-order discrete-time plant with the scalar strictly proper transfer function $P(z) = D^{-1}(z)N(z)$ given by

```
N = pol([1 1 1],2,'z')

N =

     1 + z + z^2
```

```
D = pol([4 3 2 1],3,'z')

D =

     4 + 3z + 2z^2 + z^3
```

A deadbeat regulator is designed simply by typing

```
[Nc,Dc] = debe(N,D)

Nc =

    -2.3 - 2.3z - 2.7z^2

Dc =

     0.57 + 0.71z + z^2
```

Computing the closed-loop characteristic polynomial

```
D*Dc+N*Nc

ans =

     z^5
```

reveals finite modes only and hence confirms the desired deadbeat performance.

Trying

```
[Nc,Dc,E,F,degT] = debe(N,D)
```

```
Nc =

    -2.3 - 2.3z - 2.7z^2

Dc =

     0.57 + 0.71z + z^2

E =

     4 + 3z + 2z^2 + z^3

F =

     1 + z + z^2

degT =

  -Inf
```

shows (as `degT=-Inf`) that there is no other proper deadbeat regulator such that the resulting system is of order 5. Deadbeat controllers of higher order can be found by making the design in *d* or by solving the associated Diophantine equation directly . For instance, the command

```
[Dc,Nc,F,E] = axbyc(D,N,z^6)

Dc =

    -0.47 + 0.1z + 0.25z^2 + z^3
```

```
Nc =

     1.9 - 0.88z - 1.4z^2 - 2.2z^3

F =

    -1 - z - z^2

E =

     4 + 3z + 2z^2 + z^3
```

yields a set of third order controllers parametrized by a constant *T* .

For comparison we now perform the same design in the backward-shift operator z^{-1}. To convert the plant transfer function into z^{-1}, type

```
[Nneg,Dneg] = reverse(N,D);
symbol(Nneg,'z^-1'); symbol(Dneg,'z^-1');
Nneg,Dneg
```

```
Nneg =

     z^-1 + z^-2 + z^-3

Dneg =

     1 + 2z^-1 + 3z^-2 + 4z^-3
```

Typing

```
[Ncneg,Dcneg] = debe(Nneg,Dneg)

Ncneg =

    -2.7 - 2.3z^-1 - 2.3z^-2

Dcneg =

     1 + 0.71z^-1 + 0.57z^-2
```

leads to the same regulator as previously (the only deadbeat regulator of second order). Causal regulators of higher order can be obtained from

```
[Ncneg,Dcneg,Eneg,Fneg,degTneg] = debe(Nneg,Dneg)

Ncneg =

    -2.7 - 2.3z^-1 - 2.3z^-2

Dcneg =

     1 + 0.71z^-1 + 0.57z^-2

Eneg =

     0.25 + 0.5z^-1 + 0.75z^-2 + z^-3

Fneg =
     0.25z^-1 + 0.25z^-2 + 0.25z^-3

degTneg =

     []
```

A parameter $T(z^{-1})$ of any degree may be used. For instance, the choice of $T(z^{-1})=1$ yields a third order regulator with

```
Nc_other = Ncneg+Eneg

Nc_other =

    -2.5 - 1.8z^-1 - 1.5z^-2 + z^-3

Dc_other=Dcneg-Fneg

Dc_other =

     1 + 0.46z^-1 + 0.32z^-2 - 0.25z^-3
```

The check

```
Dneg*Dc_other+Nneg*Nc_other

Constant polynomial matrix: 1-by-1

ans =

     1
```

confirms the deadbeat performance.

Example Consider the two-input two-output plant with transfer matrix $P(z) = N(z)D^{-1}(z)$ given by

```
N = [1-z z; 2-z 1]
N =
     1 - z     z
     2 - z     1
D = [1+2*z-z^2  -1+z+z^2; 2-z 2+3*z+2*z^2]
D =
     1 + 2z – z^2    -1 + z + z^2
     2 - z            2 + 3z + 2z^2
```

The deadbeat regulator is found in the form $C(z) = D_C^{-1}(z)N_C(z)$ by typing

```
[Nc,Dc] = debe(N,D,'r')
Nc =
    -3.3 - 2.7z       0.59 – 1.2z
     0.33 - 0.35z     0.41 + 0.22z
```

```
Dc =

      1.4 - z     0.39 + 0.5z

     -0.37       -0.39 + 0.5z
```

Indeed, the resulting closed-loop denominator matrix

```
Dc*D+Nc*N

ans =

      z^3     0

      0       z^3
```

reveals that only finite step modes are present

H-2 optimization

Introduction H_2 or linear-quadratic-Gaussian (LQG) control is a modern technique for designing optimal dynamic controllers. It allows to trade off regulation performance and control effort, and to take process and measurement noise into account. The Polynomial Toolbox offers the two macros listed in Table 4 for H_2 optimization by polynomial methods.

Table 4. *H*-2 optimization routines

`splqg`	Scalar *H*-2 optimization
`plqg`	Matrix *H*-2 optimization

Scalar case

The function call

```
[y,x,regpoles,obspoles] = plqg(d,n,p,q,rho,mu)
```

results in the solution of the SISO LQG problem defined by

- Response of the measured output to the control input:

$$y = P(s)u, \qquad P(s) = \frac{n(s)}{d(s)}$$

- Response of the controlled output to the control input:

$$z = Q(s)u, \qquad Q(s) = \frac{p(s)}{d(s)}$$

- Response of the measured output to the disturbance input:

$$y = R(s)v, \qquad R(s) = \frac{q(s)}{d(s)}$$

In state space form

$$\begin{aligned} \dot{x} &= Ax + Bu + Gv & \qquad P(s) &= C(sI - A)^{-1}B \\ z &= Dx & Q(s) &= D(sI - A)^{-1}B \\ y &= Cx + w & R(s) &= C(sI - A)^{-1}G \end{aligned}$$

The scalar white state noise *v* has intensity 1, and the white measurement noise *w* has intensity *μ*. The compensator *C*(*s*) = *y*(*s*)/*x*(*s*) minimizes the steady-state value of

$$E\left\{ z^2(t) + r\,u^2(t) \right\}$$

The output argument `regpoles` contains the regulator poles and `obspoles` contains the observer poles. Together the regulator and observer poles are the closed-loop poles.

Example Consider the LQG problem for the plant with transfer function

$$P(s) = \frac{10^{-4}(s^4 + 0.16s^3 + 10.0088s^2 + 0.4802s + 9.0072)}{s^2(s^4 + 0.08s^3 + 2.5022s^2 + 0.06002s + 0.56295)}$$

This is a scaled version of a mechanical positioning system discussed by Dorf, 1989 (pp. 544–546). The definition of the LQG problem is completed by choosing *Q* = *R* = *P* so that *p* = *q* = *n*. Furthermore, we let $r = 1$ and $m = 10^{-6}$.

We input the data as follows:

```
n = 1e-4*(s^4+0.16*s^3+10.0088*s^2+0.4802*s+9.0072);
```

```
d = s^2*(s^4+0.08*s^3+2.5022*s^2+0.06002*s+0.56295);
p = n; q = n; rho = 1; mu = 1e-6;
```

We can now compute the optimal compensator:

```
[y,x,regpoles,obspoles] = plqg(d,n,p,q,rho,mu)
```

MATLAB returns

```
y =
     0.9 + 34s + 7.5s^2 + 1.5e+002s^3 + 6.4s^4 + 61s^5
x =
     1 + 2.7s + 4.8s^2 + 5.5s^3 + 4.4s^4 + 1.9s^5 + s^6
regpoles =
  -0.0300 + 1.5000i
  -0.0300 - 1.5000i
  -0.0101 + 0.5000i
  -0.0101 - 0.5000i
  -0.0284 + 0.0282i
  -0.0284 - 0.0282i
```

```
obspoles =

  -0.0692 + 1.5048i

  -0.0692 - 1.5048i

  -0.6931 + 0.3992i

  -0.6931 - 0.3992i

  -0.1734 + 0.7684i

  -0.1734 - 0.7684i
```

MIMO case

Consider the linear time-invariant plant transfer matrix

$$P(v) = D^{-1}(v)N(v)$$

where v can be any of the variables s, p, z, q, z^{-1} and d. The commands

```
[Nc,Dc] = plqg(N,D,Q1,R1,Q2,R2)

[Nc,Dc] = plqg(N,D,Q1,R1,Q2,R2,'l')
```

compute an LQG optimal regulator as in Fig. 9 with transfer matrix

$$C(v) = N_C(v)D_C^{-1}(v)$$

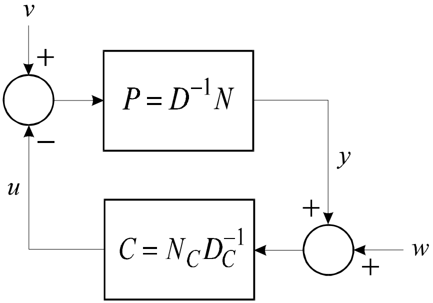


Fig. 9. LQG feedback structure

The controller minimizes the steady-state value of the expected cost

$$E\left(y^T(t)Q_2 y(t) + u^T(t)R_2 u(t)\right)$$

Q_2 and R_2 are symmetric nonnegative-definite weighting matrices. The symmetric nonnegative-definite matrices Q_1 and R_1 represent the intensities (covariance matrices) of the input and measurement white noises, respectively, and need to be nonsingular.

Examples

Example 1 (Kucera, 1991, pp. 298–303). Consider the discrete-time plant described by a left matrix fraction $D^{-1}(z)N(z)$, where

```
N = [1; 1]

D = [z-1/2 0; 0 z^2]
```

Define the nonnegative-definite matrices

```
Q1 = 1;

Q2 = [7 0; 0 7];

R1 = [0 0; 0 1];

R2 = 1;
```

The optimal LQG controller is obtained by typing

```
[Nc,Dc] = plqg(N,D,Q1,R1,Q2,R2)
```

MATLAB returns

```
Nc =

    -0.10633z^2     0

Dc =

     -0.21266z^2 - 0.85065z^3              0

     0.10633 + 0.34409z^2 - 1.3764z^3     1.1756
```

If the same plant is described by a right matrix fraction description $N(z)D^{-1}(z)$, with

```
N = [z^2; z-1/2]
```

```
D = z^2*(z-1/2)
```

then the controller results by typing

```
[Nc,Dc] = plqg(N,D,Q1,R1,Q2,R2, 'r')
```

```
 Constant polynomial matrix: 1-by-2

Nc =

     0.5     0

Dc =

     1 + 4z
```

Example 2. Consider now a continuous-time problem described by

```
N = [1; 1]

D = [s-2 0; 0 s^2+1]
```

and the weighting matrices

```
Q1 = 1;

R1 = eye(2);

Q2 = [10 0; 0 2];

R2 = 1;
```

The call

```
[Nc,Dc] = plqg(N,D,Q1,R1,Q2,R2)
```

returns the optimal LQG feedback controller

```
Nc =

    -1.3436 + 0.94635s     17.0728

Dc =

    -0.61922 + 0.12144s         5.0345 + s

     3.2092 + 2.5541s + s^2    -7.5191 - 16.4344s
```

The closed-loop poles of the optimal feedback system follow as

```
 roots(N*Nc+D*Dc)

ans =

 -3.72972478386939

 -2.22996241633438

 -0.42364369708415 + 1.07974078889846i

 -0.42364369708415 - 1.07974078889846i

 -0.38868587951944 + 1.05190312987575i
```

```
-0.38868587951944 - 1.05190312987575
```

H-infinity optimization

Introduction

H_∞ optimization is a powerful modern tool. It allows the design of high-performance and robust control systems. The Polynomial Toolbox offers two routines for H_∞ design:

- Mixed sensitivity optimization of SISO systems relying on transfer function descriptions
- A routine `dsshinf` for finding all suboptimal solutions of the general standard H_∞ optimization problem based on descriptor representations
- A very comprehensive routine `dssrch` for finding optimal solutions of the general standard H_∞ optimization problem based on descriptor representations

SISO mixed sensitivity optimization

The SISO mixed sensitivity problem consists of minimizing the square root of

$$\sup_{w \in \mathrm{R}} |V(jw)|^2 \left(|W_1(jw)S(jw)|^2 + |W_2(jw)U(jw)|^2 \right)$$

where

$$S = \frac{1}{1+PC}, \qquad U = \frac{C}{1+PC}$$

are the sensitivity and input sensitivity functions of the closed-loop system of Fig. 10, respectively.

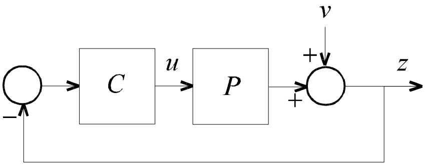


Fig. 10. SISO closed-loop system

The plant transfer function is

$$P(s) = \frac{n(s)}{d(s)}$$

The rational function

$$V(s) = \frac{m(s)}{d(s)}$$

is a shaping filter, and the rational functions

$$W_1(s) = \frac{a_1(s)}{b_1(s)}, \quad W_2(s) = \frac{a_2(s)}{b_2(s)}$$

are weighting filters. The input parameters n, d, m, a_1, b_1, a_2 and b_2 are scalar polynomials or constants. The polynomial m needs to have the same degree as d. The polynomials m, b_1 and b_2 are required to be strictly Hurwitz.

An important property of the solution of the mixed sensitivity problem is that the roots of the polynomial m return as closed-loop poles (Kwakernaak, 1993). Hence, choosing m amounts to pre-assigning as many closed-loop poles as m has roots. If these poles and also the weighting functions W_1 and W_2 are well chosen then these pre-assigned closed-loop poles are the dominant closed-loop poles and, hence, to a large extent determine the closed-loop bandwidth and other important closed-loop properties.

The SISO mixed sensitivity problem is solved by the command

```
[y,x,gopt] = mixeds(n,m,d,a1,b1,a2,b2,gmin,gmax,accuracy)
```

The input parameters `gmin` and `gmax` are lower and upper bounds for the minimal value of the mixed sensitivity criterion, respectively. The parameter `accuracy` specifies how closely the minimal norm is to be approached.

Several optional input parameters that allow more control over the algorithm are described on the manual page for `mixeds`.

The algorithm is based on the polynomial solution of the standard H_∞optimization problem (Kwakernaak, 1996). The optimal solution is approached by a binary search algorithm. After the optimal solution has been found the compensator is simplified by cancelling any common roots of the numerator and denominator of its transfer function.

Example Consider the mixed sensitivity problem for the plant with transfer function

$$P(s) = \frac{n(s)}{d(s)} = \frac{1}{s^2}$$

defined by

$$V(s) = \frac{m(s)}{d(s)} = \frac{1+s\sqrt{2}+s^2}{s^2}, \quad W_1(s) = \frac{a_1(s)}{b_1(s)} = 1, \quad W_2(s) = \frac{a_2(s)}{b_2(s)} = c(1+rs)$$

We let $c = 0.1$, $r = 0.1$. First define the input parameters:

```
c = 0.1; r = 0.1; n = 1; d = s^2; m = s^2+s*sqrt(2)+1;
a1 = 1; b1 = 1; a2 = c*(1+r*s); b2 = 1;
gmin = 1; gmax = 10; accuracy = 1e-4;
```

Next we call the routine `mixeds`:

```
[y,x,gopt] = mixeds(n,m,d,a1,b1,a2,b2,gmin,gmax,accuracy)
```

The routine returns

```
y =

     57 + 96s

x =

     79 + 16s + s^2
```

```
gopt =

  1.3834
```

The minimal value of the criterion is 1.3834 and the optimal compensator has the transfer function

$$C(s) = \frac{y(s)}{x(s)} = \frac{57 + 96s}{79 + 16s + s^2}$$

It is easy to compute the closed-loop poles of the optimal system:

```
clpoles = roots(d*x+n*y)

clpoles =

   -7.3282 + 1.8769i

   -7.3282 - 1.8769i

   -0.7071 + 0.7071i

   -0.7071 - 0.7071i
```

The commands

```
omega = logspace(-2,2);

S = bode(pol2mat(d*x),pol2mat(d*x+n*y),omega);

T = bode(pol2mat(n*y),pol2mat(d*x+n*y),omega);
```

invoke a well-known routine of the Control Toolbox to compute the magnitudes of the sensitivity function and complementary sensitivity function

$$S = \frac{1}{1+PC} = \frac{dx}{dx+ny}, \quad T = \frac{PC}{1+PC} = \frac{ny}{dx+ny}$$

respectively. The sensitivity functions may be plotted by the commands

```
subplot(1,2,1), loglog(omega,S), axis([1e-2 1e2 1e-4 10])
ylabel('S'), xlabel('omega'), grid
subplot(1,2,2), loglog(omega,T), axis([1e-2 1e2 1e-4 10])
ylabel('T'), xlabel('omega'), grid
```

Fig. 11 shows the plots. The sensitivity functions are very well behaved and predict excellent performance and good robustness.

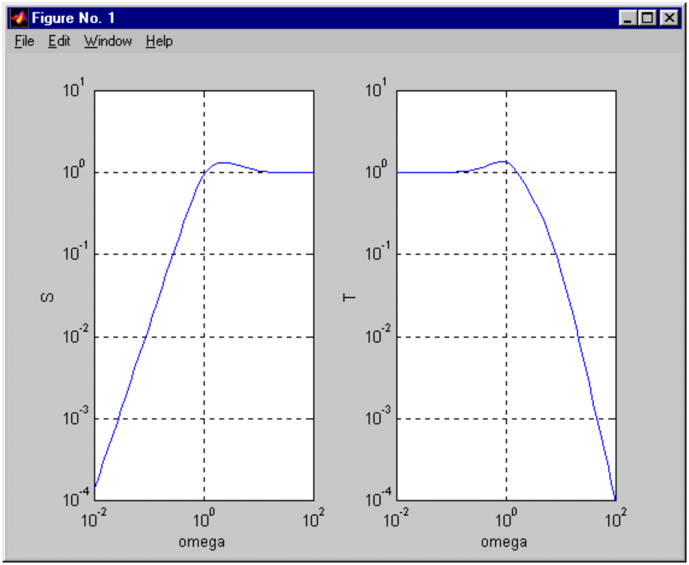


Fig. 11. Sensitivity and complementary sensitivity functions

Descriptor solution of the standard *H*-infinity problem

The standard H_∞ optimization problem is characterized by the block diagram of Fig. 12. The external input *v* represents the driving signals for disturbances, measurement noise and reference inputs, while *u* is the control input. The output *z* is the error signal, and *y* the observed signal that is available for feedback.

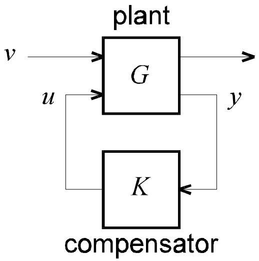


Fig. 12. The standard problem

If the generalized plant *G* is represented in transfer matrix form as

$$G(s) = \begin{bmatrix} G_{11}(s) & G_{12}(s) \\ G_{21}(s) & G_{22}(s) \end{bmatrix}$$

and the compensator has the transfer matrix $K(s)$, then the closed-loop system $\hat{z}(s) = H(s)\hat{v}(s)$ has the transfer matrix

$$H(s) = G_{11}(s) + G_{12}(s)[I - K(s)G_{22}(s)]^{-1}K(s)G_{21}(s)$$

The H_∞ problem is the problem of finding a compensator K that minimizes the ∞-norm $\|H\|_\infty$ of the closed-loop transfer matrix.

The state space solution of the H_∞ problem is very well known. The Polynomial Toolbox offers an implementation of an algorithm for a more general problem, where the generalized plant is represented in the descriptor form

$$E\dot{x} = Ax + B\begin{bmatrix} v \\ u \end{bmatrix}$$

$$\begin{bmatrix} z \\ y \end{bmatrix} = Cx + D\begin{bmatrix} v \\ u \end{bmatrix}$$

Since descriptor systems may have nonproper transfer matrices this formulation makes it possible to consider and solve H_∞ problems with nonproper weighting functions. Nonproper weighting functions arise for instance when it is desired to control the high-frequency roll-off of the closed-loop transfer matrix.

Suboptimal solutions

The command

```
[Ak,Bk,Ck,Dk,Ek,clpoles] = ...

      dsshinf(A,B,C,D,E,nmeas,ncon,gamma)
```

computes a suboptimal compensator, if any exists, such that

$$\|H\|_\infty \leq g$$

with g (`gamma`) a given nonnegative number. The input parameter `nmeas` is the number of measured variables (the dimension of y) and `ncon` the number of control inputs (the dimension of u). The suboptimal compensator is returned in descriptor form and has the transfer matrix

$$K(s) = C_k(sE_k - A_k)^{-1}B_k + D_k$$

Example

We consider the standard H_∞ problem defined by the mixed sensitivity problem that we previously studied in this section. The block diagram of Fig. 13 shows the standard plant that defines the mixed sensitivity problem.

It is easy to see from the block diagram that the generalized plant transfer matrix is

$$G(s) = \left[\begin{array}{c|c} W_1(s)V(s) & W_1(s)P(s) \\ 0 & W_2(s) \\ \hline -V(s) & -P(s) \end{array}\right]$$

Substituting for the various transfer functions we obtain

$$G(s) = \left[\begin{array}{c|c} \dfrac{a_1(s)m(s)}{b_1(s)d(s)} & \dfrac{a_1(s)n(s)}{b_1(s)d(s)} \\ 0 & \dfrac{a_2(s)}{b_2(s)} \\ \hline -\dfrac{m(s)}{d(s)} & -\dfrac{n(s)}{d(s)} \end{array}\right]$$

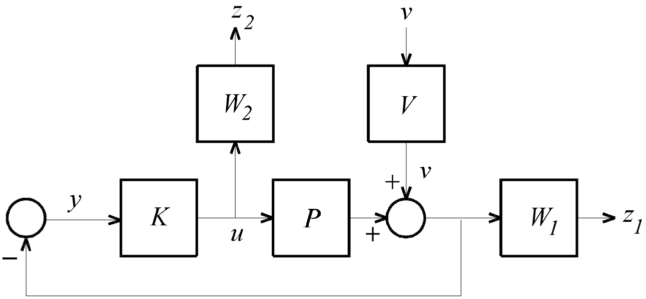


Fig. 13. Block diagram for the mixed sensitivity problem

It may be checked that *G* is given by the left coprime polynomial matrix fraction

$$G(s) = \begin{bmatrix} b_1(s) & 0 & -a_1(s) \\ 0 & b_2(s) & 0 \\ 0 & 0 & d(s) \end{bmatrix}^{-1} \begin{bmatrix} 0 & 0 \\ 0 & a_2(s) \\ -m(s) & -n(s) \end{bmatrix}$$

After defining the data we convert this fraction to descriptor form by the Polynomial Toolbox routine `lmf2dss`:

```
c = 0.1; r = 0.1; n = 1; d = s^2; m = s^2+s*sqrt(2)+1;

a1 = 1; b1 = 1; a2 = c*(1+r*s); b2 = 1;

Dg = [b1 0 -a1; 0 b2 0; 0 0 d];

Ng = [0 0; 0 a2; -m -n];
```

```
[A,B,C,D,E] = lmf2dss(Ng,Dg)
```

The output is

```
A =

     0     1     0     0
     0     0     0     0
     0     0     1     0
     0     0     0     1

B =

   -1.4142         0
   -1.0000   -1.0000
         0         0
         0    0.0100

C =

     1     0     0     0
     0     0    -1     0
     1     0     0     0
```

```
D =

   -1.0000         0
         0    0.1000
   -1.0000         0

E =

     1     0     0     0
     0     1     0     0
     0     0     0     1
     0     0     0     0
```

To compute a suboptimal compensator according to the level $g = 10$ we type

```
[Ak,Bk,Ck,Dk,Ek,clpoles] = dsshinf(A,B,C,D,E,1,1,10)

Ak =

    0.0000    0.0000    0.0000    0.0152
   -5.2085    0.8018   -2.4004    4.8467
   -4.6877    1.0045   -4.3637    6.8464
   -0.7066   -0.1166   -0.6520   -0.0449
```

```
Bk =

        0

   0.3974

  -1.6859

  -0.0000

Ck =

  -7.0313    1.1090   -4.5268    7.6311

Dk =

    0

Ek =

        0         0         0         0

   0.5155    0.0100   -0.6605    0.4651

   0.4783   -0.2523   -0.2453   -0.5436

   0.0703   -0.0111    0.0453   -0.0611

clpoles =

 -10.0526 + 0.0000i
```

```
  -2.1935 - 2.3064i
  -2.1935 + 2.3064i
  -0.7071 - 0.7071i
  -0.7071 + 0.7071i
```

To obtain the compensator $K(s) = y(s)/x(s)$ in transfer function form we use the command

```
[y,x] = dss2rmf(Ak,Bk,Ck,Dk,Ek)

y =

     1e+002 + 2e+002s

x =

     1.9e+002 + 76s + 16s^2 + s^3
```

Note that the compensator transfer function is proper. Therefore, the compensator has a state space representation. This state space representation follows with the help of the function `dssmin`, which minimizes the dimension of the descriptor representation of the compensator:

```
[ak,bk,ck,dk,ek] = dssmin(Ak,Bk,Ck,Dk,Ek)

ak =

  -10.3436   -3.2399    5.7654
```

```
         0   -4.9939    3.8368
         0   -4.1904   -0.5165
bk =
    1.6975
   -6.6721
   -4.9036
ck =
   -7.0231   -3.7929    2.7296
dk =
     0
ek =
     1     0     0
     0     1     0
     0     0     1
```

Note that ek is the 3×3 unit matrix so that we have a three-dimensional state space representation.

