WP14 16:00

Development of Advanced Control Design Software for Researchers and Engineers

Gary J. Balas* Andy Packard† John C. Doyle‡ Keith Glover§ and Roy Smith¶

MUSYN INC.
1009 Fifth St. SE
Minneapolis, MN 55414
612.378.1742 415.704.8730

1 Introduction

This paper gives a brief description of *The μ Analysis and Synthesis Toolbox* (μ-Tools), an advanced control design toolbox to be used in conjunction with MATLAB. μ-Tools introduces CONSTANT, SYSTEM, and VARYING matrix types and over a hundred commands to manipulate them, including H_∞ optimal control and μ analysis and synthesis functions. The VARYING type allows for matrices which are functions of an independent variable, and the SYSTEM type is a packed system matrix. The MATLAB implementation of these data types is described briefly in Section 2. A brief synopsis of the μ-Tools functions are listed in Section 4. As an example of their use, the following commands implement the first $D-K$ iteration of μ-synthesis for a SYSTEM matrix `sys`, using the μ-Tools commands `hinfsyn`, `frsp`, `mu`, `musynfit`, `mmult`, `minv`, `starp`, and `vplot`. A slightly more detailed example of a μ-synthesis design is given in Section 5.

```
>> minfo(sys)
system:   8 states     6 outputs     6 inputs
>> blk_struct = [2 2; 2 2]; nmeas=2; ncont=2;
>> omega = logspace(0,4,50); gamma_min=.8; gamma_max=6;
>> tolerance=.05;
>> [k1,g1,gf1]=...
hinfsyn(sys,nmeas,ncont,gamma_min,gamma_max,tolerance);
>> [bnds1,dv1,sens1,rp1] = mu(frsp(g1,omega),blk_struct);
>> [dsysl,dsysr]=...
musynfit('1st_iter',dv1,sens1,blk_struct,nmeas,ncont);
>> sys2 = mmult(dsysl,sys,minv(dsysr));
>> vplot('liv,m',bnds1,vnorm(frsp(starp(sys2,k1),omega)))
```

μ-Tools has undergone several years of refinement in short courses and at various test-site locations. It is based on experience over a ten-year period with MATLAB derived control toolboxes as well as the HoneyX system developed at Honeywell's System and Research Center, where the most extensive applications of the methods provided by μ-Tools has taken place. Beta test sites for the μ-Tools software include Honeywell, McDonnell Douglas, and Philips. The software package has also been used in graduate controls classes at Caltech, Georgia Tech, U. C. Berkeley, and the University of Minnesota, and a textbook based on the toolbox is in preparation.

μ-Tools is a unique software package in a number of respects. The most obvious feature is the introduction of control-specific data structures into the proven MATLAB environment. Perhaps more importantly, μ-Tools represents the cutting edge in making advanced control theory available both to the researcher and the applications engineer. It will continue to be an outlet for the authors' theoretical research results, and its ongoing use at key industry and government centers will insure that it has contact with the most advanced control applications. The μ-Tools authors have a strong commitment to applications and the development of theory with relevance to engineering design. Extensions under development include additional model reduction methods, analysis of systems with mixed complex and real parametric uncertainty, model validation and system identification for robust control, as well as several additional case studies.

2 The Data Structures

μ-Tools represents systems (either in state-space form or as frequency dependent input/output data) as single data entries, providing all of the information about a system in a single MATLAB variable. In addition, the μ-Tools functions which return a single variable can be nested, allowing you to build complex operations out of a few nested operations.

2.1 SYSTEM Matrices

Standard systems $\left[\begin{array}{c|c} A & B \\ \hline C & D \end{array}\right]$ are represented in μ-Tools by a single MATLAB data structure, referred to as a SYSTEM matrix. The actual format of the data storage is

$$\begin{bmatrix} A & B & nx \\ C & D & 0 \\ 0 & -Inf \end{bmatrix},$$

where the zeros correspond to a row or column of zeros. The `-Inf` in the lower right corner indicates to the μ-Tools functions that the matrix is a SYSTEM matrix. The number of states, `nx`, uniquely determines the partitioning of the rest of the data.