Optimal solutions

To find H_∞optimal — as opposed to suboptimal — solutions it is necessary to embed the routine `dsshinf` in a search loop that performs a line search on the number g . Other algorithms that have been implemented suffer from the disadvantage that close to the optimal solution the solutions often become degenerate. The algorithm `dsshinf` used in the macro `dssrch` of the Polynomial Toolbox avoids these difficulties, and, indeed, exploits the singularity at the optimum to accelerate the search by switching from binary search to a secant search method near the optimum.

Features of the macro `dssrch`

1. If no solution exists within the initial search interval [`gmin`,`gmax`] then the interval is automatically expanded.
2. The macro also computes optimal solutions for generalized plants that have unstable fixed poles.
3. The macro computes both type 1 optimal solutions (`gopt` coincides with the lowest value of `gamma` for which spectral factorization is possible) and type 2 optimal solutions (`gopt` is greater than this lowest value).
4. After the search has been completed the compensator is reduced to minimal dimension. If the compensator is proper then E_k is the unit matrix so that it is in state space form.
5. Initially the search is binary. If the solution is of type 2 and close to optimal then a secant method is used to obtain fast convergence to an accurate solution.

6. Before the search is initiated the descriptor representation of the generalized plant is regularized in the sense that it is modified so that the subblocks D_{12} and D_{21} of D have full rank.

The command line is of the form

```
[Ak,Bk,Ck,Dk,Ek,gopt,clpoles] = ...
               dssrch(A,B,C,D,E,nmeas,ncon,gmin,gmax)
```

Example We apply the optimization algorithm to the example that we looked at earlier in this section. Typing

```
[Ak,Bk,Ck,Dk,Ek,gopt,clpoles] =  dssrch(A,B,C,D,E,1,1,1,10)
```

yields

```
Ak =

   -9.9335   -2.6876

    6.6932   -6.1368

Bk =

  -15.1682

  -14.6069

Ck =
   -7.6954    1.4476

Dk =

  1.0217e-005

Ek =

     1     0

     0     1

gopt =

    1.3833

clpoles =

  -7.3281 + 1.8765i

  -7.3281 - 1.8765i

  -0.7071 + 0.7071i

  -0.7071 - 0.7071i
```

The optimal compensator is in state space form. To see it in transfer function form we use the command

```
[y,x] = dss2rmf(Ak,Bk,Ck,0,Ek)
```

Note that we have set the small direct feedthrough coefficient D_k equal to 0. The command yields the compensator that we also obtained earlier with the routine `mixeds`:

```
y =

     57 + 96s

x =

     79 + 16s + s^2
```

Robust control with parametric uncertainties

Introduction

Modern control theory addresses various problems involving uncertainty. A mathematical model of a system to be controlled typically includes uncertain quantities. In a large class of practical design problems the uncertainty may be attributed to certain coefficients of the plant transfer matrix. The uncertainty usually originates from physical parameters whose values are only specified within given bounds. An ideal solution to overcome the uncertainty is to find a *robust controller* — a simple, fixed controller, designed off-line, which guarantees desired behavior and stability for all expected values of the uncertain parameters.

The Polynomial Toolbox offers several simple tools that are useful for robust control analysis and design for systems with parametric uncertainties. The relevant macros are briefly introduced in this chapter. More details on the underlying methods as well as other solutions that can also be built from Polynomial Toolbox macros are described in Barmish (1996), Bhattacharya, Chapellat and Keel (1995) and other textbooks.

Single parameter uncertainty

Many systems of practical interest involve a single uncertain parameter. At the time of design the parameter is only known to lie within a given interval. Quite often even more complex problems (with a more complex uncertainty structure) may be reduced to the single parameter case. Needless to say that the strongest results are available for this simple case.

Even though the uncertain parameter is single it may well appear in several coefficients of the transfer matrix at the same time. Quite in the spirit of the Polynomial Toolbox the coefficients are assumed to be polynomial functions of the uncertain parameter.

Example 1 *Robust stability interval.* To analyze the single parameter uncertain polynomial

$$p(s,q) = 3 + (10 + q)s + 12s^2 + (6 + q)s^3 + s^4$$

first check whether $p(s,q)$ is stable for $q = 0$. Then find its left-sided and right-sided stability margins, that is, the smallest negative $q_{\min}$ and the largest positive $q_{\max}$ such that $p(s,q)$ remains stable for any $q \in (q_{\min}, q_{\max})$.

With the Polynomial Toolbox this is an easy task: First express the given polynomial as

$$p(s,q) = p_0(s) + qp_1(s)$$

and enter the data

```
p0 = 3 + 10*s + 12*s^2 + 6*s^3 + s^4;

p1 = s + s^3;
```

Then type

```
isstable(p0)

ans =

     1
```

to verify nominal stability (that is, stability for $q = 0$). Finally, call

```
[qmin,qmax] = stabint(p0,p1)

qmin =

   -5.6277

qmax =

   Inf
```

to determine the stability margins. This result discloses that $p(s,q)$ is not merely stable for $q = 0$ but also for all $q \in (-5.6277, \infty)$. When $q = -5.6277$ stability is lost. Nothing is claimed for $q < -5.6277$, however, only that stability is not guaranteed.

If you have also the Control System Toolbox available then you can combine it with the Polynomial Toolbox and visualize the result by plotting the root-locus of a fictitious plant $p_1(s)/p_0(s)$ under a fictitious feedback gain q ranging over $(-5.6277,\infty)$. Typing

```
rlocus(ss(p1,p0),qmin:.1:100)
```

produces the root locus plot of Fig. 14. It confirms that all the roots of $p(s,q)=p_0(s)+qp_1(s)$ stay inside the stability region for all $q\in(-5.6277,100)$. Note also the role of the macro `ss`, which converts the polynomial fraction into the Control System Toolbox state-space format.

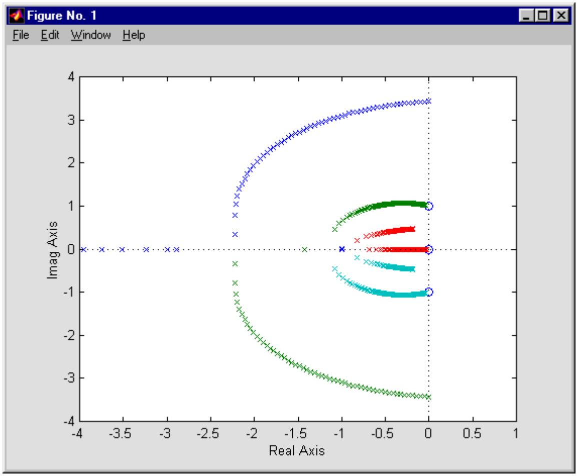


Fig. 14. Root locus plot

Example 2 *Robust stabilization.* Consider the plant with transfer matrix

$$D^{-1}(s,q)N(s,q)=\begin{bmatrix} s^2 & q \\ 1+q^2 & s \end{bmatrix}^{-1}\begin{bmatrix} 1+s & 0 \\ q & 1 \end{bmatrix}=\begin{bmatrix} \dfrac{-q^2+s+s^2}{-q-q^2+s^3} & \dfrac{-q}{-q-q^2+s^3} \\ \dfrac{-1-q^2-(1+q^2)s+qs^2}{-q-q^2+s^3} & \dfrac{s^2}{-q-q^2+s^3} \end{bmatrix}$$

which depends on an uncertain parameter q. Suppose that q may take any value in the interval $[0,1]$ and that its nominal value is $q_0=0$. The plant is described by a left-sided fraction of polynomial matrices in two variables: $D(s,q)$ and $N(s,q)$ that may be written as

$$D(s,q)=D_0(s)+qD_1(s)+q^2D_2(s)=\begin{bmatrix} s^2 & 0 \\ 0 & s \end{bmatrix}+q\begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}+q^2\begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix}$$

and

$$N(s,q)=N_0(s)+qN_1(s)=\begin{bmatrix} 1+s & 0 \\ 0 & s \end{bmatrix}+q\begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}$$

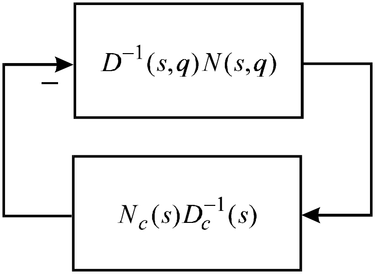


Fig. 15. Robust control structure

If a feedback controller with transfer matrix

$$N_c(s)D_c^{-1}(s)$$

is applied as in Fig. 15 then the resulting closed-loop denominator matrix is

$$P(s,q) = D(s,q)D_c(s) + N(s,q)N_c(s)$$

The denominator matrix may also be expressed as

$$\begin{aligned} P(s,q) &= P_0(s) + qP_1(s) + q^2P_2(s) \\ &= \left(D_0(s)D_c(s) + N_0(s)N_c(s)\right) + q\left(D_1(s)D_c(s) + N_1(s)N_c(s)\right) + q^2\left(D_2(s)D_c(s)\right) \end{aligned}$$

To enter the data, type

```
D0 = [ s^2 1; 1 s ];
```

```
D1 = [ 0 1; 0 0 ];
D2 = [ 0 0; 1 0 ];
```

and

```
N0 = [ 1+s 0; 0 1 ];
N1 = [ 0 0; 1 0 ];
```

Nominally (that is, for $q = 0$), the transfer matrix

$$D^{-1}(s,0)N(s,0) = D_0^{-1}(s)N_0(s) = \begin{bmatrix} s^2 & 1 \\ 1 & s \end{bmatrix}^{-1} \begin{bmatrix} 1+s & 0 \\ 0 & 1 \end{bmatrix}$$

is unstable because

```
roots(D0)
ans =
  -0.5000 + 0.8660i
  -0.5000 - 0.8660i
   1.0000
```

To stabilize the nominal plant, call

```
[Nc1,Dc1] = stab(N0,D0)
```

```
Nc1 =
     51 - 30s    -2.7 + 1.7s
    -38           4.1
Dc1 =
     38 + s    -1.7
    -1          2.7 + s
```

This controller gives rise to the feedback denominator matrix defined by

```
P0 = D0*Dc1+N0*Nc1, P1 = D1*Dc1+N1*Nc1, P2 = D2*Dc1
P0 =
     50 + 21s + 7.9s^2 + s^3     0
     0                           2.4 + 2.7s + s^2
P1 =
    -1              2.7 + s
     51 - 30s    -2.7 + 1.7s
P2 =
     0          0
```

```
        38 + s    -1.7
```

This denominator is nominally stable, as expected, because

```
roots(P0)

ans =

  -5.8119

  -1.0611 + 2.7264i

  -1.0611 - 2.7264i

  -1.3423 + 0.7885i

  -1.3423 - 0.7885i
```

To check robust stability simply type

```
[qmin,qmax] = stabint(P0,P1,P2)

qmin =

   -1.2068

qmax =

    0.4658
```

This result reveals that the closed-loop system only remains stable on the interval $q \in (-1.2068,\ 0.4658)$, which does not include the entire desired interval $[0,1]$. Hence, the controller is nominally but *not* robustly stabilizing. Let us try another one:

```
[Nc2,Dc2] = stab(N0,D0)

Nc2 =

     94 - 51s     -18 + 17s

    -55             1e+002

Dc2 =

     55 + s     -17

    -1          18 + s
```

This second controller yields

```
P0 = D0*Dc2+N0*Nc2, P1 = D1*Dc2+N1*Nc2, P2 = D2*Dc2

P0 =

     93 + 43s + 4s^2 + s^3      0

     0                          86 + 18s + s^2

P1 =
    -1                18 + s

     94 - 51s        -18 + 17s

P2 =

     0          0

     55 + s    -17
```

Again, as expected the controller is nominally stabilizing:

```
roots(P0)

ans =

  -8.8745 + 2.6175i

  -8.8745 - 2.6175i

  -0.8297 + 6.1991i

  -0.8297 - 6.1991i

  -2.3816
```

Its robust stability interval is

```
[qmin,qmax] = stabint(P0,P1,P2)

qmin =
   -0.9344

qmax =

    1.1700
```

Because $[0,1] \subset [-0.9344, 1.1700]$ the second controller evidently guarantees stability on the whole required uncertainty-bounding interval. Hence, it is the desired *robustly stabilizing* controller.

Interval polynomials

Another important class of uncertain systems is described by interval polynomials with independent uncertainties in the coefficients. An interval polynomial looks like

$$p(s,q) = \sum_{i=0}^{n} \left[q_i^-, q_i^+ \right] s^i$$

with $[q_i^-, q_i^+]$ denoting the bounding interval for the *i*th coefficient. Using the Polynomial Toolbox, it is convenient to describe interval polynomials by their “lower” and “upper” elements

$$p^-(s) = \sum_{i=0}^{n} q_i^- s^i \quad \text{and} \quad p^+(s) = \sum_{i=0}^{n} q_i^+ s^i$$

In many applications interval polynomials arise when an original uncertainty structure is known but too complex (e.g., highly nonlinear) to be tractable but may be “overbounded” by a simple interval once an independent uncertainty structure is imposed.

Example 3 *Graphical Method.* Consider the continuous-time interval polynomial (Barmish, 1996)

$$p(s,q) = [0.45, 0.55] + [1.95, 2.05]s + [2.95, 3.05]s^2 + [5.95, 6.05]s^3 + [3.95, 4.05]s^4 \\ + [3.95, 4.05]s^5 + s^6$$

The first step in the graphical test for robust stability requires establishing that at least one polynomial in the family is stable. Using the midpoint of each of the intervals we obtain

```
p_mid = pol([0.5 2 3 6 4 4 1],6)

p_mid =

     0.5 + 2s + 3s^2 + 6s^3 + 4s^4 + 4s^5 + s^6

isstable(p_mid)

ans =

     1
```

Next we enter the given interval polynomial in terms of two “lumped” polynomials

```
pminus =
0.45+1.95*s+2.95*s^2+5.95*s^3+3.95*s^4+3.95*s^5+s^6;
```

and

```
pplus =
0.55+2.05*s+3.05*s^2+6.05*s^3+4.05*s^4+4.05*s^5+s^6;
```

Using these polynomials we plot the sets $p(jw,q)$, consisting of what are called the "Kharitonov rectangles" for $0 \leq w \leq 1$ using the command

```
khplot(pminus,pplus,0:.001:1)
```

This results in Fig. 16. Since none of the rectangles touches the point $z = 0$ the Zero Exclusion Condition $0 \notin p(jw,q)$ is satisfied, and we conclude that the interval polynomial is robustly stable. Note that as long as all the polynomial coefficients are real numbers we only need to investigate $w \geq 0$. The plot for $w \leq 0$ is symmetric as $p(-jw,q) = \overline{p}(jw,q)$.

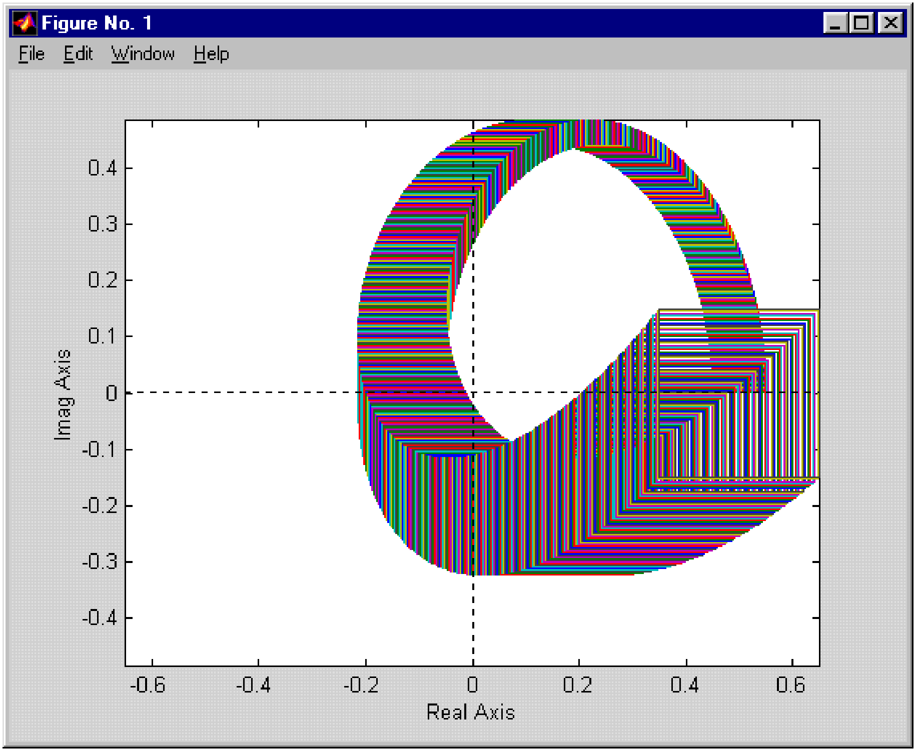


Fig. 16. Kharitonov rectangles

Example 4 *Test Using Kharitonov Polynomials.* For continuous-time interval polynomials we have an even simpler method available: An interval polynomial of invariant degree (with real coefficients) is known to be stable if and only if just its four "extreme" polynomials (called the Kharitonov polynomials)

$$\begin{aligned}
K_1(s) &= q_0^- + q_1^- s + q_2^+ s^2 + q_3^+ s^3 + q_4^- s^4 + q_5^- s^5 + q_6^- s^6 + \cdots;\\
K_2(s) &= q_0^+ + q_1^+ s + q_2^- s^2 + q_3^- s^3 + q_4^+ s^4 + q_5^+ s^5 + q_6^- s^6 + \cdots;\\
K_3(s) &= q_0^+ + q_1^- s + q_2^- s^2 + q_3^+ s^3 + q_4^+ s^4 + q_5^- s^5 + q_6^- s^6 + \cdots;\\
K_4(s) &= q_0^- + q_1^+ s + q_2^+ s^2 + q_3^- s^3 + q_4^- s^4 + q_5^+ s^5 + q_6^+ s^6 + \cdots;
\end{aligned}$$

are stable. For the interval polynomial of Example 3 the Kharitonov polynomials are computed by

```
[stability,K1,K2,K3,K4] = kharit(pminus,pplus)

stability =

     1

K1 =

     0.45 + 1.95s + 3.05s^2 + 6.05s^3 + 3.95s^4 + 3.95s^5 +
s^6

K2 =
     0.55 + 2.05s + 2.95s^2 + 5.95s^3 + 4.05s^4 + 4.05s^5 +
s^6

K3 =

     0.55 + 1.95s + 2.95s^2 + 6.05s^3 + 4.05s^4 + 3.95s^5 +
s^6

K4 =

     0.45 + 2.05s + 3.05s^2 + 5.95s^3 + 3.95s^4 + 4.05s^5 +
s^6
```

The macro also checks the stability of the Kharitonov polynomials. The resulting value of `stability` confirms that all the four polynomials are stable and we conclude that the interval polynomial is robustly stable.

Example 5 *Robust stability of discrete-time interval polynomials*. For discrete-time polynomials (of degree 4 and higher), the Kharitonov-like extremal results are not available. However, the graphical method may be applied for discrete-time polynomials as well as for other stability regions.

Consider the interval polynomial

$$p(z,q)=[10,20]+[20,30]z+[128,138]z^2+[260,270]z^3+168z^4.$$

To test its robust stability, we write

$$p(z,q)=p_0(z)+q_1p_1(z)+q_2p_2(z)+q_3p_3(z)+q_4p_4(z),$$

where $p_0(z) = 10 + 20z + 128z^2 + 260z^3 + 168z^4$, $p_1(z) = 1$, $p_2(z) = z$, $p_3(z) = z^2$ and $p_4(z) = z^3$. Such an expression is called *polytopic* and will be discussed later in a more general setting. For the moment it enables us to describe each interval coefficient by a separate uncertain parameter q_i ranging over $[0,10]$.

To analyze the interval polynomial we first enter the data

```
p0 = 10 + 20*z + 128*z^2 + 260*z^3 + 168*z^4;

p1 = 1; p2 = z; p3 = z^2; p4 = z^3;

Qbounds = [ 1 10; 1 10; 1 10; 1 10 ];
```

Next we check that p_0 is stable

```
isstable(p0)

ans =

     1
```

Then we plot the value sets $p(c,q)$ but now with c sweeping around the unit circle. Note that the value sets no longer have a rectangular shape and we must use a more general command

```
ptopplot(p0,p1,p2,p3,p4,Qbounds,exp(j*(0:0.001:1)*2*pi))
```

to obtain the plot of Fig. 17.

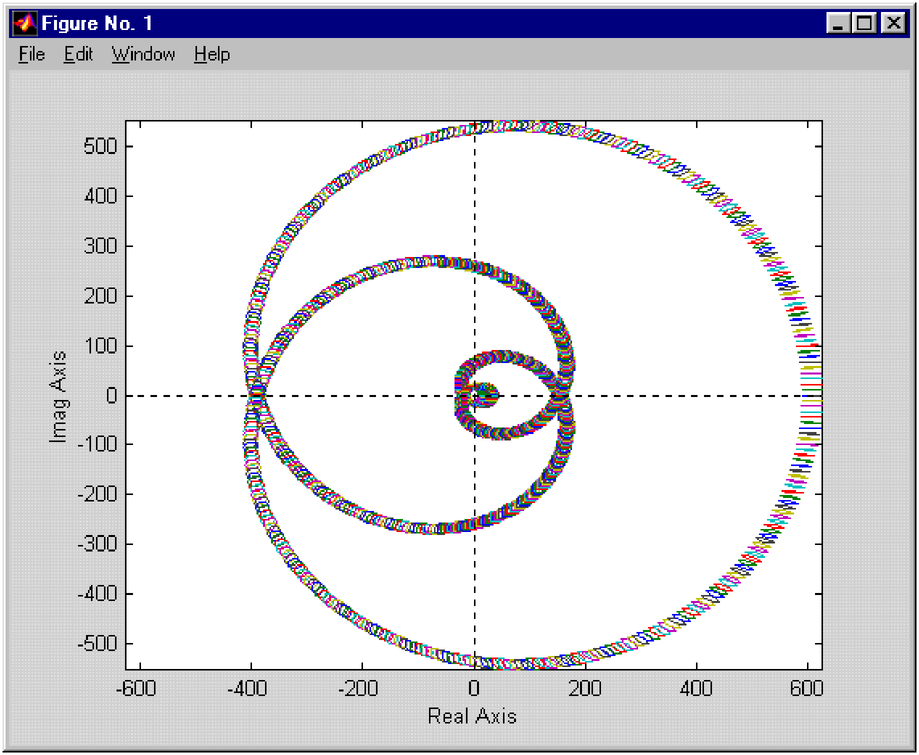


Fig. 17. Value sets

To see better what is happening in the neighborhood of the point 0 we zoom the graph for generalized frequencies e^{jw} in the critical range $w \in (0.6p, 1.4p)$:

```
ptopplot(p0,p1,p2,p3,p4,Qbounds,exp(j*(0.3:0.001:0.7)*2*pi))
```

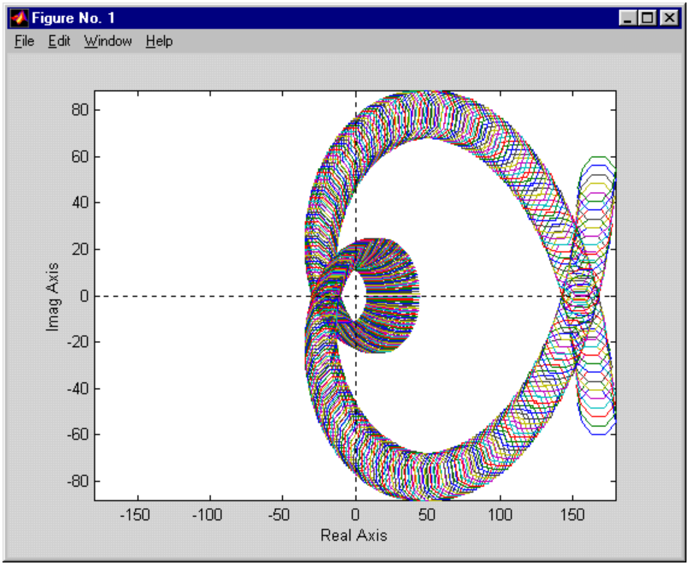


Fig. 18. Zoomed plot of value sets

This yields the plot of Fig. 18. Indeed, zero is excluded from all the octagons ($0 \notin p(c,q)$ for all c on the unit circle) and we conclude that the discrete-time interval polynomial is robustly stable.

Polytopes of polynomials

A more general class of systems is described by uncertain polynomials whose coefficients depend linearly on several parameters, but where each parameter may occur simultaneously in several coefficients. Such an uncertain polynomial may look like

$$p(s,q) = \sum_{i=1}^{n} a_i(q) s^i$$

with each coefficient $a_i(q)$ an affine function of q. That is, for each $i \in \{0, 1, 2, \ldots, n\}$, there exist a column vector a_i and a scalar b_i such that

$$a_i(q) = a_i^T q + b_i$$

Uncertain polynomials with the *affine* uncertainty structure form *polytopes* in the space of polynomials. Similarly to the single parameter case such polynomials may always be expressed as

$$p(s,q) = p_0(s) + q_1 p_1(s) + q_2 p_2(s) + \cdots + q_n p_n(s)$$

This form is preferred in the Polynomial Toolbox. Thus, a polytope of polynomials with n parameters is always described by the $n+1$ polynomials $p_0(s)$, $p_1(s)$, $\cdots$,

$p_n(s)$ along with n parameter bounding intervals $[q_1^-, q_1^+]$, $[q_2^-, q_2^+]$, …, $[q_n^-, q_n^+]$. To keep an invariant degree over the whole polytope it is usually assumed that $\deg p_0(s) \geq \deg p_i(s)$ for all *i.*

One reason why the affine linear uncertainty structure is so important is that it is preserved under feedback interconnection. To see this, consider an uncertain plant

$$P(s,q) = \frac{N(s,q)}{D(s,q)}$$

connected in the standard feedback configuration of Fig. 15 with a compensator

$$C(s) = \frac{N_C(s)}{D_C(s)}$$

A simple calculation leads to the closed loop transfer function

$$P_{CL}(s,q) = \frac{N(s,q)D_C(s)}{N(s,q)N_C(s) + D(s,q)D_C(s)}$$

If the plant has have an affine linear uncertainty structure then the closed-loop transfer function has an affine linear uncertainty structure as well. Indeed, if we write

$$N(s,q) = N_0(s) + \sum_{i=1}^{l} q_i N_i(s)$$

and

$$D(s,q) = D_0(s) + \sum_{i=1}^{l} q_i D_i(s)$$

then the closed-loop characteristic polynomial follows as

$$D_{CL}(s,q) = D_0(s)D_C(s) + N_0(s)N_C(s) + \sum_{i=1}^{l} q_i [D_i(s)D_C(s) + N_i(s)N_C(s)]$$

while the numerator of the closed-loop transfer function is

$$N_{CL}(s,q) = N_0(s)D_C(s) + \sum_{i=1}^{l} q_i N_i(s)D_C(s)$$

Inspection shows that $D_{CL}(s,q)$ and $N_{CL}(s,q)$ have affine linear uncertainty structures. In fact, every transfer function of practical interest has this structure.

The affine linear uncertainty structure is also (roughly speaking) preserved under linear fractional transformation of s and has many other interesting features.

Example 6 *Improvement over rectangular bounds.* For the polytope of polynomials P described by (Barmish, 1996, p. 146)

$$p(s,q) = (q_1 - 2q_2 + 2) + (q_2 + 1)s + (2q_1 - q_2 + 4)s^2 + (2q_2 + 1)s^3 + s^4$$

with $q_1 \in [-0.5, 2]$ and $q_2 \in [-0.3, 0.3]$, we carry out two robust stability analyses.

Part 1: Conservatism of Overbounding. First replace $p(s,q)$ by the overbounding interval polynomial $\overline{p}(s,\overline{q})$ described by

$$\overline{p}(s,\overline{q}) = [0.9,4.6] + [0.7,1.3]s + [2.7,8.3]s^2 + [0.4,1.6]s^3 + s^4$$

Using the Kharitonov polynomials

```
pminus = pol([0.9 0.7 2.7 0.4 1],4);

pplus = pol([4.6 1.3 8.3 1.6 1],4);

[stable,K1,K2,K3,K4] = kharit(pminus,pplus); stable

stable =

     0
```

we conclude that $\overline{p}(s,\overline{q})$ is not robustly stable. It is easy to verify that the third Kharitonov polynomial is unstable:

```
isstable(K3)

ans =

     0
```

Part 2: Value Set Comparison. To begin with the second analysis, we express $p(s,q)$ as

$$p(s,q) = p_0(s) + q_1 p_1(s) + q_2 p_2(s)$$

where

$$\begin{aligned} p_0(s) &= 2+s+4s^2+s^3+s^4 \\ p_1(s) &= 1+2s^2 \\ p_2(s) &= -2+s-s^2+2s^3 \end{aligned}$$

The data are entered as

```
p0 = pol([2 1 4 1 1],4);

p1 = pol([1 0 2],2);

p2 = pol([-2 1 -1 2],3);

Qbounds = [-0.5 2; -0.3 0.3]
```

Next we verify the critical precondition for application of the Zero Exclusion Condition. Indeed, $p_0(s)$ is a stable member of the given interval polynomial family as shown by

```
isstable(p0)

ans =

     1
```

Next we generate 80 polygonal values corresponding to frequencies evenly spaced between $w=0$ and $w=2$:

```
ptopplot(p0,p1,p2,Qbounds,j*(0:0.025:2))
```

Within computational limits, we conclude from Fig. 19 that $0 \notin p(jw,Q)$ for all $w \leq 0$. Hence, by the Zero Exclusion Condition we conclude that P is robustly stable.

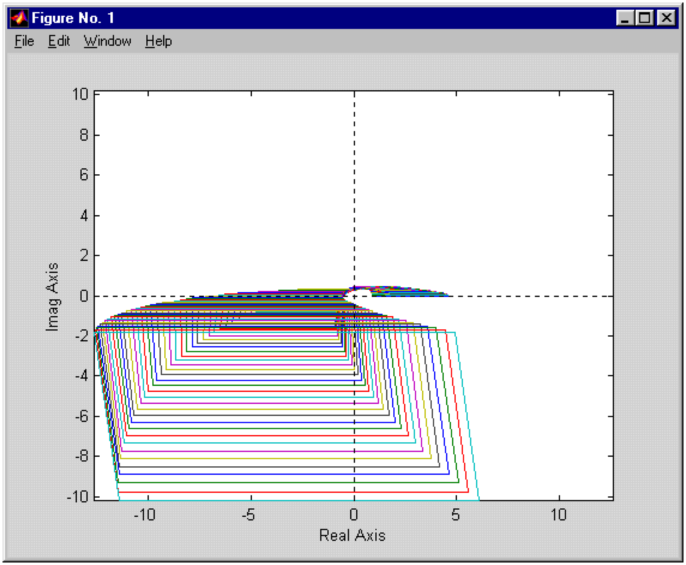


Fig. 19. Extremal polygons

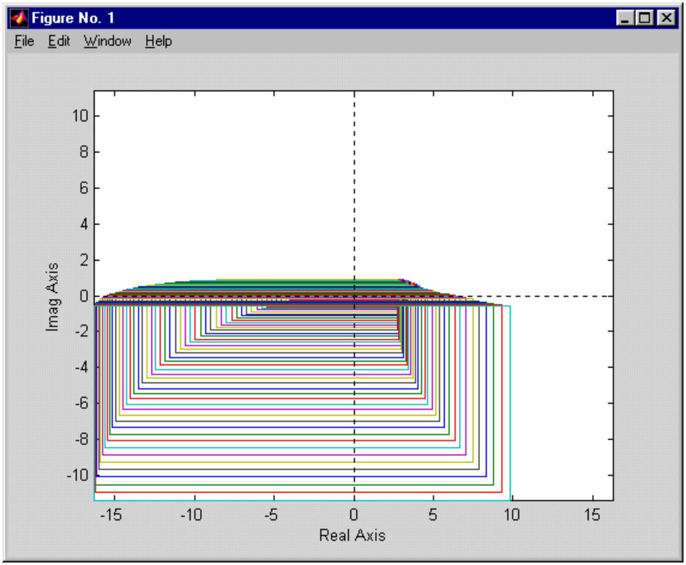


Fig. 20. Kharitonov rectangles

It may also be illuminating to picture the Kharitonov rectangles for the overbounding interval polynomial $\bar{p}(s,\bar{q})$:

```
khplot(pminus,pplus,0:0.025:2)
```

This command results in Fig. 20. It is clear that the Zero Exclusion Condition is violated for the Kharitonov rectangles even though it holds for the polygons of the previous plot.

Summarizing, working with the overbounding interval polynomial is inconclusive while working with polygonal value sets leads us to the unequivocal conclusion that $p(s,q)$ is robustly stable.

Example 7 *Robust stability degree design for a polytopic plant.* Consider the plant transfer function

$$\frac{N(s,q)}{D(s,q)} = \frac{(1+q_1)+(1+q_2)s}{(2+q_1)+(1+q_2)s+(2+2q_2)s^2-2s^3}$$

with two uncertain parameters $q_1 \in [0, 0.2]$ and $q_2 \in [0, 0.2]$. The plant is to be robustly stabilized with robust stability degree $s = 0.9$.

Both the numerator and the denominator of the transfer function are uncertain polynomials with a polytopic (affine) uncertainty structure. Write

$$N(s,q) = N_0(s) + q_1 N_1(s) + q_2 N_2(s) = (1+s) + q_1 + q_2 s$$

and

$$D(s,q) = D_0(s) + q_1 D_1(s) + q_2 D_2(s) = \left(2+s+2s^2-2s^3\right) + q_1\left(1+2s^2\right) + q_2(-3s)$$

and enter the data:

```
D0 = 2+s+2*s^2–2*s^3;

D1 = 1+2*s^2;

D2 = -3*s;

N0 = 1+s;

N1 = 1;

N2 = s;

Qbounds = [ 0 0.2; 0 0.2 ]
```

As the nominal plant

$$\frac{N(s,0)}{D(s,0)} = \frac{N_0(s)}{D_0(s)} = \frac{1+s}{2+s+2s^2-2s^3}$$

is unstable

```
isstable(D0)

ans =

     0
```

we stabilize it by placing the closed-loop poles at -2, -3, -4 and $-2 \pm j$:

```
[Nc,Dc] = pplace(N0,D0,[-2,-2+j,-2-j,-3,-4])
```

```
Nc =

     128.2 + 115.9s + 73.3s^2

Dc =

    -4.1 - 7s - 0.5s^2
```

Note that the nominal positions of the closed-loop poles well satisfy the required stability degree $s = 0.9$.

When the resulting controller

$$\frac{N_C(s)}{D_C(s)} = \frac{128.2 + 115.9s + 73.3s^2}{-4.1 - 7s - 0.5s^2}$$

is connected with the uncertain plant the closed-loop characteristic polynomial becomes uncertain but the polytopic structure is preserved. The characteristic polynomial may be written as

$$p_0(s) + q_1 p_1(s) + q_2 p_2(s)$$

where

```
P0 = D0*Dc+N0*Nc, P1 = D1*Dc+N1*Nc, P2 = D2*Dc+N2*Nc

P0 =

     120 + 226s + 173s^2 + 67s^3 + 13s^4 + s^5
```

```
P1 =

     124.1 + 108.9s + 64.6s^2 – 14s^3 - s^4

P2 =

     140.5s + 136.9s^2 + 74.8s^3
```

Recall that we require achieving a robustly stable system with stability degree 0.9. The polytopic family naturally has a member that is stable with at least the stability degree required (remember the roots of $p_0(s)$) and we can test the motion of polygonal value set $p(c,q)$ by sweeping c along the shifted stability boundary $c = -s + jw = -0.9 + jw$. Starting with values $0 \le w \le 4$ we type

```
ptopplot(10*P0,10*P1,10*P2,Qbounds,-.9+j*(0:.01:4))
```

The plot of Fig. 21 seems to indicate that zero is excluded. To be completely confident, we must zoom the picture to see the critical range $0 \le w \le 1$:

```
ptopplot(P0,P1,P2,Qbounds,-.9+j*(0:.01:1))
```

It is evident from Fig. 22 that $0 \notin p(-0.9 + jw, q)$ for all w and, hence, the Zero Exclusion Condition is verified. We conclude that the desired design specifications are satisfied: The closed-loop system is robustly stable with *robust stability degree 0.9*.

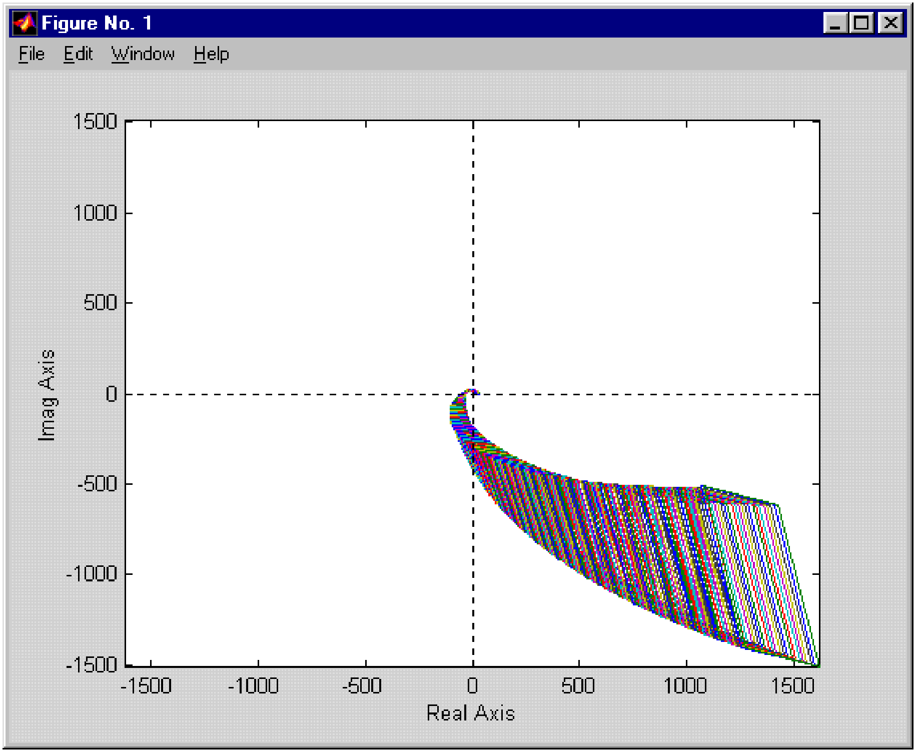


Fig. 21. Value sets

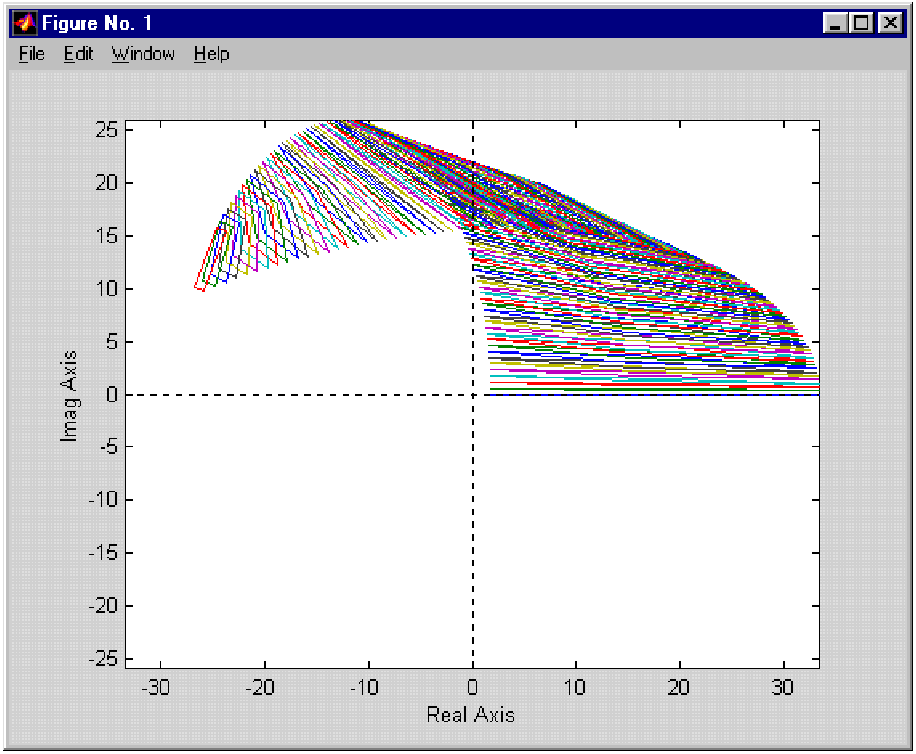


Fig. 22. Zoomed value sets

Numerical methods for polynomial matrices

Introduction

The algorithms that are used in the Polynomial Toolbox use different types of numerical techniques. The techniques may be classified as follows:

- methods based on equating indeterminate coefficients
- polynomial reduction based on elementary row and column operations
- interpolation methods
- state space methods
- other methods

To learn quickly how the first four methods work, look at some easy examples.

Example 1: Scalar linear polynomial equation

Consider solving the scalar polynomial equation of the form

$$a(s)x(s)+b(s)y(s)=c(s)$$

with *a, b* and *c* given polynomials, for the unknown polynomials *x* and *y.* For simplicity we assume that deg *a* = deg *b* = deg *c* = 2. Then whenever the equation is solvable there exists at least one solution *x*, *y* characterized by

$$\deg x \le 1, \quad \deg y \le 1$$

The equation may be solved by equating indeterminate coefficients, polynomial reduction or interpolation as described below. In the Polynomial Toolbox this task is done by the macro `axbyc`.

Example 2: Determinant of a polynomial matrix

For a simple 2×2 polynomial matrix of degree 2

$$P(s) = P_0 + P_1 s + P_2 s^2$$

consider the computation of its determinant

$$p(s) = \det P(s) = p_0 + p_1 s + p_2 s^2 + p_3 s^3 + p_4 s^4$$

Note that $p_4 \ne 0$ if and only if P_2 is nonsingular.

The determinant may be found by polynomial reduction, by interpolation or by state space methods as described below. In the Polynomial Toolbox the computation of the determinant is handled by the macro `det`.

Equating indeterminate coefficients

Let us see how the scalar linear polynomial equation may be solved by equating indeterminate coefficients.

Example 1: Scalar linear polynomial equation

We begin by writing

$$a(s) = a_0 + a_1 s + a_2 s^2$$
$$b(s) = b_0 + b_1 s + b_2 s^2$$
$$c(s) = c_0 + c_1 s + c_2 s^2$$

where the coefficients are all known. Likewise, we write

$$x(s) = x_0 + x_1 s$$
$$y(s) = y_0 + y_1 s$$

with the unknown coefficients x_0, x_1, y_0 and y_1 to be found.

Step 1. By expanding the expression *xa* + *yb* and equating coefficients of like powers in the indeterminate variable *s* we obtain a set of linear equations of the form

$$\begin{bmatrix} x_0 & y_0 & x_1 & y_1 \end{bmatrix} \underbrace{\begin{bmatrix} a_0 & a_1 & a_2 & 0 \\ b_0 & b_1 & b_2 & 0 \\ 0 & a_0 & a_1 & a_2 \\ 0 & b_0 & b_1 & b_2 \end{bmatrix}}_{S} = \begin{bmatrix} c_0 & c_1 & c_2 & 0 \end{bmatrix}$$

The matrix *S* has a highly structured form and is called the *Sylvester resultant matrix* corresponding to the polynomials *a* and *b.*

Step 2. Solve the constant matrix equation to obtain the unknown coefficients of the polynomials *x* and *y* and, hence, the polynomials themselves. If the set of linear equations is not solvable then the polynomial matrix has no solution.

Discussion. The procedure is quite natural. It applies whenever

- the degree of the expected solution is known, and
- the constant matrix system is linear, of a reasonable size and easily constructed. Then it may efficiently be solved by any standard numerically stable linear system solver.

The knowledge of the resulting degree is crucial. It guarantees that the correspondence between the polynomial equation and the linear system is one-to-one.

If the degree is not known then it is necessary to proceed heuristically. Just try a large enough degree and check whether or not the resulting linear system is solvable. If it is then the desired polynomial solution has been found. If it is not then nothing can be concluded. There may or may not exist polynomial solutions of higher degree.

In the Polynomial Toolbox, the method of equating indefinite coefficients is employed in equation solvers such as `axb`, `axbyc`, `axybc`, `axxa2b`.

For further reading see the Bibliography.

Polynomial reduction

To solve the scalar linear polynomial equation by polynomial reduction we proceed as follows.

Example 1 : Scalar linear polynomial equation

Step 1. Use the polynomials *a* and *b* to form the polynomial matrix

$$\begin{bmatrix} a(s) & 1 & 0 \\ b(s) & 0 & 1 \end{bmatrix}$$

Use elementary row operations to reduce the matrix until the entry in the lower left corner equals 0. After completion the matrix has the form

$$\begin{bmatrix} g(s) & p(s) & r(s) \\ 0 & q(s) & t(s) \end{bmatrix}$$

The polynomial *g* is the greatest common divisor of *a* and *b* while *p, q*, *r* and *t* are coprime polynomials that satisfy

$$\begin{aligned} p(s)a(s) + q(s)b(s) &= g(s) \\ r(s)a(s) + t(s)b(s) &= 0 \end{aligned}$$

Step 2. Extract *g* from the right hand side polynomial *c* to obtain a polynomial $\bar{c}$ so that

$$c(s) = \bar{c}(s)g(s)$$

If this is not possible then stop because the polynomial equation is unsolvable.

Step 3. Take

$$\begin{aligned} x(s) &= \bar{c}(s)p(s) \\ y(s) &= \bar{c}(s)q(s) \end{aligned}$$

to be the desired solution. Moreover, *all* solutions to the polynomial equation may be expressed as

$$\begin{aligned} x(s) &= \bar{c}(s)p(s) + r(s)u(s) \\ y(s) &= \bar{c}(s)q(s) + t(s)u(s) \end{aligned}$$

with u an arbitrary free polynomial parameter.

Example 2: Computation of the determinant

To compute the determinant of the polynomial matrix P by polynomial reduction we proceed in the following way.

Step 1. Using elementary row operations, transform the given matrix P into a lower triangular matrix of the form

$$\begin{bmatrix} t_{11}(s) & 0 \\ t_{21}(s) & t_{22}(s) \end{bmatrix}$$

Step 2. Because elementary operations preserve the determinant the desired result may immediately be calculated as

$$\det P(s) = t_{11}(s)t_{22}(s)$$

Discussion. The polynomial reduction method is a traditional method of real “polynomial flavor.” A typical feature of this method is that no attention is paid to the degree of the polynomials, which may grow alarmingly during the computation.

Unfortunately, the method is not numerically stable and, if the given data are “bad” then the performance of the method heavily depends on effective zeroing.

Finally, the method turns out to be rather slow when programmed in MATLAB.

In the Polynomial Toolbox polynomial reductions are employed in a few functions such as `hermite`, `pdg`, and `smith`. The method is avoided whenever possible.

For further reading see the Bibliography.

Interpolation

Interpolation is a very effective technique.

Example 1: Linear polynomial equation

The “interpolation way” to determine the unknown polynomials x and y from the equation $ax + by = c$ consists of the following steps.

Step 1. For the problem at hand, choose four distinct complex interpolation points

$$s_1, s_2, s_3, s_4$$

and substitute them into the polynomials *a, b* and *c* to obtain the scalar constants

$$a(s_i), a(s_i), a(s_i), \quad i = 1, 2, 3, 4$$

Step 2. Form the linear equation system

$$\begin{bmatrix} x_0 & y_0 & x_1 & y_1 \end{bmatrix} \begin{bmatrix} a(s_1) & a(s_2) & a(s_3) & a(s_4) \\ b(s_1) & b(s_2) & b(s_3) & b(s_4) \\ s_1 a(s_1) & s_2 a(s_2) & s_3 a(s_3) & s_4 a(s_4) \\ s_1 b(s_1) & s_2 b(s_2) & s_3 b(s_3) & s_4 b(s_4) \end{bmatrix} = \begin{bmatrix} p(s_1) & p(s_2) & p(s_3) & p(s_4) \end{bmatrix}$$

Step 3. Solve the equation system for the desired coefficients of the polynomials x and y.

Example 2: Computation of the determinant

Quite similarly the determinant may be computed by interpolation.

Step 1. For the problem at hand, choose five distinct interpolation points

$$s_i, \quad i = 1, 2, 3, 4, 5$$

and substitute them into the given matrix P to obtain the five constant matrices

$$P(s_i), \quad i = 1, 2, 3, 4, 5$$

Step 2. Calculate the determinants

$$p(s_i) = \det P(s_i), \quad i = 1, 2, 3, 4, 5$$

Step 3. Recover the desired coefficients of

$$p(s) = \det P(s) = p_0 + p_1 s + p_2 s^2 + p_3 s^3 + p_4 s^4$$

by solving the linear equation system

$$\begin{bmatrix} p_0 & p_1 & p_2 & p_3 & p_4 \end{bmatrix} \underbrace{\begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ s_1 & s_2 & s_3 & s_4 & s_5 \\ s_1^2 & s_2^2 & s_3^2 & s_4^2 & s_5^2 \\ s_1^3 & s_2^3 & s_3^3 & s_4^3 & s_5^3 \\ s_1^4 & s_2^4 & s_3^4 & s_4^4 & s_5^4 \end{bmatrix}}_{V} = \begin{bmatrix} p(s_1) & p(s_2) & p(s_3) & p(s_4) & p(s_5) \end{bmatrix}$$

The matrix V is called a *Vandermonde matrix*.

Discussion. Interpolation for polynomial matrix computations is quite modern and efficient. Obviously, the larger the part of computation is that is done with constant matrices the more efficient the method is.

Like other methods, it requires the resulting degrees to be correctly determined *a priori*. If no justified guess is available then the method becomes quite heuristic. If an incorrect degree is assumed then neither solvability of the linear equation system guarantees the existence of a solution to the polynomial solution nor unsolvability of the linear system implies the non-existence of a solution to the polynomial problem.

The Vandermonde matrix appearing in the linear equation system often is ill conditioned. In many cases this does not matter as a special “Vandermonde solver” may be employed. If this is not possible then the condition number limits the problem size.

In the Polynomial Toolbox the interpolation method is encountered in routines such as `det` and all linear polynomial matrix equation solvers..

For further reading see the Bibliography.

State space methods

Example 2: Computation of the determinant

To compute the determinant an indirect method may be used based on “state space” notions from linear system theory.

Step 1. From the matrix coefficients of

$$P(s) = P_0 + P_1 s + P_2 s^2$$

form the pair of generalized block companion matrices

$$E = \begin{bmatrix} I_{2\times 2} & 0 \\ 0 & P_2 \end{bmatrix}, \quad A = \begin{bmatrix} 0_{2\times 2} & I_{2\times 2} \\ -P_0 & -P_1 \end{bmatrix}$$

This pair defines a descriptor system $E\dot{x} = Ax$.

Step 2. Compute the generalized eigenvalues corresponding to the matrices *E* and *A,* that is, the roots of $\det(lE - A)$. Remove the infinite roots and denote the remaining finite roots as

$$s_i, \quad i = 1, 2, 3, 4$$

These roots equal the finite roots of $p(s)$.

Step 3. Recover $p(s)$ = det $P(s)$ from its finite roots using the formula

$$p(s) = c(s - s_1)(s - s_2)(s - s_3)(s - s_4)$$

Discussion. There are state space counterparts to many polynomial problems. As quite advanced numerical procedures have been developed for state-space problems the detour via systems theory is sometimes rewarding.

In the Polynomial Toolbox, state-space-like methods may be found in `root` and `spf`.

For further reading see the Bibliography.

Demos and applications

Introduction

In this chapter we present several demos and applications that illustrate the wide variety of problems that may be solved with the Polynomial Toolbox. The demos successively deal with

- achievable feedback performance
- computing the covariance function of an ARMA process
- control of a batch process
- the polynomial solution of the SISO mixed sensitivity problem, and
- applications to behavioral system theory

Achievable feedback performance

Introduction Many of the properties of the MIMO feedback loop of Fig. 23 are characterized by its sensitivity matrix

$$S = (I + PC)^{-1}$$

and complementary sensitivity matrix

$$T = (I + PC)^{-1} PC$$

These properties include the sensitivity of the feedback loop to disturbances and measurement noise, its response to reference inputs, and its stability and performance robustness (Kwakernaak, 1995).

H_∞ optimization is a powerful tool for shaping the sensitivity functions so that all the design requirements are satisfied. In particular the mixed sensitivity problem is a useful design paradigm. The Polynomial Toolbox contains the routine `mixeds` for solving SISO mixed sensitivity problems.

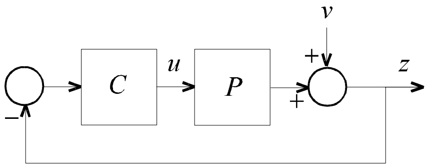


Fig. 23. LTI feedback system

Especially for complex design problems it is highly recommended to devote some time to *exploratory analysis* before attempting the actual design. This exploratory analysis serves to reveal the inherent possibilities and limitations of the control system. Part of this exploratory analysis always is the computation of the poles and zeros of the plant. Any right-half plane zeros and poles that are present impose essential constraints on the closed-loop bandwidth that may be achieved or is necessary (again see Kwakernaak, 1995, for a review).

For MIMO systems the open-loop pole and zero locations do not fully reveal the design limitations. For instance, in the SISO case the presence of a nearly cancelling pole-zero pair in the right-half plane predicts poor performance, with high peaks in the sensitivity functions. The closer the pole and zero are, the higher the peak. In the case of MIMO systems the pole of such a pair may occur in a different "channel" than the zero so that the pole and zero do not interact and their adverse effects are not amplified such as in the SISO case.

To reveal the a priori design limitations more fully it may be useful to compute the minimal peak values of the ∞-norms of the sensitivity functions S and T. This is the purpose of this demo. Polynomial techniques lend themselves very well for the computation of these bounds and explicitly reveal their relation to the open-loop zero and pole locations. The peak values themselves are lower bounds for the peak values that are obtained for more realistic designs matched to the design specifications.

SISO minimal peak values for the sensitivities

For the SISO case the minimal peak values are derived in Kwakernaak (1985). We assume that the plant has the transfer function

$$P(s)=\frac{N(s)}{D(s)}=\frac{N_+(s)N_-(s)}{D_+(s)D_-(s)}$$

The polynomial N is the plant numerator polynomial and the polynomial D is the plant denominator polynomial. D_+ and N_+ are polynomials whose roots have strictly positive part and hence lie in the open right-half complex plane. The roots of the polynomials D_- and N_- all have nonpositive real part.

The minimal peak value of the ∞-norm of the sensitivity function S in the SISO case equals the minimal peak value of the magnitude of the sensitivity function $|S(jw)|$, $w \in \mathbf{R}$. This minimal norm may be found by solving a special H_∞ optimization problem, namely that of minimizing $\|S\|_\infty$. The solution of this minimum sensitivity problem follows by solving the polynomial equation

$$\frac{1}{g} D_+^* X_+^* = D_+ X_+ + N_+ Y_+$$

for the scalar g and the polynomials X_+ and Y_+. If D_+ has degree d and N_+ has degree n then X_+ has degree $n-1$ and Y_+ has degree $d-1$. As we shall see this equation is equivalent to a generalized eigenvalue problem. The compensator that solves the minimum sensitivity problem has the transfer function

$$C_{\text{opt}}(s) = \frac{D_-(s) Y_+(s)}{N_-(s) X_+(s)}$$

If the plant is strictly proper then C_{opt} is nonproper. The optimal sensitivity function is given by

$$S_{\text{opt}}(s) = g \frac{D_+(s) X_+(s)}{D_+^*(s) X_+^*(s)}$$

S_{opt} has the so-called *equalizing property,* that is, $|S_{\text{opt}}(jw)|$ is constant for all real w and equals $|g|$. The minimal peak value hence is $|g|$.

The properties of the minimum peak value may be summarized as follows:

- If the plant has no right-half plane zeros then $\|S_{\text{opt}}\|_\infty = 0$, which is achieved by an infinite-gain compensator.
- If the plant has no right-half plane poles but at least one right-half plane zero then $\|S_{\text{opt}}\|_\infty = 1$, which is obtained by taking $C(s) = 0$.
- If the plant has at least one right-half plane zero and at least one right-half plane pole then $\|S_{\text{opt}}\|_\infty > 1$.
- If the plant has a coinciding right-half plane pole-zero pair then $\|S_{\text{opt}}\|_\infty = \infty$.

For the minimal peak value of the complementary sensitivity function T similar results hold. The critical equation that needs to be solved now is

$$\frac{1}{g} N_+^* Y_+^* = D_+ X_+ + N_+ Y_+$$

and the optimal complementary function is

$$S_{\text{opt}}(s) = g \frac{N_+(s) Y_+(s)}{N_+^*(s) Y_+^*(s)}$$

This is the summary of the results for the minimization of $\|T\|_\infty$:

- If the plant has no right-half plane poles then $\|T_{\mathrm{opt}}\|_\infty = 0$, which is achieved by an infinite-gain compensator.
- If the plant has no right-half plane zeros but at least one right-half plane pole then $\|T_{\mathrm{opt}}\|_\infty = 1$, which is obtained by taking $C(s) = 0$.
- If the plant has at least one right-half plane zero and at least one right-half plane pole then $\|T_{\mathrm{opt}}\|_\infty = \|S_{\mathrm{opt}}\|_\infty > 1$.
- If the plant has a coinciding right-half plane pole-zero pair then $\|T_{\mathrm{opt}}\|_\infty = \infty$.

Note that minimal sensitivity and minimal complementary sensitivity are not simultaneously achieved by the same compensator.

SISO computation of the minimum peak value

The only situation where some serious computation needs to be done once the open-loop poles and zeros are available arises when the plant has both right half plane zeros and poles. We then need to solve the polynomial equation

$$\frac{1}{g} D_+^* X_+^* = D_+ X_+ + N_+ Y_+ \qquad (1)$$

The solution of this problem provides the minimal peak value of both the sensitivity and the complementary sensitivity function. If D_+ has degree d and N_+ has degree n then X_+ has degree $n-1$ and Y_+ has degree $d-1$. Write X_+ and Y_+ in terms of their coefficients as

$$X_+(s) = X_0 + X_1 s + \cdots + X_{n-1} s^{n-1}, \quad Y_+(s) = Y_0 + Y_1 s + \cdots + Y_{d-1} s^{d-1}$$

and introduce the column vectors

$$x = \begin{bmatrix} X_0 \\ X_1 \\ \cdots \\ X_{n-1} \end{bmatrix}, \quad y = \begin{bmatrix} Y_0 \\ Y_1 \\ \cdots \\ Y_{d-1} \end{bmatrix}$$

Then it may be verified that the polynomial equation (1) we need to solve is equivalent to the matrix equation

$$\frac{1}{g} S_{n+d-1}(D_+^*) J_n x = S_{n+d-1}(D_+) x + S_{n+d-1}(N_+) y \qquad (2)$$

$S_m(A)$ denotes the (column) Sylvester resultant matrix of order m of the polynomial A, and J_n is the matrix

$$J_n = \mathrm{diag}(1, -1, 1, -1, \cdots, (-1)^{n-1})$$

To show that (2) amounts to a generalized eigenvalue problem we rearrange it in the form

$$\frac{1}{g} \begin{bmatrix} S_{n+d-1}(D_+^*) J_n & 0 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} S_{n+d-1}(D_+) & S_{n+d-1}(N_+) \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

This is equivalent to the equation

$$Az = l Bz \qquad (3)$$

where

$$A = \begin{bmatrix} S_{n+d-1}(D_+) & S_{n+d-1}(N_+) \end{bmatrix}, \quad B = \begin{bmatrix} S_{n+d-1}(D_+^*)J_n & 0 \end{bmatrix}$$

$$z = \begin{bmatrix} x \\ y \end{bmatrix}, \quad l = \frac{1}{g}$$

We need the solution of (3) that corresponds to the real eigenvalue l with the smallest size. The minimal peak sensitivity equals the inverse of the magnitude of the smallest eigenvalue.

The macro minsens

We develop a new Polynomial Toolbox function `minsens` that computes the minimal peak value of the sensitivity functions. Its input arguments are the numerator polynomial N and the denominator polynomial D of the SISO plant.

As we develop the macro we test it for the plant with transfer function

$$P(s) = \frac{s^2 + 2s - 3}{s^2 - 4/9}$$

We input the data accordingly as

```
N = s^2+2*s-3;
```

```
D = s^2-4/9;
```

The first few lines of the m-file are

```
% minsens
% The function

%    p = minsens(N,D)

% computes the minimum peak value of the sensitivity and

% complementary sensitivity functions for the SISO plant

% with transfer function P = N/D that may be achieved by

% feedback

function p = minsens(N,D)
```

This provides the help text, and declares the function, its input arguments and its output arguments.

Normally a sequence of tests needs to follow this preamble to check whether *N* and *D* are really scalar polynomials but we dispense with this for the purpose of this demo.

Given the numerator and denominator polynomials we now compute their roots and use these to define the polynomials N_+ and D_+ :

```
% Compute the polynomials Nplus and Dplus whose roots are

% the roots of N and D, respectively, with positive real
parts
```

```
rootsN = roots(N); rootsNplus =
rootsN(find(real(rootsN)>0));

Nplus = mat2pol(poly(rootsNplus));

rootsD = roots(D); rootsDplus =
rootsD(find(real(rootsD)>0));

Dplus = mat2pol(poly(rootsDplus));

keyboard
```

The MATLAB command `poly` is used to construct the polynomials `Nplus` and `Dplus` from their roots after which they are converted to Polynomial Toolbox format with the Toolbox command `mat2pol`. While developing the macro we end it with the `keyboard` command so that the results at that point may be inspected. In the present case typing the command

```
minsens(N,D)
```

results in the output

```
K»
```

Editing this to

```
K» Nplus, Dplus
```

and ending the line with a return results in the output

```
Nplus =

    -1 + s

Dplus =

    -0.67 + s
```

We now include two tests to see whether the peak value is either 0 or 1.

```
% Check whether p = 0 or p = 1

if isempty(Nplus)

   p = 0; return

elseif isempty(Dplus)

   p = 1; return

end
```

For the example that we are pursuing both tests fail so we are in the situation where the generalized eigenvalue problem needs to be solved. We first set it up.

```
% Solve the generalized eigenvalue problem

A = [ sylv(Dplus,'col',n-1) sylv(Nplus,'col',d-1) ];

J = 1;
```

```
for i = 2:n
   J(i,i) = -1*J(i-1,i-1);
end
B = [ sylv(Dplus','col',n-1)*J zeros(n+d,d) ];
```

Only one more line is needed to complete the macro:

```
p = 1/min(abs(eig(A,B)));
```

Calling

```
minsens(N,D)
```

results for the example at hand in

```
ans =
  5.0000
```

We test a few more examples

- No right-half plane zeros:

```
minsens(s+1,s-1)
ans =
     0
```

- One right-half plane zero but no right-half plane poles:

```
minsens(s-1,s+1)

ans =

     1
```

- Coinciding right-half plane pole-zero pair:

```
minsens(s-1,s-1)

Warning: Divide by zero.

ans =

   Inf
```

MIMO minimal peak values for the sensitivities

For the MIMO case no simple results to compute the minimal peak sensitivity such as for the SISO case are available. The only option seems to be to compute a full solution of the minimum sensitivity problem. The Polynomial Toolbox provides the routine `dssrch` for this purpose. It solves the standard H_∞ optimization problem, is numerically well-conditioned, fast, computes optimal solutions accurately and reliably, and can handle nonproper generalized plants as well as problems where the optimal compensator is nonproper. The routine requires the generalized plant of the standard problem to be in descriptor form. The Toolbox supplies the necessary conversion routines from polynomial matrix fraction form to descriptor form and vice-versa.

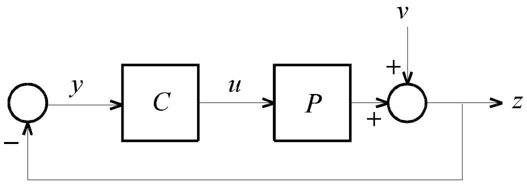


Fig. 24. Minimum sensitivity problem

To bring the minimum sensitivity problem into standard form we consider the block diagram of Fig. 24. When the loop is opened the signals are related as

$$\begin{bmatrix} z \\ y \end{bmatrix} = \begin{bmatrix} I & P \\ -I & P \end{bmatrix} \begin{bmatrix} v \\ u \end{bmatrix}$$

This defines the generalized plant of the standard problem as

$$G = \begin{bmatrix} I & P \\ -I & -P \end{bmatrix}$$

If the plant P has the left coprime representation $P = D^{-1}N$ then the generalized plant has the left coprime representation

$$G = \begin{bmatrix} D & 0 \\ I & I \end{bmatrix}^{-1} \begin{bmatrix} D & N \\ 0 & 0 \end{bmatrix}$$

After converting this left coprime fraction to descriptor form the routine `dssrch` may be called to solve the H_∞ optimization problem.

MIMO example

We generate a random 2×2 plant $P = D^{-1}N$ by the commands

```
D = prand([1;2],2,2,'int')

D =

    -8 + s      1 - 6s

     6 + 6s     2 + s - s^2

N = prand([1;2],2,2,'int')

N =

    -3 + 11s    -1 + s

     5          -4 + s - 7s^2
```

The zeros and poles of the plant are

```
Zeros = roots(N)

Zeros =

   0.0379 + 0.8051i

   0.0379 - 0.8051i
   0.3399
```

```
Poles = roots(D)
```

```
Poles =

   45.5167

   -1.0000

    0.4833
```

Inspection shows that the plant has both right-half plane poles and zeros so by analogy to the SISO problem we expect a minimum peak sensitivity of at least 1.

We first convert the generalized plant into descriptor form:

```
Dg = [D zeros(2,2); eye(2,2) eye(2,2)];

Ng = [D N; zeros(2,2) zeros(2,2)];

[A,B,C,D,E] = lmf2dss(Ng,Dg)
```

```
A =

     8   -47     0

    -6    37     1

    -6    38     0
```

```
B =

         0         0   85.0000  336.0000

         0         0  -66.0000 -264.0000

         0         0  -61.0000 -276.0000

C =

     1    -6     0

     0    -1     0

    -1     6     0

     0     1     0

D =

    1.0000         0   11.0000   43.0000

         0    1.0000         0    7.0000

   -1.0000         0  -11.0000  -43.0000

         0   -1.0000         0   -7.0000

E =

     1     0     0
     0     1     0
     0     0     1
```

The solution of the H_∞ problem is initiated by typing

```
 [Ak,Bk,Ck,Dk,Ek,gopt,clpoles] =
dssrch(A,B,C,D,E,2,2,0.5,5)
```

The resulting output is

```
Ak =

    0.5209    1.0787

   -0.9706   -0.1370

Bk =

   -0.9241   -9.2055

    1.7746   -0.6237

Ck =

    0.0410   -0.0035

   -0.1003    0.0981

Dk =
   -0.3750    0.0976

    0.0547   -0.0142

Ek =

     1     0

     0     1

gopt =

    1.1595

clpoles =

 -45.5167 + 0.0000i

  -0.1755 - 0.9418i

  -0.1755 + 0.9418i

  -0.4833 - 0.0000i

  -1.0000
```

We observe that the minimal norm is 1.1595. The fact that this number is not all that much greater than 1 indicates that a design without exaggerated peaking of the sensitivity functions is possible as long as the design specifications — in particular the desired bandwidth — are compatible with the limitations imposed by the right-half plane zeros and poles of the plant.

Computing the covariance function of an ARMA process

Introduction

The problem of computing the covariance function for a given multivariable ARMA process is often encountered in estimation, filtering, stochastic control and communications.

Consider the ARMA process

$$A(z)y(t) = B(z)e(t), \qquad t \in \mathbf{Z}$$

where z is the shift operator defined by $zy(t) = y(t+1)$. A and B are square polynomial matrices in z with possibly complex-valued coefficients. The random sequence e is white noise so that

$$Ee(t)e^H(t) = \begin{cases} 0 & \text{for } t \neq s \\ I & \text{for } t = s \end{cases}$$

The superscript H indicates the complex conjugate transpose. A is assumed to be monic (that is, its leading coefficient matrix is nonsingular) with all its roots strictly inside the unit circle. Under these assumptions the ARMA process y is well-defined and asymptotically stationary.

The covariance matrix function that is to be found is defined by

$$r(t) = \lim_{t \to \infty} Ey(t+t)y^H(t)$$

Algorithm

The covariance matrix function may be computed by inverse *z*-transformation of the spectral density matrix

$$\Phi(z) = A^{-1}(z)C(z)C^H(1/z)\left(A^{-1}(1/z)\right)^H$$

The computation of the covariance function (Söderström, Jezek and Kucera, 1997) follows by partial fraction expansion of the spectral density in the form

$$\Phi(z) = A^{-1}(z)X(z) + X^H(1/z)\left(A^{-1}(1/z)\right)^H$$

This partial fraction expansion is equivalent to solving the symmetric two-sided polynomial matrix equation

$$C(z)C^H(1/z) = X(z)A^H(1/z) + A(z)X^H(1/z)$$

for the polynomial matrix X. Once X is available we may expand

$$A^{-1}(z)X(z) = \sum_{t=0}^{\infty} \hat{r}(t)z^{-t}$$

by long polynomial division. Inspection of the right-hand side shows that

$$r(t) = \begin{cases} \hat{r}(t) & \text{for } t > 0 \\ \hat{r}(0) + \hat{r}^H(0) & \text{for } t = 0 \\ \hat{r}^H(-t) & \text{for } t < 0 \end{cases}$$

Example 1: A scalar process

To illustrate the procedure first consider a scalar ARMA process *y* given by

$$A(z) = 1 - 2.4z + 143z^2$$
$$C(z) = 1$$

We develop a Polynomial Toolbox function called `covf`, which takes the polynomial matrices *A* and *C* as input arguments and has the desired covariance function *r* as output. Because a macro with the same name exists in the System Identification Toolbox we need to *overload* this function. Practically this means that the macro is placed in the `pol` subdirectory of the main Polynomial Toolbox directory. When MATLAB detects that `covf` is called with one or more polynomial objects as input argument then it uses the version of `covf` that is located in this subdirectory.

The macro covf

The first lines of the macro are

```
% covf

%

% This function computes the covariance function

% of the discrete-time ARMA process y defined by

%

% A(z)y(t) = C(z)e(t)

%
```

```
% with e standard white noise.


function r = covf(A,C,n)
```

The third input argument *n* is the number of time shifts over which the covariance function is required.

Normally at this point each Polynomial Toolbox function performs a number of correctness checks on the input arguments. We dispense with these for the purpose of this demo.

To solve the symmetric polynomial equation

$$X(z)A^H(1/z)+A(z)X^H(1/z)=C(z)C^H(1/z)$$

we use the Toolbox function `xaaxb`. Only one line is needed:

```
% Solve the two-sided polynomial matrix equation

X = xaaxb(A,C*C');
```

For the example at hand the intermediate solution at this point is

```
X =

   -20 + 8.3z + 28z^2
```

From the given polynomial A and the computed polynomial X the desired covariance function is recovered by "long division" of $X^{-1}A$. For this purpose the macro `longldiv` is available. Given the square polynomial matrix D and the polynomial matrix N this function finds the first $n + d + 1$ terms of the Laurent series expansion

$$D^{-1}(z)N(z) = Q_n z^n + Q_{n-1} z^{n-1} + \cdots + Q_1 z + R_0 + R_1 z^{-1} + R_2 z^{-2} + \cdots + R_d z^{-1} + \cdots$$

If the fraction $D^{-1}N$ is proper then the macro returns the first $d+1$ terms (with d an input parameter) of the expansion

$$D^{-1}(z)N(z) = R_0 + R_1 z^{-1} + R_2 z^{-2} + \cdots + R_d z^{-1} + \cdots$$

This is exactly what we need. Thus, the appropriate command is

```
% Apply long division to X\A

[Q,R] = longldiv(X,A,n);
```

R is returned as a polynomial matrix in the variable z^{-1} of degree n. For convenience we also return the desired covariance function as a polynomial in this variable:

```
% Construct the covariance function

r = R;

r{0} = r{0}+r{0}';
```

The macro is now complete and we may apply it to the example:

```
r = covf(1-2.4*z+1.43*z^2,1,40);
```

The resulting covariance function r may be plotted using standard MATLAB commands:

```
plot(0:40,r{:},'o')

title('Covariance function - Example 1')

xlabel('tau'), ylabel('r'), grid on
```

Fig. 25 shows the plot of the covariance function.

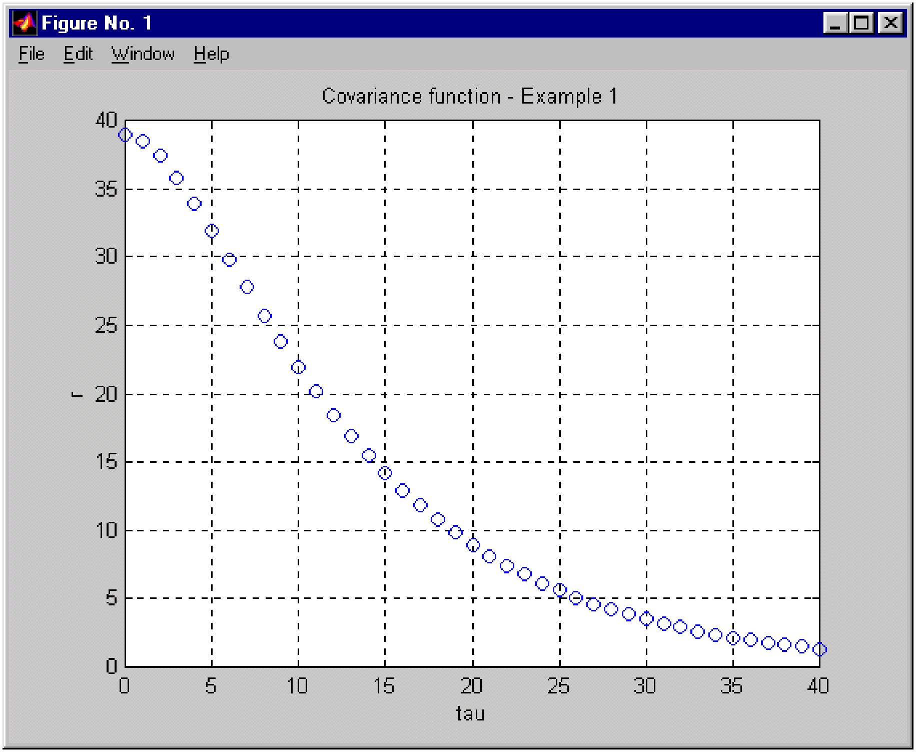


Fig. 25. Covariance function of the ARMA process of Example 1

Example 2: Two-variable ARMA process

As another example consider the computation of the covariance matrix function of the two-variable ARMA process *y* defined by

$$A(z) = \begin{bmatrix} 1-2z & 0 \\ 6 & 1-2.5z \end{bmatrix}, \quad C(z) = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

As our newly created function is ready for multivariable processes we may use it as it is after defining the data:

```
A = [1-2*z 0; 6 1-2.5*z]; C = eye(2,2);

r = covf(A,C,10);
```

The output *r* now is a 2×2 polynomial matrix of degree 10. The coefficients *r*{*i*}, *i* = 1, 2, $\cdots$, 10, constitute the desired covariance function. They may be plotted in a single frame by the following sequence of standard MATLAB commands that we include at the end of the macro `covf`:

```
% Plot the covariance function

figure; clf

k = length(C);

for i = 1:k

   for j = 1:k

      subplot(k,k,(i-1)*k+j)
```

```
        plot(0:n,r{:}(i,j),'o')
        grid on; xlabel('tau')
      end
    end
```

The resulting plot is shown in Fig. 26. The subplot in position (*i, j*) shows the scalar covariance function $r_{ij}(\tau) = \lim_{t\to\infty} \operatorname{cov}[y_i(t+\tau), y_j(t)]$.

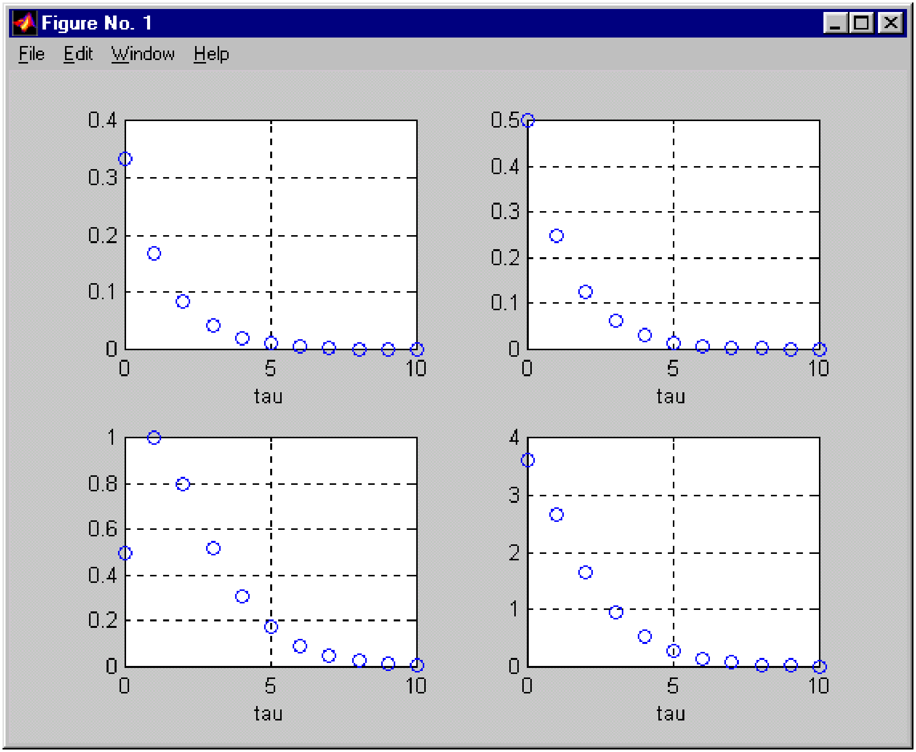


Fig. 26. Covariance function of the ARMA process of Example 2

Control of a batch process

Introduction

Many photographic films and papers are manufactured in a batch-like mode. In this mode batches of sensitized material are made up and then coated onto a base. To guarantee that the photographic properties are kept within limits, strips of product are regularly sent to testing for assessment. If the product is drifting off aim then it is possible to add dye or change the laydown to move the product back on target.

However, there frequently are more outputs than “control knobs” to use for adjustment, and the inputs frequently affect many outputs simultaneously. Testing delay, more outputs than inputs, the desire for a first-order return rather than a step return to target for some products, and stochastic disturbances make this an interesting control problem.

Below is the transfer function plus disturbance model for a process like the one described above.

$$\begin{bmatrix} y_1(t) \\ y_2(t) \\ y_3(t) \end{bmatrix} = \begin{bmatrix} -77.0z^{-3} & 0.33z^{-3} \\ 0 & 0.03z^{-3} \\ 0 & 0.1z^{-3} \end{bmatrix} \begin{bmatrix} u_1(t) \\ u_2(t) \end{bmatrix} + \begin{bmatrix} \dfrac{1-0.6z^{-1}}{1-z^{-1}} & 0 & 0 \\ 0 & \dfrac{1}{(1-0.5z^{-1})(1-z^{-1})} & 0 \\ 0 & 0 & \dfrac{1-0.55z^{-1}}{1-z^{-1}} \end{bmatrix} \begin{bmatrix} v_1(t) \\ v_2(t) \\ v_3(t) \end{bmatrix}$$

Here z^{-1} is the delay operator and the $v_i(t)$ are disturbance signals.

The three outputs y_1, y_2 and y_3 successively are the deviations from aim of photographic speed, contrast, and upper density point. The two inputs u_1 and u_2 are dye and laydown changes from preset starting values, and the delay of three accounts for the zero order hold and testing delay.

The three disturbance models are first order integrated moving average, first order integrated autoregressive, and first order integrated moving average, respectively.

Design targets

We look for a feedback controller that

- reduces the effect of random disturbances, and
- eliminates the effect of long term drifts in the disturbances

Theory

A useful control design approach is the Internal Model Control (IMC) approach discussed in MacGregor and Harris (1987). Fig. 27 shows the arrangement.

The control law *H* for this problem follows by the minimization of a mean square error criterion of the form

$$E\left(e^T(t)Q_1e(t)+\nabla u^T(t)Q_2\nabla u(t)\right)$$

E denotes the expectation, $e(t)$ is the deviation of the output $y(t)$ from its set point, and $\nabla u(t)=u(t)-u(t-1)$ is the incremental control action. Q_1 and Q_2 are weighting matrices.

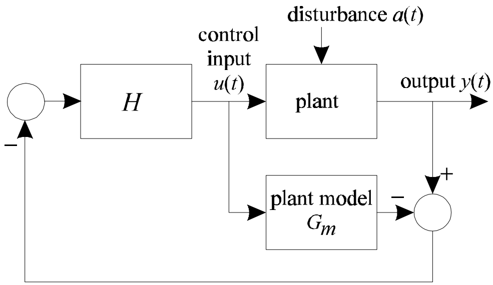


Fig. 27. Internal Model Control

Process and disturbance models

The control is computed by a polynomial algorithm. To this end, the plant-disturbance model is rendered as

$$y(t) = G_m(z^{-1})u(t) + D(t)$$

with $D(t)$ the effect of the disturbances on the output y. The process model G_m is described by the right polynomial matrix fraction

$$G_m(z^{-1}) = L(z^{-1})R^{-1}(z^{-1})$$

with L and R polynomial matrices in the delay operator z^{-1}. The disturbance model is taken as

$$D(t) = q(z^{-1})\Phi^{-1}(z^{-1})\nabla^{-d}v(t)$$

with v a column vector of white noise inputs. The quantities q and Φ are polynomial matrices in the delay operator, with Φ diagonal, and

$$\nabla^d = (1 - z^{-1})^d I$$

Algorithm

The first step of the algorithm is to find the approximate inverse model

$$\tilde{G}_m^{-1}(z^{-1}) = R(z^{-1})\Gamma^{-1}(z^{-1})$$

for the plant. The polynomial matrix $\Gamma(z^{-1})$ follows by the spectral factorization

$$\Gamma^*(z^{-1})\Gamma(z^{-1}) = L^*(z^{-1})Q_1 L(z^{-1}) + (\nabla^d)^* Q_2 \nabla^d$$

The superscript * denotes the adjoint, that is,

$$\Gamma^*(z^{-1}) = \Gamma^T(z)$$

The spectral factor $\Gamma(z^{-1})$ needs to have all its roots outside the unit circle.

The controller H may now be expressed as

$$H(z^{-1}) = \tilde{G}_m^{-1}(z^{-1})F(z^{-1}), \quad F(z^{-1}) = T(z^{-1})q^{-1}(z^{-1})$$

where the polynomial matrix T, together with the polynomial matrix P, is the solution of the two-sided equation

$$L^*(z^{-1})Q_1 q(z^{-1}) = \Gamma^*(z^{-1})T(z^{-1}) + zP^*(z^{-1})\nabla^d \Phi(z^{-1})$$

Application to the batch process problem

To apply the algorithm to the batch control problem we need to obtain the process and disturbance models in the required form. Inspection of

$$G_m(z^{-1}) = \begin{bmatrix} -77.0z^{-3} & 0.33z^{-3} \\ 0 & 0.03z^{-3} \\ 0 & 0.1z^{-3} \end{bmatrix} = L(z^{-1})R^{-1}(z^{-1})$$

reveals that

$$L(z^{-1}) = \begin{bmatrix} -77.0z^{-3} & 0.33z^{-3} \\ 0 & 0.03z^{-3} \\ 0 & 0.1z^{-3} \end{bmatrix}. \qquad R(z^{-1}) = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

Furthermore, writing

$$\begin{bmatrix} \dfrac{1-0.6z^{-1}}{1-z^{-1}} & 0 & 0 \\ 0 & \dfrac{1}{(1-0.5z^{-1})(1-z^{-1})} & 0 \\ 0 & 0 & \dfrac{1-0.55z^{-1}}{1-z^{-1}} \end{bmatrix}$$

$$
= \begin{bmatrix} 1-0.6z^{-1} & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1-0.55z^{-1} \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1-0.5z^{-1} & 0 \\ 0 & 0 & 1 \end{bmatrix}^{-1} (1-z^{-1})^{-1}
$$

$$
= q(z^{-1})\Phi^{-1}(z^{-1})\nabla^{-d}
$$

we see that

$$
q(z^{-1}) = \begin{bmatrix} 1-0.6z^{-1} & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1-0.55z^{-1} \end{bmatrix} (1-z^{-1})^{-1}, \quad \Phi(z^{-1}) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1-0.5z^{-1} & 0 \\ 0 & 0 & 1 \end{bmatrix}, \quad d = 1
$$

The weighting matrices, finally, are selected as

$$
Q_1 = \begin{bmatrix} 0.01 & 0 & 0 \\ 0 & 8 & 0 \\ 0 & 0 & 2.25 \end{bmatrix}, \quad Q_2 = 0
$$

Computation The first step of the computation is to define the input data. Thus, the first few lines of the script `demoB.m` are

```
% demoB

% Script for the demo "Control of a batch process"
% Define the data

L = [-77*z^-3 0.33*z^-3; 0 0.03*z^-3; 0 0.1*z^-3];

R = eye(2,2);

theta = diag([1-0.6*z^-1 1 1-0.55*z^-1]);

Phi = diag([1 1-0.5*z^-1 1]);

d = 1;

Q1 = diag([0.01 8 2.25]);

Nabla = (1-z^-1);
```

Spectral factorization

The first computational step is to determine the polynomial matrix Γ by spectral factorization:

```
% Spectral factorization

Gamma = spf(L'*Q1*L);
```

The result is

```
Gamma

Constant polynomial matrix: 2-by-2

Gamma =
7.7          -0.033

0.00074       0.17
```

Solution of the two-sided equation

The next computational step is to solve the two-sided matrix equation

$$L^*(z^{-1})Q_1 q(z^{-1}) = \Gamma^*(z^{-1})T(z^{-1}) + zP^*(z^{-1})\nabla^d\Phi(z^{-1}) \tag{4}$$

for the matrices *T* and *P.* For this we use the routine `axybc`. This solves the two-sided polynomial matrix equation

$$AX + YB = C$$

To turn (4) into a polynomial matrix equation we need to multiply it by a suitable power *n* of z^{-1} so that $z^{-n}L^*(z^{-1})$ and $z^{-n}\Gamma^*(z^{-1})$ both are polynomial and also $U(z^{-1}) = z^{-n+1}P^*(z^{-1})$ is a polynomial matrix.

Choosing $n = 3$ we have

```
% Solution of the two-sided equation

n = 3;

A = z^-n*Gamma';

B = Nabla^d*Phi;

C = L'*Q1*theta;

[T,U] = axybc(A,B,C);
```

T turns out to be 2×3 matrix of degree 1, as predicted by the theory (MacGregor and Harris, 1987):

```
T =

        -0.04         -0.00025 + 0.00012z^-1     -5.6e-005

        -3.8e-006      2.6 - 1.2z^-1               0.59
```

The controller

The controller transfer matrix is

$$
\begin{aligned}
H(z^{-1}) &= \tilde{G}_m^{-1}(z^{-1})F(z^{-1}) = R(z^{-1})\Gamma^{-1}(z^{-1})T(z^{-1})q^{-1}(z^{-1}) \\
&= f(z^{-1})q^{-1}(z^{-1})
\end{aligned}
$$

where

$$
f(z^{-1}) = R(z^{-1})\Gamma^{-1}(z^{-1})T(z^{-1})
$$

is polynomial because Γ is a constant matrix. We compute the controller as

```
% Compute the controller H = phi/theta

phi = R/Gamma*T;
```

The result is

```
phi =

        -0.0052        0.065 - 0.03z^-1      0.015
```

```
-1.3e-016     15 - 7.1z^-1          3.4
```

Response to disturbances

We study the effect of disturbances $D(t)$ at the plant output such that

$$y(t) = G_m(z^{-1})u(t) + D(t)$$

The closed-loop response to the disturbances $D(t)$ follows from the sensitivity matrix S:

$$\begin{bmatrix} y_1(t) \\ y_2(t) \\ y_3(t) \end{bmatrix} = S(z^{-1}) \begin{bmatrix} D_1(t) \\ D_2(t) \\ D_3(t) \end{bmatrix}$$

It is not difficult to find from the block diagram of Fig. 27 that if the plant model exactly matches the plant then

$$S(z^{-1}) = I - G_m(z^{-1})H(z^{-1})$$

It follows that

$$\begin{aligned} S(z^{-1}) &= I - L(z^{-1})R^{-1}(z^{-1})R(z^{-1})\Gamma^{-1}(z^{-1})T(z^{-1})q^{-1}(z^{-1}) \\ &= [q(z^{-1}) - L(z^{-1})\Gamma^{-1}(z^{-1})T(z^{-1})]q^{-1}(z^{-1}) \\ &= y\,(z^{-1})q^{-1}(z^{-1}) \end{aligned}$$

where

$$y(z^{-1}) = q(z^{-1}) - L(z^{-1})\Gamma^{-1}(z^{-1})T(z^{-1})$$

Thus we add the lines

```
% Compute the sensitivity matrix S = psi/theta

psi = theta-L/Gamma*T;
```

Disturbance impulse and step responses

The Control Toolbox and even more SIMULINK are very well equipped to compute, plot and manipulate time responses. We stay within the confines of the Polynomial Toolbox, however, and use long polynomial division to compute the impulse and step responses to disturbances (see the demo "Computing the covariance function of an ARMA process".)

```
% Compute and plot the disturbance impulse response matrix r

% Apply long division to psi/theta

n = 10;

[q,r] = longrdiv(psi,theta,n);

% Plot the data

figure; clf

k = length(r);

for i = 1:k
```

```
   for j = 1:k
      subplot(k,k,(i-1)*k+j)
      plot(0:n,r{:}(i,j))
      grid on; axis([0 n -1.5 1.5])
   end
end
```

Matlab produces the plot of Fig. 28 for the impulse response matrix. The step responses may be computed similarly and are shown in Fig. 29.

```
% Compute and plot the disturbance step response matrix s
% Apply long division to (psi/theta)*1/(1-z^-1)
n = 10;
[q,s] = longrdiv(psi,theta*(1-z^-1),n);
% Plot
figure; clf
k = length(s);
for i = 1:k
```

```
   for j = 1:k
      subplot(k,k,(i-1)*k+j)
      plot(0:n,s{:}(i,j))
      grid on; axis([0 n -1.5 1.5])
   end
end
```

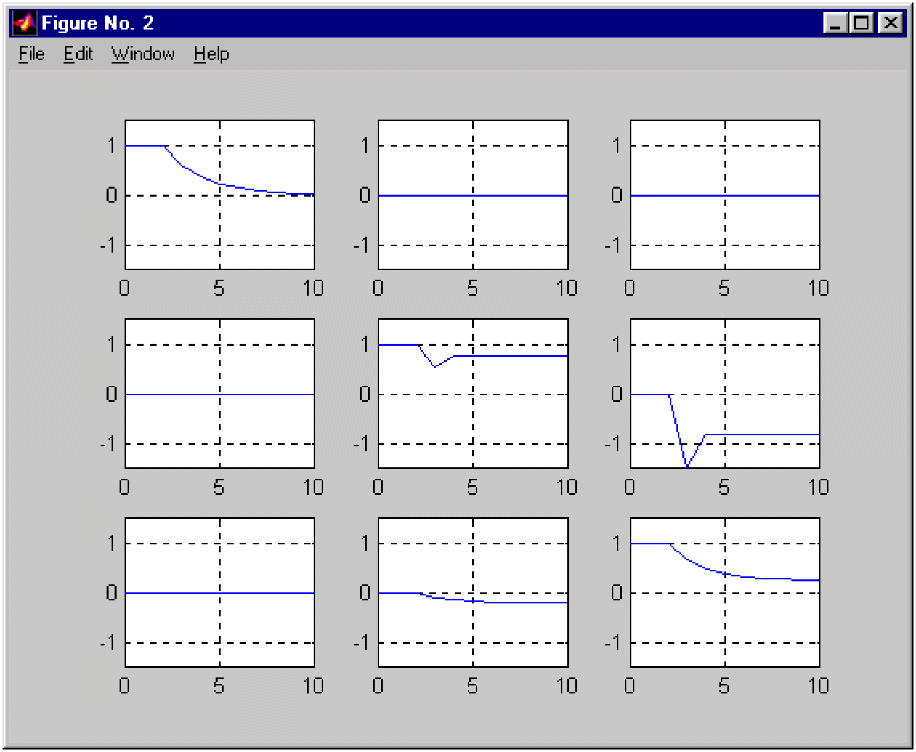


Fig. 28. Disturbance impulse response matrix

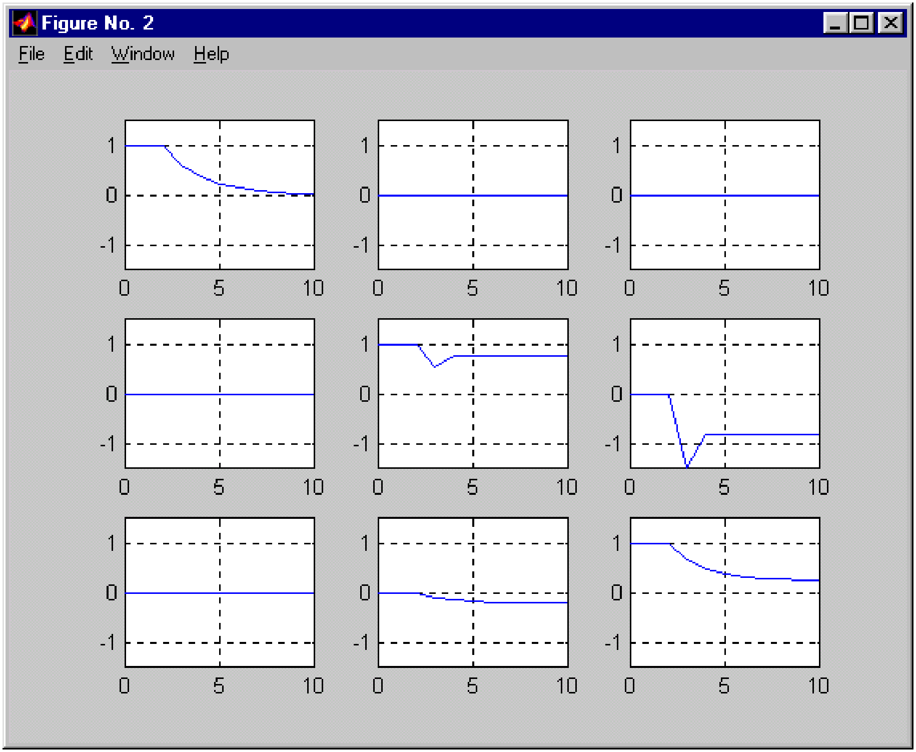


Fig. 29. Disturbance step response matrix

Assessment

Inspection of the step response matrix shows this:

- The (1,2) and (1,3) entries of the step response matrix are identical to zero. This means that the first output is insensitive to the second and third components of the disturbances.
- Likewise, the (2,1) and (3,1) entries are zero. This means that the second and third outputs are insensitive to the first component of the disturbance.
- All nonzero entries show a dead time of three time steps. This is a consequence of the dead time of the process.
- The step response in the (1,1) entry eventually goes to zero. This means that constant disturbances are suppressed in this channel.
- The step responses in the remaining zero entries do not approach zero, which means that constant disturbances in the second and third channel are not suppressed. The reason for this is that there are three components to the disturbance but only two control inputs so that a full degree of freedom is lacking. The resulting loss in performance is divided between the second and third channels. The balance of this division may be shifted by adjusting the entries of the weighting matrix Q_1.

Polynomial solution of the SISO mixed sensitivity *H*-infinity problem

Synopsis Mixed sensitivity optimization is a powerful design tool for linear single-degree-of-freedom feedback systems. It allows simultaneous design for performance and robustness, and relies on shaping the two critical closed-loop sensitivity functions with frequency dependent weights.

To obtain satisfactory high-frequency roll-off nonproper weighting functions may be needed. These cannot be directly handled in the conventional state space solution of the *H*-infinity problem. Nonproper weighting functions present no problems in the frequency domain solution of the *H*-infinity problem. The frequency domain solution may be implemented in terms of polynomial matrix manipulations.

We present the mixed sensitivity problem and its solution for single-input-single-output plants. Step by step the Toolbox function `mixeds` is developed that implements the algorithm. A simple but relevant example is used for illustration.

Introduction To demonstrate the capabilities of the Polynomial Toolbox we implement the polynomial solution of a mixed sensitivity *H*-infinity problem. Consider the single-degree-of-freedom feedback loop of Fig. 30. *P* is the plant transfer matrix and *C* the compensator transfer matrix.

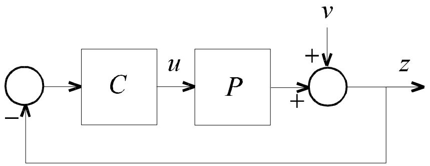


Fig. 30. Single-degree-of-freedom feedback loop

The sensitivity matrix S and input sensitivity matrix U of the feedback system are defined as

$$S = (I + PC)^{-1}, \quad U = C(I + PC)^{-1}$$

The mixed sensitivity problem is the problem of minimizing the infinity norm $\|H\|_\infty$ of

$$H = \begin{bmatrix} W_1 SV \\ W_2 UV \end{bmatrix}$$

with suitably chosen weighting matrices W_1 and W_2, and V a suitably chosen shaping matrix. In the SISO case the infinity norm is given by

$$\|H\|_\infty^2 = \sup_{-\infty < w < \infty} \left(|W_1(jw)S(jw)V(jw)|^2 + |W_1(jw)U(jw)V(jw)|^2 \right)$$

The problem may be reduced to a “standard” *H*-infinity problem by considering the block diagram of Fig. 31, which includes the weighting and shaping filters.

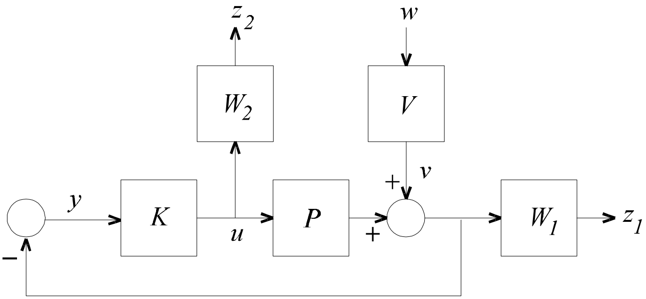


Fig. 31. Mixed sensitivity configuration

The diagram of Fig. 31 defines a standard problem whose “generalized plant” has the transfer matrix

$$G = \left[\begin{array}{c|c} W_1V & W_1P \\ 0 & W_2 \\ \hline -V & -P \end{array}\right]$$

The closed-loop transfer matrix from w to

$$z = \begin{bmatrix} z_1 \\ z_2 \end{bmatrix}$$

is precisely the function H whose infinity-norm we wish to minimize.

There are many ways to solve this problem, but only the frequency domain solution (Kwakernaak, 1996) allows the generalized plant to have a nonproper transfer matrix G. To enhance robustness at high frequencies it usually is necessary to make the weighting filter W_2 nonproper, which in turn makes G nonproper.

For simplicity we develop the solution for the SISO case only. Suppose that

$$P=\frac{n}{d},\quad V=\frac{m}{d},\quad W_1=\frac{a_1}{b_1},\quad W_2=\frac{a_2}{b_2}$$

where the numerators and denominators are (scalar) polynomials. Note that P and V have the same denominators d — this makes partial pole placement possible (Kwakernaak, 1993).

Numerical example

We study in particular the following numerical example (Kwakernaak, 1963):

$$P(s)=\frac{1}{s^2},\quad V(s)=\frac{1+s\sqrt{2}+s^2}{s^2},\quad W_1(s)=1,\quad W_2(s)=c(1+rs)$$

where we let $c = r = 1$. Note that W_2 is nonproper and, hence, the generalized plant G is nonproper. The various polynomials are defined by the following command lines:

```
% Define the data

n = 1; d = s^2; m = s^2+s*sqrt(2)+1;

c = 1; r = 1;  a1 = 1; b1 = 1; a2 = c*(1+r*s); b2 = 1;
```

Left coprime polynomial matrix fraction representation

The frequency domain solution of the H_∞ problem of Kwakernaak (1996) is based on polynomial matrix manipulations. It requires G to be represented in left coprime polynomial matrix fraction form. The desired left coprime factorization is

$$G = \left[\begin{array}{c|c} \dfrac{a_1 m}{b_1 d} & \dfrac{a_1 n}{b_1 d} \\ 0 & \dfrac{a_2}{b_2} \\ \hline -\dfrac{m}{d} & -\dfrac{m}{d} \end{array}\right] = \underbrace{\left[\begin{array}{cc|c} b_1 & 0 & a_1 \\ 0 & b_2 & 0 \\ 0 & 0 & d \end{array}\right]}_{[D_1 \;\; D_2]}{}^{-1} \underbrace{\left[\begin{array}{c|c} 0 & 0 \\ 0 & a_2 \\ -m & n \end{array}\right]}_{[N_1 \;\; N_2]}$$

The partitioning is needed for the solution of the H_∞ problem. The following code lines define the various polynomial matrices:

```
% Define the various polynomial matrices
D1 = [b1 0; 0 b2; 0 0];
D2 = [a1  ; 0   ; d  ];
N1 = [ 0;  0; -m ];
N2 = [ 0; a2; -n ];
```

These code lines are actually taken from the macro `mixeds`, which automates the entire computation.

Frequency domain solution of the *H*-infinity problem

As usual in the solution of the H_∞ problem we consider the problem of finding a compensator that stabilizes the system and makes the infinity norm of the closed-loop transfer matrix less than or equal to a given number g . Define the rational matrix

$$\Pi_g^{-1} = \begin{bmatrix} N_2^* \\ D_2^* \end{bmatrix} (D_1 D_1^* - \frac{1}{g^2} N_1 N_1^*)^{-1} \begin{bmatrix} N_2 & D_2 \end{bmatrix}$$

The * denotes conjugation, that is, $A^*(s) = A^T(-s)$. In the next subsection it is seen how a spectral factorization

$$\Pi_g^{-1} = M_g^* J M_g$$

of Π_g^{-1} may be obtained. The spectral factor M_g is a square rational matrix such that both M_g and M_g^{-} have all their poles in the open left-half plane. The matrix J is a signature matrix of the form

$$J = \begin{bmatrix} I & 0 \\ 0 & -I \end{bmatrix}$$

where the two unit matrices do not necessarily have the same dimensions.

Given this spectral factorization, *all* compensators whose norm is less than the number g are of the form $K = X^{-1} Y$, where

$$\begin{bmatrix} X & Y \end{bmatrix} = \begin{bmatrix} I & U \end{bmatrix} M_g$$

U is a rational stable matrix such that $\|U\|_\infty \leq 1$. In particular one may chose $U = 0$.

Spectral factorization

We consider the spectral factorization

$$\Pi_g^{-1} = M_g^* J M_g$$

It requires the following steps.

1. Do the polynomial spectral cofactorization

$$D_1 D_1^* - \frac{1}{g^2} N_1 N_1^* = Q_g J_o Q_g^{-1}$$

2. Perform the “left-to-right” conversion

$$Q_g^{-1} \begin{bmatrix} N_2 & D_2 \end{bmatrix} = \Delta_g \Lambda_g^{-1}$$

3. Do the polynomial factorization

$$\Delta_g^* J_o \Delta_g = \Gamma_g^* J \Gamma_g$$

Then the desired spectral factor is

$$M_g = \Gamma_g \Delta_g^{-1}$$

The first polynomial spectral factorization

The first spectral factorization may actually be done analytically, because we have

$$D_1 D_1^* - \frac{1}{\gamma^2} N_1 N_1^* = \begin{bmatrix} D_1 & N_1 \end{bmatrix} \begin{bmatrix} I & 0 \\ 0 & -\frac{1}{\gamma^2} I \end{bmatrix} \begin{bmatrix} D_1 & N_1 \end{bmatrix}^*$$

$$= \begin{bmatrix} b_1 & 0 & 0 \\ 0 & b_2 & 0 \\ 0 & 0 & m \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -\frac{1}{\gamma^2} \end{bmatrix} \begin{bmatrix} b_1 & 0 & 0 \\ 0 & b_2 & 0 \\ 0 & 0 & m \end{bmatrix}^*$$

Inspection shows that if each of the polynomials b_1, b_2 and m is strictly Hurwitz then the desired spectral factorization may be rendered in slightly modified form as

$$D_1 D_1^* - \frac{1}{\gamma^2} N_1 N_1^* = Q J_\gamma^{-1} Q^*, \qquad Q = \begin{bmatrix} b_1 & 0 & 0 \\ 0 & b_2 & 0 \\ 0 & 0 & m \end{bmatrix}, \qquad J_\gamma = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -\gamma^2 \end{bmatrix}$$

Left-to-right conversion

The computation of the left-to-right conversion

$$Q^{-1} \begin{bmatrix} N_2 & D_2 \end{bmatrix} = \Delta \Lambda^{-1}$$

may easily be coded.

```
% Left-to-right conversion
Q = diag([b1 b2 m]);
```

```
[Del,Lam] = lmf2rmf([N2 D2],Q)
```

The result is

```
Del =
    -0.71                  0.71 + s
     0.71 + 1.7s + s^2     0.71 + 0.71s
    -0.71                 -0.71 + s
Lam =
     0.71 + s     0.71
    -0.71         0.71 + s
```

Second polynomial factorization

The second spectral factorization now takes the form

$$\Delta^* J_g \Delta = \Gamma_g^* J \Gamma_g$$

It is not difficult to write the necessary code lines

```
% Define gamma
gamma = 4;
% Spectral factorization
Jgamma = eye(3); Jgamma(3,3)= -gamma^2;
```

```
DelDel = Del'*Jgamma*Del;

[Gam,J] = spf(DelDel)

Gam =

    -1.3e+002 - 50s - s^2        -2.2e+002 - 0.046s

     1.3e+002 + 48s + 0.17s^2     2.2e+002 - 3.8s

J =

     1     0

     0    -1
```

Computation of the compensator

To determine the numerator Y and denominator X of the compensator we need to compute

$$[X \quad Y] = [I \quad U] M_g = [I \quad U]\, \Gamma_g\, \Delta^{-1}$$

where for simplicity we choose $U = 0$. It is advantageous to implement this computation as a right-to-left conversion

$$[I \quad U] \Gamma_g\, \Delta^{-1} = z^{-1} [x \quad y]$$

so that the compensator transfer function is

$$K = X^{-1}Y = x^{-1}y = \frac{y}{x}$$

Again this may be coded straightforwardly:

```
% Computation of the compensator
xy = rmf2lmf([1 0]*Gam,Lam);
x = xy(1,1), y = xy(1,2)
```

The output is

```
x =
     2.4e+002 + 1.6e+002s + 50s^2 + s^3
y =
     63 + 1.8e+002s - 0.66s^2
```

Computation of the closed-loop poles

The closed-loop characteristic polynomial is

$$f = dx + ny$$

We use it to test closed-loop stability and to compute the closed-loop poles.

```
% Computation of the closed-loop characteristic polynomial
% and closed-loop poles
```

```
phi = d*x+n*y;

clpoles = roots(phi)

clpoles =

 -46.7980

  -0.9727 + 0.6271i

  -0.9727 - 0.6271i

  -0.7071 + 0.7071i

  -0.7071 - 0.7071i
```

Approaching the optimal solution

It is easy to collect the command lines listed so far in an m-script and to run the script repeatedly for different values of g .

We first run the script with `gamma` = 4. The closed-loop poles all have negative real parts, and, hence, the closed-loop system is stable.

Next, we run the macro with `gamma` = 3.5. The script returns

```
clpoles =

   4.9995

  -0.9590 + 0.5600i

  -0.9590 - 0.5600i
```

```
    -0.7071 + 0.7071i

    -0.7071 - 0.7071i
```

The closed-loop system is unstable. Hence, `gamma` has been chosen too small.

Note that four of the five closed-loop poles do not change much with `gamma`. The fifth pole is very sensitive to changes in `gamma`.

We test this dependence by running the script several times for different values of `gamma` without showing the output. Table 5 shows the results. Apparently as `gamma` decreases the fifth pole crosses over from the left- to the right-half complex plane, but does so through infinity.

For the final run we take `gamma` = 3.9515, close to the optimal value but such that the closed-loop system is stable. The corresponding numerator polynomial *y* and the denominator polynomial *x* of the compensator are

```
y =

     4e+003 + 1.1e+004s - 0.66s^2

x =

   1.5e+004 + 1e+004s + 3e+003s^2 + s^3
```

Table 5. Dependence of the fifth pole on `gamma`

gamma	the fifth pole

4	–46.798
3.5	4.9995
3.75	11.369
3.875	30.297
3.9375	173.86
3.9688	–127.80
3.95	315.38
3.951	3153.8
3.9515	–2987.6

Calculation of the optimal compensator

Inspection of the numerator and denominator polynomials *y* and *x* of the compensator obtained for `gamma` = 3.9515 shows that their coefficients are large, except for the leading coefficients.

In fact, we may cancel the leading coefficients and simplify *y* and *x*. This amounts to cancelling the compensator pole-zero pair and eliminating the corresponding closed-loop pole that pass through infinity as `gamma` passes through the optimal value.

Recalculation of the closed-loop poles after this cancellation confirms that the large closed-loop pole has disappeared. These calculations are performed by typing a few simple command lines

```
y{2} = 0; x{3} = 0;

y = y/x{2}, x = x/x{2}

y =

     1.3 + 3.8s

x =

      5.1 + 3.4s + s^2

phi = d*x+n*y;

clpoles = roots(phi)

clpoles =

  -0.9703 + 0.6201i

  -0.9703 - 0.6201i

  -0.7075 + 0.7072i

  -0.7075 - 0.7072i
```

Assessment of the design

To assess the design we calculate and plot the sensitivity function *S* and the complementary sensitivity function *T* of the closed-loop system. They are given by

$$S = \frac{dx}{f}, \quad T = \frac{ny}{f}$$

We type in the command lines

```
omega = logspace(-2,2); j = sqrt(-1);

S = bode(pol2mat(d*x),pol2mat(phi),omega);

T = bode(pol2mat(n*y),pol2mat(phi),omega);

figure(1);

loglog(omega,abs(S),'k'); hold on

loglog(omega,abs(T),'b'); grid on

text(.1,.01,'S'), text(10,.01,'T')
```

Fig. 32 shows the plots of *S* and *T*. For a discussion of the design and the mixed sensitivity design methodology see Kwakernaak (1993).

Alternative spectral factorization

As we have seen, the spectral factorization behaves poorly as the optimal solution is approached. The reason is that the factorization becomes “noncanonical” (Kwakernaak, 1996). This difficulty may be remedied by using an alternative form for the second polynomial spectral factorization. Instead of factoring

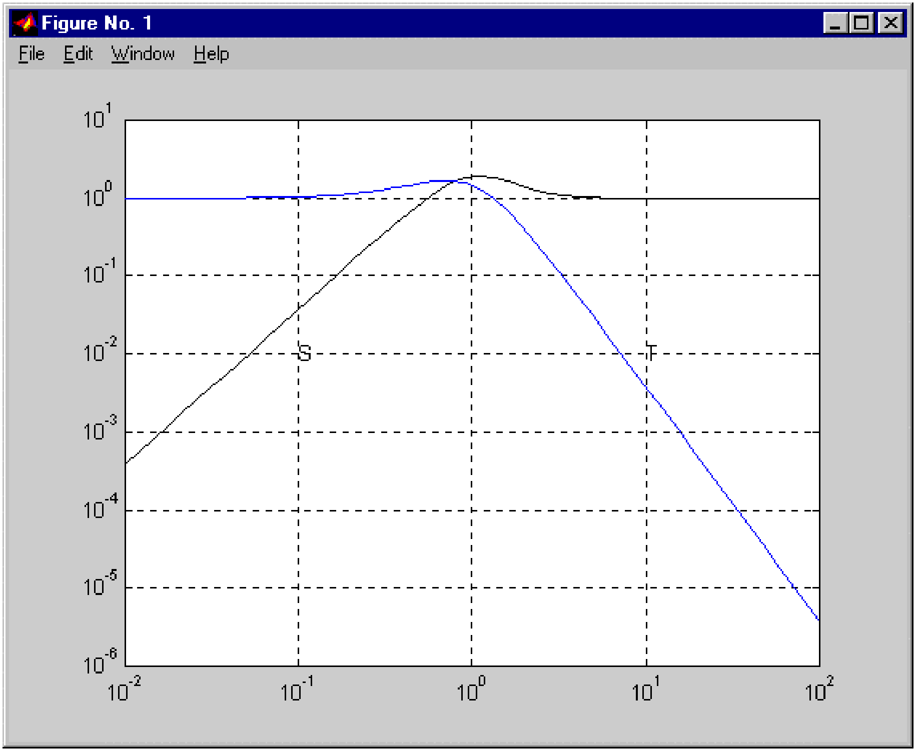


Fig. 32. Magnitude plots of the sensitivity functions

$$\Delta^* J_g \Delta = \Gamma_g^* J \Gamma_g$$

we rearrange the factorization as

$$\Delta^* J_g \Delta = \Gamma_g^* L^{-1} \Gamma_g$$

L is diagonal in the form

$$L = \text{diag}(L_1, -L_2)$$

where L_1 and L_2 are diagonal nonnegative-definite but not unit matrices. If the factorization is close to noncanonical then L is close to nonsingular. The large numbers disappear.

The alternative factorization is obtained by an option in the `spf` command. The computation of the compensator is not affected, so we only need to change the code line that contains the `spf` command to

```
[Gam,J] = spf(DelDel,'nnc');
```

Rerunning the script with this modification for `gamma` = 3.9515 shows that the large numbers have disappeared. We also see that instead of the large closed-loop poles and corresponding large pole and zero of the compensator we now have a closed-loop pole, compensator pole and zero at –1:

```
y,x
```

```
y =

     1.3 + 5.1s + 3.8s^2

x =

     5.1 + 8.4s + 4.4s^2 + s^3

rootsx = roots(x), rootsy = roots(y), clpoles

rootsx =

  -1.6787 + 1.5027i

  -1.6787 - 1.5027i

  -1.0000

rootsy =

   -1.0000

   -0.3476

clpoles =

  -0.7071 + 0.7071i

  -0.7071 - 0.7071i

  -1.0002
  -0.9715 + 0.6197i

  -0.9715 - 0.6197i
```

Cancellation of the common root of *y* and *x* leads to the same optimal compensator that was previously obtained:

```
x/(s-rootsx(3)), y/(s-rootsy(1))

ans =

     5.1 + 3.4s + s^2

ans =

     1.3 + 3.8s
```

Automating the search

Now that the numerical instability in the computation of the compensator has been removed it is simple to automate the search process. We implement a binary search that involves the following steps:

1. Specify a minimal value `gmin` and a maximal value `gmax` for `gamma`.
2. Test if a stabilizing compensator is found at `gamma` = `gmax`. If not then stop.
3. Test if a stabilizing compensator is found at `gamma` = `gmin`. If yes then stop.
4. Let `gamma` = (`gmin+gmax`)/2. If a stabilizing compensator is found then let `gmax` = `gamma`, otherwise let `gmin` = `gamma`.
5. If `gmax-gmin` is greater than a prespecified accuracy then return to 4.

6. Retain the solution for `gamma` = `gmax` and stop.

This search algorithm has been implemented in the macro `mixeds`. Calling the routine in the form

```
gmin = 3.5; gmax = 4; accuracy = 1e-4;

[y,x,gopt] =
mixeds(n,m,d,a1,b1,a2,b2,gmin,gmax,accuracy,'show')
```

executes the search while showing the intermediate results. The search stops if `gmax-gmin` is less than the input parameter `accuracy.` This is the output:

```
gamma   test result

-----   -----------

4   stable

3.5   unstable

3.75  unstable

3.875   unstable

3.9375 unstable

3.96875 stable

3.95313 stable
```

```
3.94531 unstable

3.94922 unstable

3.95117 stable

3.9502  unstable

3.95068 unstable

3.95093 stable

3.95081 stable

3.95074 stable

Cancel root at -0.999999

y =

     1.3 + 3.8s

x =

     5.1 + 3.4s + s^2

gopt =

    3.9507
```

Cancelling coinciding pole-zero pairs

The numerator *x* and the denominator *y* of the optimal compensator turn out to have a common root. This often happens. The precise location of this spurious pole-zero pair is unpredictable. It needs to be cancelled in the compensator transfer function $C = y/x$.

Rather than relying on one of the polynomial division routines we write a few dedicated code lines. Suppose that the numerator *y* has a root *z*. Then by polynomial division we have for the denominator *x*

$$x(s) = q(s)(s - z) + r$$

where the remainder *r* is a constant. By substituting $s = z$ we see that actually $r = x(z)$. Expanding the polynomials *x* and *q* as

$$x(s) = x_n s^n + x_{n-1} s^{n-1} + \cdots + x_0, \qquad q(s) = q_{n-1} s^{n-1} + q_{n-2} s^{n-2} + \cdots + q_0$$

it follows that the quotient *q* and the remainder *r* may recursively be computed as

$$\begin{aligned} q_{n-1} &= x_n \\ q_{k-1} &= x_k + q_k z, \quad k = n, n-1, \cdots, 1 \\ r &= x_0 + q_0 z \end{aligned}$$

Note that this way of computing $r = x(z)$ is nothing else than Horner's algorithm. If the remainder *r* is small then we cancel the factor $s–z$. By the same algorithm the factor $s–z$ may be cancelled from the numerator *y*.

These are the necessary code lines. A tolerance `tolcncl` is used to test if the remainder is small.

```
% Cancel any common roots of xopt and yopt
rootsy = roots(yopt);
xo = xopt{:}; degxo = deg(xopt);
yo = yopt{:}; degyo = deg(yopt);
for i = 1:length(rootsy)
   z = rootsy(i);
   % Divide xo(s) by s-z
   q = zeros(1,degxo);
   q(degxo) = xo(degxo+1);
   for j = degxo-1:-1:1
      q(j) = xo(j+1)+z*q(j+1);
   end
   % If the remainder is small then cancel
   % the factor s-z both in xo and in yo
   if abs(xo(1)+z*q(1)) < tolcncl*norm(xo,1)
      xo = q; degxo = degxo-1;
      p = zeros(1,degyo);
      p(degyo) = yo(degyo+1);
      for j = degyo-1:-1:1
         p(j) = yo(j+1)+z*p(j+1);
      end
      yo = p; degyo = degyo-1;
      if show
        disp(sprintf('\nCancel root at %g\n',z));
      end
   end
end
```

The function mixeds

The full command

```
[y,x,gopt] = ...
    mixeds(n,m,d,a1,b1,a2,b2,gmin,gmax,accuracy,tol,'show')
```

includes a four-dimensional optional tolerance parameter

```
tol = [tolcncl tolstable tolspf tollr]
```

which allows to fine tune the macro. For a description of the various tolerances consult the manual page for `mixeds`.

Applications in behavioral system theory

Introduction

Behavioral system theory (Polderman and Willems, 1998) is a very general approach to system theory. It defines a system as a relation between the signals that constitute the environment of the system. A distinction is made between *latent* (or internal) and *manifest* (or external) signals but not necessarily between input and output signals.

Polynomial matrices play an important role in the behavioral theory of linear systems. The purpose of this demo is to show that the Polynomial Toolbox provides many useful routines for dealing with problems and questions in behavioral linear system theory.

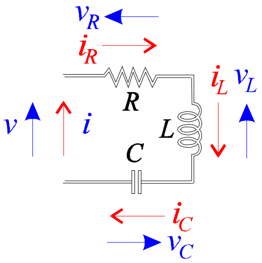


Fig. 33. RCL network

Example

By way of illustration we consider the simple electrical network of Fig. 33. The signals of interest are the currents i_R, i_L and i_C through the resistor, inductor and capacitor, respectively, the voltages v_R, v_L and v_C across these same network elements, the current i that flows into the network, and the voltage v across the network. The relations between the signals are given by

- the element equations

resistor: $v_R = Ri_R$

inductor: $v_L = L\dfrac{di}{dt}$

capacitor: $i_C = C\dfrac{dv_C}{dt}$

- the interconnection equations (Kirchhoff's laws)

$$i = i_R, \quad i_R = i_L, \quad i_L = i_C$$

$$v = v_R + v_L + v_C$$

All these equations can be combined in the form

$$Q(\frac{d}{dt})z = 0 \tag{5}$$

where the polynomial matrix Q and the vector-valued signal z are given by

$$Q(s) = \begin{bmatrix} R & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & sL & 0 & 0 & -1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & -sC & 0 & 0 \\ 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 & -1 & -1 & 0 & 1 \end{bmatrix}, \qquad z = \begin{bmatrix} i_R \\ i_L \\ i_C \\ v_R \\ v_L \\ v_C \\ i \\ v \end{bmatrix}$$

Kernel representation

The representation (5) is called a *kernel representation* of the system, because it defines the system as all signals that are in the kernel (or null space) of the operator $Q(d/dt)$.

Full and manifest behavior

The equation (5) characterizes what is known as the *full behavior* of the system, because it includes all latent and manifest signals. Such a characterization is typically obtained when setting up the system equations from first principles. For the electrical network the latent variables ℓ and the manifest variables w could be chosen as

$$\ell = \begin{bmatrix} i_R \\ i_L \\ i_C \\ v_R \\ v_L \\ v_C \end{bmatrix}, \qquad w = \begin{bmatrix} i \\ v \end{bmatrix}$$

If the latent variables are eliminated from the behavior then the *manifest behavior* is obtained.

Computation of the manifest behavior

We consider how to compute the manifest behavior from the full behavior. Partitioning the matrix Q as $Q = \begin{bmatrix} Q_1 & Q_2 \end{bmatrix}$ we have for the full behavior

$$Q_1(s)\ell + Q_2(s)w = 0$$

where s represents the differentiation operator. Let the rows of the polynomial matrix N be a minimal polynomial basis for the left null space of Q_1. Then the manifest behavior has the kernel representation

$$R(\frac{d}{dt})w = 0$$

where

$$R(s) = N(s)Q_2(s)$$

Example

In the example, choose the numerical values

$$R = 3, \quad L = 1, \quad C = \frac{1}{2}$$

We input the polynomial matrix Q as

```
R = 3; L = 1; C = 1/2;
Q = [ R   0   0  -1   0   0   0   0
      0  s*L  0   0  -1   0   0   0
      0   0   1   0   0 -s*C  0   0
      1  -1   0   0   0   0   0   0
      0   1  -1   0   0   0   0   0
     -1   0   0   0   0   0   1   0
      0   0   0  -1  -1  -1   0   1];
```

The polynomial matrices Q_1 and Q_2 follow as

```
Q1 = Q(:,1:6); Q2 = Q(:,7:8);
```

From this, the matrices N and R may be computed as

```
N = null(Q1')'; R = N*Q2
```

```
R =

     2 + 3s + s^2    -s
```

Hence, the manifest behavior is described by the differential equation

$$(2+3\frac{d}{dt}+\frac{d^2}{dt^2})i(t)-\frac{d}{dt}v(t)=0$$

Controllability

A behavioral system in kernel representation $R(d/dt)w=0$ is *controllable* iff the polynomial matrix R is left prime (Polderman and Willems, 1998). For the example we can easily verify controllability by typing

```
isprime(R)
```

```
ans =

     1
```

Image representation

A well-known fact from behavioral theory is that given a controllable kernel representation

$$R(\frac{d}{dt})w=0$$

there always exists an equivalent *image representation* of the form

$$w = M(\frac{d}{dt})\ell$$

with ℓ a latent variable. Inspection shows that the columns of $M(s)$ need to be a minimal basis for the right null space of $R(s)$. Thus, for the example we may compute M as

```
M = null(R)
```

```
M =

          s

          2 + 3s + s^2
```

Hence, the image representation of the system is

$$i = \frac{d}{dt}\ell$$

$$v = (2 + 3\frac{d}{dt} + \frac{d^2}{dt^2})\ell$$

Note that physically the latent variable ℓ corresponds to Cv_C, which happens to be the charge of the capacitor.

State representation

Another fact from behavioral system theory is that the manifest behavior of any finite-dimensional linear system may be represented in the equivalent state space form

$$\begin{aligned}\dot{x} &= Ax + Bw\\ 0 &= Cx + Dw\end{aligned}$$

The latent variable x is the state of the system. This representation is by no means unique, and may be constructed in the following way from the kernel representation $R(d/dt)w = 0$.

First, assume that R is row-reduced. If it is not then it may be unimodularly transformed to be row-reduced without changing the behavior.

Let S be a square, nonsingular, row-reduced matrix whose row degrees equal the row degrees of R, and chosen such that $S^{-1}R$ is left coprime. Obviously the behavior $R(d/dt)w = 0$ is equivalent to the behavior defined by

$$S(\frac{d}{dt})z = R(\frac{d}{dt})w, \quad z = 0$$

Let

$$\begin{aligned}\dot{x} &= Ax + Bw\\ z &= Cx + Dw\end{aligned}$$

be a minimal realization of the left coprime matrix fraction $S^{-1}R$. Then clearly

$$\begin{aligned}\dot{x} &= Ax + Bw\\ 0 &= Cx + Dw\end{aligned}$$

is a state realization of the behavior.

Example

We further pursue the example. Given the row-reduced polynomial matrix

```
R

R =

     2 + 3s + s^2    -s
```

we choose

```
S = s^2;
```

A minimal realization of $S^{-1}R$ is obtained as

```
[A,B,C,D] = lmf2ss(R,S)

A =

     0     1

     0     0

B =

    3.0000   -1.0000

    2.0000         0

C =

     1     0
```

```
D =

      1      0
```

Construction of IO system

So far no distinction has been made among the manifest variable between “input” and “output” variables, which the obvious connotation of “causes” for inputs and “effects” for outputs. Indeed, in the electrical network example there is no a priori reason which of the two manifest variables v and i is the input and which is the output because the circuit could be connected to a voltage or a to a current source.

If no compelling reason exists to designate certain manifest variables as inputs and other variables as outputs then *possible* partitionings of the manifest variables into sets of input variables and output variables may be determined on the basis of the (plausible) requirement that the outputs are causally affected by the inputs.

To make this more concrete, suppose that we have a kernel representation $R(d/dt)w = 0$ such that R has full row rank with rank equal to r. Select r components of w as outputs and permute the components of w and the corresponding columns of R such that the selected outputs are the first r components of w. We write the resulting kernel representation as

$$\begin{bmatrix} R_1 & -R_2 \end{bmatrix} \begin{bmatrix} y \\ u \end{bmatrix} = 0$$

with y the output and u the input. Then the proposed selection of outputs and inputs is deemed acceptable if $R_1^{-1}R_2$ is proper.

Consider the state representation

$$\begin{aligned}\dot{x} &= Ax + Bw\\ 0 &= Cx + Dw\end{aligned}$$

where C and D have r rows*.* Then in this context a causal IO representation may be constructed by selecting r columns of D so that the resulting square submatrix is nonsingular. By designating the corresponding entries of w as outputs and the remaining entries as inputs the equation $0 = Cx + Dw$ may be rearranged as $y = cx + du$. Substitution of y into the equation $\dot{x} = Ax + Bw$ results in a state representation of the IO system.

Example

We found that the electrical network has the state representation

$$\dot{x} = \underbrace{\begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}}_{A} x + \underbrace{\begin{bmatrix} 3 & -1 \\ 2 & 0 \end{bmatrix}}_{B} \begin{bmatrix} i \\ v \end{bmatrix}$$

$$0 = \underbrace{\begin{bmatrix} 1 & 0 \end{bmatrix}}_{C} x + \underbrace{\begin{bmatrix} 1 & 0 \end{bmatrix}}_{D} \begin{bmatrix} i \\ v \end{bmatrix}$$

Inspection of the matrix D shows that the only available option is to take the current i as the output and, hence, the voltage as the input. Substitution of the resulting output

$$i = \begin{bmatrix} -1 & 0 \end{bmatrix} x$$

into the first equation yields the corresponding state differential equation. Not wanting to make any mistakes we invoke MATLAB:

```
c = [-1 0]; B1 = B(:,1); B2 = B(:,2);
a = A+B1*c, b = B2
a =
   -3.0000    1.0000
   -2.0000         0
b =
   -1.0000
         0
```

Thus, we have the IO representation

$$\dot{x} = \underbrace{\begin{bmatrix} -3 & 1 \\ -2 & 0 \end{bmatrix}}_{a} x + \underbrace{\begin{bmatrix} -1 \\ 0 \end{bmatrix}}_{b} v$$

$$i = \underbrace{\begin{bmatrix} -1 & 0 \end{bmatrix}}_{c} x$$

This selection of input and output corresponds to connecting the network to a voltage source. Connecting it to a current source does *not* lead to a causal IO system because the admittance

$$\frac{2+3s+s^2}{s}$$

of the network is nonproper.

Reference

Introduction

In this chapter we review the properties and structure of polynomial matrix objects as defined and used in the Polynomial Toolbox.

Global properties and initialization

Global polynomial properties

The global polynomial properties store important general information that is implicitly used by many operations and functions when they are called during a Polynomial Toolbox session.

- When a polynomial matrix is created but no symbol is explicitly specified to denote its variable the current value of a global property called *variable symbol* is used.
- If all computations are to be performed up to a certain relative tolerance then it is sufficient to set properly the global property named *zeroing tolerance*.
- The format used for displaying polynomial matrices is stored in another global property called *display format*.

- Finally, a global property flag called *verbose level* controls the amount of information that is displayed during the computation.

The system of global properties is quite flexible. The Polynomial Toolbox is equipped with several special macros that modify the global property values and allow to customize the session environment. On the other hand, many functions allow a temporary local change of a property just by inserting its value among the input arguments.

All four global properties are listed in Table 6 along with their admissible values. The global properties are created and assigned at the beginning of every Polynomial Toolbox session by the initialization macro `pinit`. Unless explicitly specified the properties are set to the default values of Table 7.

The global property values can be modified anytime during the session in several ways: by a general macro `gprop`, by specialized macros such as `pformat`, `symbol`, `tolerance`, and `verbose`, or simply by re-using `pinit`.

Table 6. Global properties and their admissible values

property name	**function**	**admissible values**
variable symbol	implicitly used variable symbol	s, p, z, q, z^{-1}, d
zeroing tolerance	global tolerance used for zeroing	any real number

display format	how to display polynomial matrices	`symb, symbs, symbr,`
		`coef, rcoef, block`

verbose level	the amount of messages displayed	`no, yes`

Table 7. Default property values

property name	**default value**
variable symbol	s
zeroing tolerance	10^{-8}
display format	`coef`
verbose level	`no`

Initialization

Every Polynomial Toolbox session starts with the initialization command `pinit`:

```
pinit
```

```
Polynomial Toolbox initialized. To get started, type one of
these: helpwin or poldesk. For product information, visit
www.polyx.com or www.polyx.cz.
```

This function creates the global polynomial properties and assigns their default values. Any other settings may be assigned directly by including the desired values in arbitrary order such as

```
pinit z 1.0e-6 symb;
```

which is the same as

```
pinit z, symb 1.0e-6;
```

If you forget to initialize the Polynomial Toolbox and open with another Polynomial Toolbox command then an error message is returned:

```
pol(1)

??? Error using ==> pol/pol

Use PINIT to initialize Polynomial Toolbox.
```

Displaying and altering global property values

The current values of the global properties may be displayed by macro `gprop`. Typing

```
gprop
```

returns the following table

```
Global polynomial properties:

PROPERTY NAME:      CURRENT VALUE:    AVAILABLE VALUES:

variable symbol     s                'x','p','z^-
1','d','z','q'

zeroing tolerance   1e-008            any real number

verbose level       no               'no', 'yes'

display format      symbs            'symb', 'symbs',
'symbr'
```

`'coef'`, `'rcoef'`, `'block'`

The table summarizes both the current and the admissible values of all global properties. The same macro may also alter one or more global property values. Typing

```
gprop z^-1
```

switches the global variable symbol to z^{-1} leaving all other properties unchanged. Similarly,

```
gprop 0
```

sets the global zeroing tolerance equal to 0, that is, deactivates the process of zeroing completely.

Alternatively, the global property values may be changed by re-using `pinit`. However, the effects of `gprop` and `pinit` differ slightly: `gprop` leaves all unspecified properties unchanged while `pinit` sets them to their default value.

Users are strongly recommended to handle the global properties only by the Polynomial Toolbox functions (`pformat`, `symbol`, `tolerance`, `verbose`, or `pinit` and `gprop`). Only experienced programmers should employ the following details.

The global properties are stored in a global structure called `PGLOBAL`. The structure is invisible in the MATLAB workspace or elsewhere unless it is declared as `global`. After such a declaration the self-explanatory fields of the structure may be viewed . For instance,

```
global PGLOBAL

PGLOBAL

PGLOBAL =

     ZEROING: 1.0000e-008

     VERBOSE: 'no'

     FORMAT: 'coef'

     VARIABLE: 's'
```

One should be careful when modifying the global structure directly. The global properties are not tested for correctness. Mistakenly introduced incorrect values may therefore cause unexpected behavior. Whenever such a strange behavior is observed or suspected type `pinit` to restore the standard conditions.

Polynomial Matrices

Introduction

The Polynomial Toolbox offers extensive tools to manipulate and analyze polynomials and polynomial matrices. Polynomials are scalar functions or algebraic entities described as

$$a(s) = a_0 + a_1 s + a_2 s^2 + \cdots + a_d s^d$$

The integer d is the *degree*, the real (or complex) numbers $a_0, a_1, \cdots, a_d$ are the *coefficients* and the operator s is called the *variable* or *indeterminate*. For instance,

$$a(s) = s + 2s^2 + 3s^3$$

is a polynomial of degree 3 with coefficients 0, 1, 2, and 3, respectively, in the variable s.

Polynomial matrices are matrices whose entries are polynomials, such as

$$P(s) = \begin{bmatrix} a_{11}(s) & a_{12}(s) \\ a_{21}(s) & a_{22}(s) \end{bmatrix}$$

Alternatively, polynomial matrices may be thought of as matrix polynomials

$$P(s) = P_0 + P_1 s + \cdots + P_d s^d$$

where d is the *polynomial matrix degree,* while the real (or complex) constant matrices $P_0, P_1, \cdots, P_d$ are the *matrix coefficients*. For instance, the polynomial matrix

$$P(s) = \begin{bmatrix} 1+s & 2 & 3+s^2 \\ -s & 2s & 3-s^2 \\ 1+s+s^4 & 1+5s^2 & 8 \end{bmatrix}$$

may also be written as

$$P(s)=\begin{bmatrix}1&2&3\\0&0&3\\1&1&8\end{bmatrix}+\begin{bmatrix}1&0&0\\-1&2&0\\1&0&0\end{bmatrix}s+\begin{bmatrix}0&0&1\\0&0&-1\\0&5&0\end{bmatrix}s^2+\begin{bmatrix}0&0&0\\0&0&0\\1&0&0\end{bmatrix}s^4$$

Polynomial matrix objects

For convenience, the Polynomial Toolbox provides a customized data structure for polynomial matrices called a *polynomial matrix object* or simply POL object. This object encapsulates the polynomial matrix data and makes it possible to manipulate polynomial matrices as single entities rather than collections of data vectors or matrices. For instance

```
a = pol([0 1 2 3],3);
```

creates a polynomial matrix (POL) object a that stores the scalar polynomial

$$a(s)=s+2s^2+3s^3$$

This polynomial may now be manipulated by referring to the single MATLAB variable a. Its roots may for instance be calculated by typing

```
roots(a)
```

The polynomial matrix object implementation relies on the object-oriented programming capabilities of MATLAB version 5 and later. Polynomial matrix objects are MATLAB structures with an additional flag indicating their *class* (POL). Like MATLAB structures, they have pre-defined fields called *object properties*. For polynomial matrix objects these properties include degree, size, coefficients, variable symbol and user data. The functions operating on a particular object are called the object *methods*. These may include customized versions of simple operations such as addition or multiplication. For instance,

```
b = 1+a
```

performs the polynomial matrix addition

$$b(s) = 1 + a(s) = 1 + s + 2s^2 + 3s^3$$

The object-specific versions of such standard operations are referred to as *overloaded* operations. For more details on objects, methods, and object-oriented programming refer to Chapter 14 of *Using MATLAB*.

Creating polynomial matrix objects

The functions `pol` and `lop` create polynomial matrices. These functions take the coefficients and degree as input and produce a polynomial matrix object (POL) that stores this data in a single MATLAB variable. In what follows we show how to construct polynomial matrix objects.

Common construction

The primary polynomial matrix object constructor is the function `pol`. If the coefficients of a scalar polynomial

$$a(s) = a_0 + a_1 s + a_2 s^2 + \cdots + a_d s^d$$

are arranged into a constant row vector $A = \begin{bmatrix} a_0 & a_1 & \cdots & a_d \end{bmatrix}$, with the coefficients ordered according to ascending powers of s, then the command

```
a = pol(A,d)
```

creates the scalar polynomial $a(s)$. The variable a is a polynomial matrix object containing all the necessary information. For instance,

```
a = pol([1 2 3],2)
```

creates the polynomial

$$a(s) = 1 + 2s + 3s^2$$

Polynomial matrices are created similarly. If the matrix coefficients of a polynomial matrix

$$P(s) = P_0 + P_1 s + P_2 s^2 + \cdots + P_d s^d$$

are arranged in a constant block row matrix $A = \begin{bmatrix} P_0 & P_1 & \cdots & P_d \end{bmatrix}$ with the coefficients matrices ordered according to ascending powers of s, then the command

```
P = pol(A,d)
```

creates the polynomial matrix $P(s)$. The variable P is now a polynomial matrix object containing all the necessary information. For instance,

```
P = pol([[1 0; 0 1] [0 1; 1 0]],1)
```

creates the 2×2 polynomial matrix

$$P(s) = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} + \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} s = \begin{bmatrix} 1 & s \\ s & 1 \end{bmatrix}$$

Alternative construction

The notation used above for polynomial matrices is common, yet, especially in pure mathematics, indexing the coefficients by descending powers is sometimes preferred. In this case the standard notation is

$$P(s) = P_0 s^d + P_1 s^{d-1} + \cdots + P_{d-1} s^1 + P_d$$

To accommodate this convention the Polynomial Toolbox offers an alternative polynomial matrix constructor called[1] `lop.` It works just as `pol` but expects the coefficients in reversed order. Thus, the scalar polynomial $a(s)$ considered previously may be also be constructed by the command

```
a = lop([3 2 1],2)
```

The polynomial matrix $P(s)$ we created before may similarly be obtained as

```
P = lop([[0 1; 1 0] [1 0; 0 1]],1)
```

Building polynomials by special functions

The Polynomial Toolbox provides several functions that create special simple polynomials. The commands `s, p, z, q, zi, d` and `v` create single term polynomials (*monomials*) consisting only of the first power of s, p, z, q, z^{-1}, d and of the current value of the variable symbol, respectively. These commands can be used directly to create other scalar polynomials by arithmetic operations such as

```
a = 1+s+s^2+s^4
```

or

---

[1] Note the reversed order of the characters letters of the command.

```
b = 1+z*(2-z)
```

Polynomial matrices can easily be built by concatenating monomials, for instance

```
P = [zi zi^2; 1 zi^5]
```

Several monomials can be assembled simultaneously by the function `mono` that works with an integer vector input argument, for instance,

```
mono([1 2 4 5])

ans =

     s     s^2     s^4     s^5
```

It also works with an integer matrix argument such as

```
mono(ones(3))

ans =

     s     s     s

     s     s     s

     s     s     s
```

More sophisticated polynomials may easily be produced by combining `mono` with other functions such as in

```
sum(mono(1:10))
```

```
ans =

  s + s^2 + s^3 + s^4 + s^5 + s^6 + s^7 + s^8 + s^9 + s^10
```

Important note No MATLAB function can work in a workspace in which a MATLAB variable of the same name exists. In particular, each of the monomial functions listed before is unusable as soon as a MATLAB variable of the same name is created. To re-activate the function, the user must clear the variable. For instance, the monomial function

```
s

ans =

     s
```

does not work if you created a variable with the same name, such as

```
s = [0 0]; s

s =

     0     0
```

To make the function active again, just clear the troublesome variable

```
clear s

s

ans =
s
```

This behavior is rather standard in MATLAB . Compare the performance of the built-in functions `i, j, pi, eps` and alike.

Displaying polynomial matrix objects

Despite its unique internal representation a polynomial matrix can be displayed in various ways. By default, the polynomial matrix

$$P(s) = \begin{bmatrix} 1+0.5s & s^2 \\ 1 & 0.33333s \end{bmatrix}$$

which is generated by the command

```
P = [1+0.5*s s^2; 1 0.33333*s];
```

is displayed in symbolic short format `symbs` as

```
P

P =

       1 + 0.5s     s^2

       1            0.33s
```

The display format uses short (1 or 2 digit) coefficients and is easily readable. There are two more symbolic formats available: symbolic long (`symb`) with longer decimal coefficients

```
pformat symb, P

P =

     1 + 0.5s     s^2

     1            0.33333s
```

and symbolic rational (`symbr`) with rational coefficients

```
pformat symbr, P

P =

     1 + 1/2*s     s^2

     1             33333/100000*s
```

The symbolic formats look natural and are easily readable for small matrices of low degrees. For large polynomial matrices or matrices of high degree these formats are slow and result in messy output. If the matrix is large, has high degree, or if we need to see its coefficients in full detail, then one of the numeric formats may be used. The most important is the `coef` format:

```
pformat coef, P

Polynomial matrix in s: 2-by-2,  degree: 2

P =

Matrix coefficient at s^0 :
     1     0

     1     0

Matrix coefficient at s^1 :

     0.5000         0

     0    0.3333

Matrix coefficient at s^2 :

     0     1

     0     0
```

Also available is the reversed order coefficient format `rcoef`

```
pformat rcoef, P

Polynomial matrix in s: 2-by-2,  degree: 2

P =

Matrix coefficient at s^2 :

     0     1

     0     0

Matrix coefficient at s^1 :
    0.5000         0
         0    0.3333

Matrix coefficient at s^0 :

     1     0
     1     0
```

For completeness, the matrix coefficients may also be arranged in a block row constant matrix by the `block` format

```
pformat block, P

Polynomial matrix in s: 2-by-2,  degree: 2

P =

    1.0000         0    0.5000         0         0
1.0000
    1.0000         0         0    0.3333         0
0
```

that was used in Polynomial Toolbox version 1.6. In all the numeric formats the scalar coefficient format may be set by MATLAB command `format`.

Changing display formats

The way a polynomial matrix is displayed is controlled by global property *display format*. To switch among the various formats one can use one of the general commands `pinit` and `gprop` or even better the special function `pformat` that has been used previously. So either of the commands

```
pinit coef
```

```
gprop coef
```

```
pformat coef
```

assigns the display format to be `coef` causing all subsequent polynomial matrices to be displayed by their coefficients. Note that `gprop` and `pformat` work exactly as described while `pinit` also re-initializes the toolbox so that all other global properties are restored to their default values. Therefore, the use of `pformat` is recommended here.

Plotting polynomial matrices

Besides the symbolic and numeric display formats a polynomial matrix can also be shown in the form of 2-D and 3-D color plots. Look how a polynomial matrix

```
M = [ s   s^3    s^4    s^5

     1+s s^2-1 3*s^3   s^4

     1   2+s   2*s^2  s+s^3

     0    2     3*s   s^2+s ];
```

appears in the 2-D plot of Fig. 34, which is created by the command

```
pplot(M)
```

Note that different degrees are expressed by different colors. A 3-D color plot such as in Fig. 35 can be obtained by a similar macro called `pplot3.` It works as follows

```
pplot3(M)
```

The lego-like plot may be rotated with the help of the mouse.

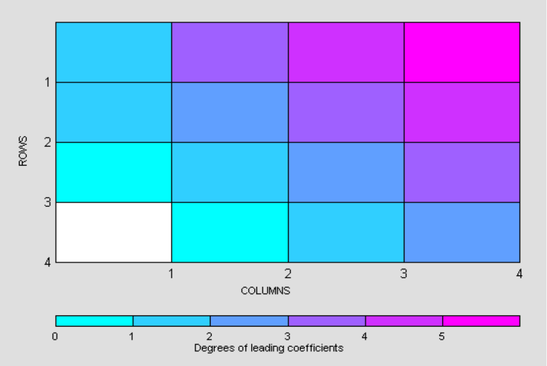


Fig. 34. Two-dimensional plot of a polynomial matrix

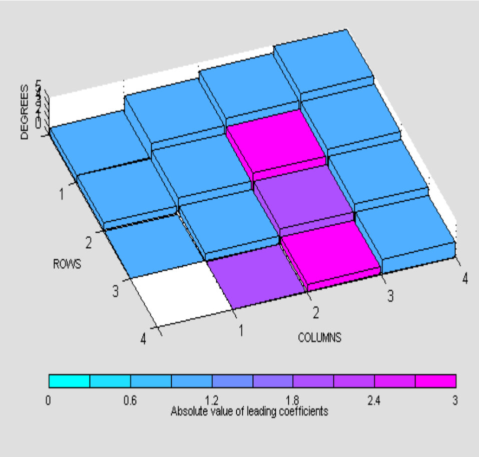


Fig. 35. Three-dimensional plot of a polynomial matrix

Polynomial matrix properties

The preceding section shows how to create POL objects that encapsulate polynomial matrices. Optionally, POL objects may store additional information called *user data*. This section gives a complete overview of the POL properties, that is, various pieces of information that characterize polynomial matrices or may be attached to them. Type `help pprop` for online help on the available POL properties.

From the point of view of implementation the POL properties are the various fields in the POL internal data structure. These fields have names. To define a polynomial object values need to be assigned to each of these fields[2].

Degree, size and coefficients

A polynomial matrix is defined by its degree, size and coefficients. In a polynomial object P the degree is stored in the field `P.deg.` Another field denoted `P.size` contains the size of the polynomial matrix. It is a two-element row vector with nonnegative integral entries. It can be assigned any nonnegative integer value or `-Inf` for a zero polynomial matrix. The matrix coefficients of P are stored in a further field called `P.coef` that is a 3-D array, composed of the constant matrix coefficients as horizontal layers (ordered from bottom to top). As these three basic

---

[2] A comparison with the Control System Toolbox 2.0 is now in order. Users familiar with the CST may find our system very similar. In fact, it is simpler: Instead of the *Property/Value* pairs encountered in CST we use just one argument called *Value* or sometimes *PropertyValue*. As the number of POL properties is small each property can uniquely be identified by its value.

properties must be consistent they can only be joined when the polynomial matrix is created by a constructor. In addition they can only be changed jointly as a result of some operation or function.

Variable symbol property

Every polynomial matrix has a symbol to denote its variable. This symbol is stored in a separate field of the POL object called `P.variable.` Its value is not crucial for most operations but it makes a difference when the matrix is displayed and also in certain special functions.

The Polynomial Toolbox offers six different strings[3] that can be used as a symbol denoting the polynomial matrix variable: *s,* (default) *p, z, q,* z^{-1} and *d.* One of them is included in every polynomial matrix object. The symbol is used whenever the object is displayed and also for certain special macros that need to distinguish between continuous-time and discrete-time operations or functions. In particular,

- *s* and *p* are used for differential operators in continuous-time systems
- *z* and *q* play the role of forward-shift discrete-time operators,
- and finally, z^{-1} and *d* stand for delay (backward-shift) discrete-time operators.

When creating a polynomial matrix object with the function `pol` the variable string is set equal to the current value of the global variable symbol, which is *s* by default. To select another string one can toggle the global variable before by typing the `pol` command or use the `pprop` function afterwards.

---

[3] This was inspired by the CST with a minor difference: The symbol d was added and the meaning of q was changed to be equivalent to z (rather than z^{-1}).

User data property

In addition to the obligatory properties described above another property called user data has been included for convenience. It is stored in the field `P.user` and may take any value acceptable by MATLAB.

The polynomial matrix properties are summarized in Table 8.

Table 8. Polynomial matrix properties

Property name	Field in the POL object	Admissible values
degree	`P.deg`	nonnegative integer or `–Inf`
size	`P.size`	two-entry nonnegative integer vector
coefficients	`P.coef`	3-D double array
variable symbol	`P.variable`	`'s', 'p', 'z', 'q', 'z^-1', 'd'`
user data	`P.user`	any

Constant polynomial matrices

The Polynomial Toolbox operations and functions allow mixing polynomial objects and standard MATLAB matrices (2-D “double” arrays). Whenever necessary, the standard MATLAB matrix is automatically converted to a special POL object called constant polynomial matrix that naturally possesses a zero degree. For instance, the user can add the constant 1 to a polynomial $a(s)$ simply by typing

```
1+a
```

The sum is obtained by Polynomial Toolbox overloaded function `plus` that begins with the conversion of the standard MATLAB double variable 1 to a constant polynomial object `pol(1)` and proceeds by adding the two polynomial objects

```
pol(1)+a
```

in an obvious manner.

For practical reasons, a constant polynomial matrix is assigned an empty string variable symbol whenever it is directly created from a MATLAB “double” array:

```
c = pol(1)

Constant polynomial matrix: 1-by-1

c =

     1

c.variable

ans =

     ''
```

If the constant results from an operation on polynomials, however, then it inherits the variable symbol of its “parents,” for instance,

```
c=(1+s)-s

Constant polynomial matrix: 1-by-1
```

```
c =

     1

c.variable

ans =

     ''
```

Zero polynomial matrices A zero polynomial matrix is a particular case of a constant matrix that is characterized by zero coefficients. Its degree is always `–Inf` and its variable symbol is always an empty string:

```
Z=pol(zeros(2))

Zero polynomial matrix: 2-by-2,  degree: -Inf

Z =

     0     0

     0     0

pprop(Z)

ZERO POLYNOMIAL MATRIX

size            2-by-2

degree      -Inf
```

```
PROPERTY NAME:    CURRENT VALUE:     AVAILABLE VALUES:

variable symbol                     's','p','z^-
1','d','z','q'

user data         []                 arbitrary
```

Empty polynomial matrices

Similarly to empty MATLAB matrices, an empty polynomial matrix object is of size 0×0, $0\times m$ or $n\times 0$. It possesses an empty coefficient array, empty degree and empty variable symbol string. It may be constructed from an empty 2-D “double” array, such as

```
CE = ones(3,0)

CE =

   Empty matrix: 3-by-0

PE = pol(CE)

PE =

Empty polynomial matrix: 3-by-0

pprop(PE)

EMPTY POLYNOMIAL MATRIX

size        3-by-0
```

PROPERTY NAME:	CURRENT VALUE:	AVAILABLE VALUES:
`variable symbol`		`'s','p','z^-1','d','z','q'`
`user data`	`[]`	`arbitrary`

Properties of computed matrices

When a polynomial matrix arises from a standard operation or function its *size*, *degree* and *coefficients* are completely defined by the operation or function. Its *variable symbol* is set according to the symbols of the input matrices. If — usually by mistake — two input matrices have different variable symbols then the resulting symbol is set equal to the current value of the global variable symbol property.

Adding, multiplying, concatenating, etc., a constant never changes the variable symbol property.

User data, in contrast to all other properties, are not transferred through operations and functions. This property may only be assigned directly. It is empty for any polynomial matrix that results from an operation or function.

Setting POL properties

The three basic polynomial matrix properties – degree, size and coefficients – are automatically fixed at the moment the matrix is created. They can only be changed by various operations or functions on the matrix that are described later.

Adding the desired symbol string as an input argument of the constructor function sets the variable symbol property. Thus,

```
a = pol([1 1],1,'s')
```

creates a polynomial matrix in *s* while

```
b = pol([1 1],1,'z')
```

forms a polynomial matrix in *z*. Once the desired symbol is explicitly specified the global symbol value becomes irrelevant.

Similarly,

```
a = pol([1 1],1,'s','January 25')
```

sets the user data of *a*(*s*) to `'January 25'`.

The symbol and user data properties may also be assigned or changed by means of the macro `pprop`: the command

```
a = pprop(a,'d','12345')
```

changes the variable symbol of *a* to *d* and the user data to `'12345'`.

Displaying POL properties

The current and admissible values of the polynomial properties are displayed by `prop`:

```
pprop(a)
```

Accessing property values

One way to query polynomial matrix properties is by structure-like referencing. Thus, typing `P.size`, `P.deg`, `P.coef`, `P.variable` and `P.user` returns the size, degree, coefficients, variable and user data of a polynomial matrix `P`, respectively. Unlike for MATLAB structures you need not type the entire field name or be concerned with upper-case letters. Thus `P.size`, `P.Size`, `P.s` and `P.S` all return the same.

The variable symbol and user data may also be assigned in this manner. Thus, you can type

```
P.variable = 'z';
```

```
P.user = 'blabla';
```

and so on. For safety reasons, however, `P.size`, `P.deg` and `P.coef` are not allowed on the left-hand side of an `'='` symbol.

An alternative way to access certain polynomial matrix property values is to use special functions

```
deg
```

```
size
```

```
lcoef
```

and so on. These functions usually offer more information. For instance, the commands

```
deg(p,'row'), deg(p,'col'), deg(p,'ent') and deg(p,'diag')
```

return the row degrees, column degrees, entry degrees and diagonal degrees, respectively.

Indexing

The entries, rows, columns and sub-blocks of a polynomial matrix may be indexed as is usual in MATLAB. Thus, for the matrix

```
A = [1 s s^2; 2 2*s 2*s^2; 3 3*s 3*s^2]

A =

     1     s      s^2

     2     2s     2s^2

     3     3s     3s^2
```

the expressions

```
 A(3,3)

ans =

     3s^2

A(1,:)

ans =

     1     s     s^2

A(:,3)

ans =
     s^2

     2s^2

     3s^2

A(1:2,1:2)

ans =

     1     s

     2     2s
```

represent the bottom right entry, the first row, the first column and the upper left 2×2 sub-block of A. Similarly,

```
A(1,1) = 2+s+s^2

A =

     2 + s + s^2     s      s^2

     2               2s     2s^2

     3               3s     3s^2
```

assigns the (1,1) entry,

```
A(:,3) = []
```

```
A =

     2 + s + s^2        s

     2                  2s

     3                  3s
```

deletes the last column, etc[4].

Similarly, curly braces are employed to index polynomial matrix coefficients. Thus,

```
A{0}

ans =

     2     0

     2     0

     3     0
```

is the constant (zero power) coefficient matrix of A. The two kinds of indexing may be combined:

```
A{0}(1,1)

ans =
```

---

[4] Owing to a bug in Matlab 5.2 the Matlab function `end` fails to work when it is used to index a polynomial matrix.

```
        2
```

is the scalar constant coefficient of the (1,1) element.

Various degrees and leading coefficients

The degree of a polynomial matrix such as

```
A = [1 s s^2; s^3 s 1; 0 1 0]

A =

        1         s      s^2

        s^3       s      1

        0         1      0
```

may be identified in various ways:

```
deg(A)

ans =

        3

size(A,3)

ans =

        3

A.deg
```

```
ans =

     3
```

Besides this standard degree also the row degrees, column degrees and entry degrees may be obtained as

```
rowdeg = deg(A,'row')

rowdeg =

     2

     3

     0

coldeg = deg(A,'col')

coldeg =

     3     1     2

entdeg = deg(A,'ent')

entdeg =

     0     1     2

     3     1     0
```

```
   -Inf      0  -Inf
```

The leading coefficient matrices corresponding to these degrees are successively

```
Matrix_leading_coef_matrix = lcoef(A)
Matrix_leading_coef_matrix =
     0     0     0
     1     0     0
     0     0     0
Row_leading_coef_matrix = lcoef(A,'row')
Row_leading_coef_matrix =
     0     0     1
     1     0     0
     0     1     0
Column_leading_coef_matrix = lcoef(A,'col')
Column_leading_coef_matrix =
     0     1     1
     1     1     0
     0     0     0
Entry_leading_coef_matrix = lcoef(A)
Entry_leading_coef_matrix =
     0     0     0
     1     0     0
     0     0     0
```

Complex coefficients

The Polynomial Toolbox supports polynomial matrices with complex coefficients such as

```
C = [1+s 1;2 s]+i* [ s 1;1 s]
C =
     1+0i + (1+1i)s     1+1i
     2+1i               (1+1i)s
```

Many Toolbox operations and functions handle them accordingly. In addition, there are several special functions for complex coefficient polynomial matrices such as

```
real(C)
ans =
     1 + s     1
     2         s
imag(C)
ans =
     s     1
     1     s
C'
ans =
     1+0i - (1-1i)s     2-1i
     1-1i              -(1-1i)s
conj(C)
ans =
     1+0i + (1-1i)s     1-1i
     2-1i               (1-1i)s
```

Operations on polynomial matrices

Basic operations

All the basic arithmetic operations, such as addition, subtraction, multiplication, division without remainder (factor extraction), taking integer powers, may be performed on polynomial matrices by standard MATLAB operators because all these operations have been overloaded for POL objects. In fact, everything works just as for standard MATLAB matrices. For instance, consider two square polynomial matrices

```
A = [1 s; 1+s 1-s], B = [s^2 1; 2 1-s^2]
A =
     1         s
     1 + s     1 - s
B =
     s^2     1
     2       1 - s^2
```

The matrices are added by typing

```
C = A+B
```

```
C =

     1 + s^2     1 + s

     3 + s       2 - s - s^2
```

Subtracting B from C recovers the original matrix A

```
C-B

ans =

     1          s

     1 + s      1 - s
```

A and B are multiplied by typing

```
D = A*B

D =

     2s + s^2                  1 + s - s^3

     2 - 2s + s^2 + s^3        2 - s^2 + s^3
```

Now

```
D/B

ans =
     1         s

     1 + s     1 - s
```

which clearly equals `A`, while similarly `A\D` returns `B`:

```
A\D

ans =

     s^2     1

     2       1 - s^2
```

Note, however, that

```
D/A

Constant polynomial matrix: 2-by-2

ans =

     NaN     NaN

     NaN     NaN
```

as `D` is not divisible[5] from the right by `A`.

---

[5] As polynomial matrix multiplication is not commutative the matrix `D = A*B` is, without a remainder, divisible by `A` from the left but not from the right. That is, there exists no

Finally, taking powers

```
A^2

ans =

      1 + s + s^2      2s - s^2

      2 + s - s^2      1 - s + 2s^2
```

works as expected.

Zeroing

Theoretically, the degree of a polynomial

$$a(s) = a_0 + a_1 s + a_2 s^2 + \cdots + a_d s^d$$

is d whenever $|a_d| \neq 0$. In numerical computations, however, one often encounters the case of $|a_d|$ very small (much smaller than the other coefficients) yet non-zero. By way of example, consider two simple polynomials

$$f(s) = 2 + (1 + e)s + s^2$$
$$g(s) = 1 + s + s^2$$

where e is almost (but not quite) zero. When computing the difference

---

polynomial matrix `F` such that `D = F*A`. For right division with remainder see the manual page of the function `pdiv`.

$$f(s) - g(s) = 1 + e\,s$$

a question on its degree may arise[6].

It is necessary to compare e with the norms of the coefficients in *f* and *g* to decide whether or not the corresponding term should be deleted. This process is called *zeroing*. The performance of many algorithms for polynomial problems depends critically on the way zeroing is done, in particular when elementary operations are used.

When using the Polynomial Toolbox zeroing is automatically performed with a relative tolerance controlled by the global property called *zeroing tolerance.* To deactivate zeroing you must set the global property equal to zero. For instance, by default the following summation works correctly

```
A  = [ 1+0.7*s 0.2*s -0.9*s ];

r = sum(A)

Constant polynomial matrix: 1-by-1
```

---

[6] The phenomenon of subtracting two similar numbers is known and called *cancellation* by numerical mathematicians (see for instance Higham, 1996). Cancellation usually does not bring inaccuracy by itself. The problem is that it amplifies the effect of inaccuracy introduced previously. If the number e in f_2 is correct (coming from real data or otherwise) then the whole computation is accurate and the resulting difference $f - g$ is really of degree 1. If, on the other hand, e is incorrect (arising from inaccurate computing done before) then the correct degree of $f - g$ is of course 0.

```
r =

     1
```

If, however, zeroing is switched off by the command

```
gprop 0
```

then an incorrect result is returned[7]

```
r = sum(A)

r =

     1 - 1.1e-016s
```

In terms of value the inaccuracy seems to be small and not substantial as the coefficient r_1 is very small. In terms of degree, however, the inaccuracy becomes critical. As r_1 is the leading coefficient the resulting degree appears to be 1 while it should be 0. Whenever the computation to follow depends on the degree of r_1 such a small inaccuracy may become crucial and could lead to a complete failure of the calculation process.

Reliable algorithms for polynomial matrices are resistant to such episodes. For other routines, the zeroing facility is ready to help. For reasonable data

---

[7] The inaccuracy is caused here by the finite precision of the internal binary representation of the coefficients 0.7, 0.2 and –0.9. This becomes evident when typing `sum([0.7 0.2 –0.9])`, which returns `ans = -1.1102e-016`.

encountered in real life, the built-in zeroing with the default tolerance works quite well. For some special critical examples one must manually decrease the value of the global zeroing tolerance or employ a smaller local tolerance whenever necessary.

Conflict of variable symbols

The current version of the Polynomial Toolbox only supports operations between polynomial matrices in the same variable symbol such as $C(s) = A(s) + B(s)$. Operations with different polynomial variable symbols always result in a polynomial matrix having its variable equal to the current value of the global polynomial variable symbol. If for instance the current global variable symbol equals its default value s then upon typing

```
p+z
```

MATLAB returns

```
Warning: Inconsistent variables

> In c:\Matlab\toolbox\polbox2\@pol\plus.m at line
```

Bibliography

Books

Barmish, B. R. (1996). *New Tools for Robust stability of Linear Systems.* Macmillan, New York.

Barnett, S. (1983), *Polynomial and Linear Control Systems*, Pure and Applied Mathematics, Marcel Dekker, New York.

Bhattacharya, S. P., H. Chapellat and L. H. Keel, (1995). *Robust Control: The Parametric Approach.* Prentice Hall, Upper Saddle River, N. J.

Bini, D. and V. Pan (1994). *Polynomial and Matrix Computations*. Vol. I. *Fundamental Algorithms.* Birkhäuser, Basel.

Callier, F. M. and C. A. Desoer (1982). *Multivariable Feedback Systems*. Springer-Verlag, New York.

Chen, C. T. (1984). *Linear Systems Theory and Design.* Holt, Rinehart and Winston, New York.

Dorf, R. C. (1989), *Modern Control Systems*, 5th Ed. Addison-Wesley.

Gantmacher, F. R. (1960), *The Theory of Matrices*, Vols. I and II, Chelsea Publishing Company, New York.

Gohberg, I., P. Lancaster and I. Rodman (1982), *Matrix Polynomials*, Academic Press, New York.

Golub, G. H. and C. F. Van Loan (1989), *Matrix Computations*, John Hopkins University Press, second edition.

Higham, N. J. (1996), *Accuracy and Stability of Numerical Algorithms*. SIAM, Philadelphia, 1996.

Kailath, T. (1980), *Linear Systems,* Prentice Hall, Englewood Cliffs, N. J.

Kucera, V. (1979). *Discrete Linear Control: The Polynomial Equation Approach*. Wiley, Chichester.

Kucera, V. (1991). *Analysis and Design of Discrete Linear Control Systems.* Prentice-Hall, London.

Polderman, J. W. and J. C. Willems (1998), *Introduction to Mathematical Systems Theory.* Springer, New York, etc.

Rosenbrock, H. H. (1970), *State Space and Multivariable Theory*. Nelson, London.

Rugh, W. J. (1993), *Linear System Theory.* Prentice-Hall, Englewood Cliffs, NJ.

Stefanidis, P., A. P. Paplinski and M. J. Gibbard (1992), *Numerical Operations with Polynomial Matrices: Application to Multi-Variable Dynamic Compensator Design*. Lecture Notes in Control and Information Sciences, Vol. 171, Springer-Verlag.

Van der Waerden, B. L. (1966), *Modern Algebra*, Frederic Ungar Publishing Co., New York. Volumes I and II, Sixth Edition.

Vardulakis, A. I. G. (1991), *Linear Multivariable Control: Algebraic Analysis and Synthesis Mehods.* Wiley, New York.

Wolovich, W. A. (1974), *Linear Multivariable Systems*, Springer, New York.

Papers

Beelen, T. G. J., G. J. van den Hurk and C. Praagman (1988), “A new method for computing a column reduced polynomial matrix.” *Systems and Control Letters*, Vol. 10, pp. 217–224.

Bitmead, R. R., S. Y. Kung, B. D. O. Anderson and T. Kailath (1978), “Greatest Common Divisor via Generalized Sylvester and Bézout Matrices.” *IEEE Transactions on Automatic Control,* Vol. 23, No. 6, pp. 1043–1047.

Callier, F. M. (1985), “On polynomial matrix spectral factorization by symmetric factor extraction,” *IEEE Trans. Aut. Control* 30, pp. 453–464.

Clements, D.J. (1993). “Rational spectral factorization using state-space methods.” *Systems and Control Letters*, 20, pp. 335–343.

Forney Jr., G. D. (1975), “Minimal bases of rational vector spaces with applications to multivariable linear systems.” *SIAM Journal on Control,* Vol. 13, No. 3, pp. 493–520.

Henrion, D. and M. Sebek (1998a), “Numerical methods for polynomial rank evaluation.” *Proceedings of the IFAC Conference on System Structure and Control*, Nantes, France, July 1998.

Henrion, D. and M. Sebek (1998b), “Efficient algorithms for polynomial matrix triangularization.” *Proc. IEEE Mediterranean Conference on Control and Automation*, Alghero, Sardinia, Italy, June 1998.

Henrion, D. and M. Sebek (1998c), “Symmetric matrix polynomial equation: Interpolation results,” *Automatica*, Vol. 34, No. 7, pp. 811–824.

Henrion, D. and M. Sebek (1998d) “An efficient numerical method for the discrete time symmetric matrix polynomial equation,” *Proceedings of the European Control Conference*, Brussels, Belgium, July 1997. Also to appear in *IEE Proceedings, Control Theory and Applications*, 1998.

Henrion, D. and M. Sebek (1998e), “An algorithm for polynomial matrix factor extraction.” Submitted for publication.

Henrion, D. and M. Sebek (1999), “Reliable numerical methods for polynomial matrix triangularization.” To appear as a regular paper in the *IEEE Transactions on Automatic Control,* possibly in March, 1999.

Hromcik, M. and Sebek, M. (1998), “New algorithms for polynomial matrices based on FTT,” submitted to *Automatica*.

Jezek, J. and V. Kucera (1985), “Efficient algorithm for matrix spectral factorization.” *Automatica*, Vol. 21, No. 6, pp. 663–669.

Kucera, V. and Sebek M. (1984). “On deadbeat controllers.” *IEEE Transactions on Automatic Control*, Vol. AC-29, pp. 719-722.

Kwakernaak, H. (1985). “Minimax frequency domain performance and robustness optimization of linear feedback systems.” *IEEE Transactions on Automatic Control,* Vol. AC-30, pp. 994–1004.

Kwakernaak, H. (1992). “Polynomial computation of Hankel singular values.” *Proc. 31th IEEE Decision and Control Conference,* Tucson, AZ, Dec. 16–18, pp. 3595–3599.

Kwakernaak, H. (1993), “Robust control and H_∞ optimization.” *Automatica*, 29, pp. 255–273.

Kwakernaak, H. (1996), “Frequency domain solution of the standard H_∞ problem.” In M. J. Grimble and V. Kucera (Eds), *Polynomial Mehotds for Control Systems Design.* Springer, London, etc.

Kwakernaak, H. (1998). “Frequency domain solution of the H_∞ problem for descriptor systems.” In Y. Yamamoto and S. Hara (Eds), *Learning, Control and Hybrid Systems.* Lecture Notes in Control and Information Sciences, vol. 241, Sprnger, London, etc.

Kwakernaak, H. and and M. Sebek (1994), “Polynomial *J*-spectral factorization.” *IEEE Transactions on Automatic Control,* Vol. 39, No. 2, pp 315–328.

Labhalla, S., H. Lombardi and R. Marlin (1996), “Algorithmes de calcul de la réduction de Hermite d'une matrice à coefficients polynomiaux.” *Theoretical Computer Science*, Vol. 161, No. 1–2, pp. 69–92.

MacGregor, J. F. and T. J. Harris (1987), "Design of multivariable Linear-Quadratic Controllers using transfer functions." *AIChE Journal*, Vol. 33, No. 9, pp. 1481–1495.

Meinsma, G. (1995), "Elementary proof of the Routh-Hurwitz test*." Systems & Control Letters* 25, pp. 237–242.

T. Söderström, J. Jezek and V. Kucera (1997), "An efficient and versatile algorithm for computing the covariance function of an ARMA process." Accepted for publication in *IEEE Trans. Signal Processing*.

Strijbos, R. C. W. (1995). "Calculation of right matrix fraction descriptions; an algorithm." Memorandum No. 1287, Faculty of Applied Mathematics, University of Twente.

Strijbos, R. W. C. (1996), "Calculation of right matrix fraction descriptions; an algorithm." Proceedings of the 4th IEEE Mediterranean Symposium on New Directions in Control and Automation, Maleme, Krete, Greece, June 10–13, pp. 478–482.

Zhang, S. Y., C. T. Chen (1983), "An algorithm for the division of two polynomial matrices." *IEEE Transactions on Automatic Control*, Vol. 28, No. 2, pp. 238–240.