2.2 VARYING Matrices

Matrix-valued functions of a single, independent real variable are represented in μ-Tools with a data structure called a VARYING matrix. Frequency response and time responses of systems are the most common use for this type of structure, but other uses are possible. For example, a system which depended on a parameter could be represented as a VARYING matrix with the parameter as the independent variable and the matrix-valued function actually being a SYSTEM. Similarly, one can have VARYING matrices of VARYING matrices, and so on.

Consider a matrix valued function of a single real variable $G : \mathbf{R} \to \mathbf{C}^{n \times m}$, evaluated at N discrete values of $x \in \mathbf{R}$, call them $x_1, x_2, \ldots, x_N$. Let $G_i \in \mathbf{C}^{n \times m}$ be defined as $G(x_i)$. Then, in μ-Tools, the actual data representation of the function G is a MATLAB matrix with $n \cdot N + 1$ rows, and $m + 1$ columns, as shown below:

*Dept. of Aerospace Engineering and Mechanics, University of Minnesota
†Dept. of Mechanical Engineering, University of California, Berkeley
‡Dept. of Electrical Engineering, California Institute of Technology
§Dept. of Engineering, Cambridge University, England
¶Dept. of Electrical Engineering, University of California, Santa Barbara

$$\begin{bmatrix} [G_1] & x_1 \\ \vdots & \vdots \\ [G_i] & x_N \\ \vdots & 0 \\ & \vdots \\ [G_N] & 0 \\ & \vdots \\ 0 \cdot 0 \;\; N & \texttt{Inf} \end{bmatrix}$$

The `Inf` in lower right corner tags this as a **VARYING** matrix. The number just to the left of the `Inf`, in this example, N, indicates how many data points are represented, and then the first N values in the rightmost column are the independent variables values. The function data consists of the matrices, which are stacked one on top of one another.

2.3 CONSTANT Matrices

If a MATLAB entity is neither a SYSTEM nor a VARYING matrix it is treated by μ-**Tools** as a CONSTANT matrix. CONSTANT matrices can be arguments to functions which normally expect VARYING or SYSTEM matrix arguments.

3 Plotting VARYING Matrices

The function `vplot` plots VARYING matrices. The arguments for `vplot` are similar to MATLAB's `plot` command, with the exception that it is not necessary to specify the values for the x-axis. The x-axis data corresponds to the independent variables, which are already stored within each VARYING matrix. An important feature of `vplot` is its ability to plot multiple VARYING matrices on the same plot without having to have the same independent variables.

In the MATLAB `plot` command, different axis types are provided by different functions, `loglog`, `semilogx`, and others. `vplot` provides this capability by an optional string argument. The default is a linear/linear scale. The generic `vplot` function call looks like

```
vplot('axistype',vmat1,'linetype',vmat2,...).
```

The `axistype` argument, a character string, allows the specification of logarithmic or linear axes as well as: magnitude, log magnitude, and phase. There are also some control specific options: `bode`, `nyq`, and `nic` which specify Bode, Nyquist, and Nichols plots respectively. The `linetype` arguments are optional and are identical to those provided by MATLAB for the `plot` command.

4 Commands Grouped by Function

Standard Operations/ Basic Functions	
`abv`	Stack constant/varying/system matrices above one another
`cjt`	Conjugate transpose of varying/system matrices
`crand`	Complex random matrix generator
`daug`	Diagonal augmentation of constant/varying/system matrices
`madd`	Addition of constant/varying/system matrices
`minv`	Inverse of constant/varying/system matrices
`mmult`	Multiplication of constant/varying/system matrices
`mscl`	Scale (by a scalar) a system or varying matrix
`msub`	Subtraction of constant/varying/system matrices
`sbs`	Stack matrices next to one another
`sclin`	Scale system input
`sclout`	Scale system output
`sel`	Select rows/columns or outputs/inputs
`starp`	Redheffer star product
`transp`	Transpose of varying/system matrices

Matrix Information, Display and Plotting	
`drawmag`	Interactive moused-based sketch and fitting tool
`minfo`	Information on a matrix
`mprintf`	Formatted printing of a matrix
`rifd`	Display real, imaginary, frequency and damping data
`see`	Display varying/system matrices
`seeiv`	Display independent variables of a varying matrix
`seesys`	Formatted varying/system display
`vplot`	Plot a varying matrix
`vzoom`	Mouse driven axis selection of plot window

Modeling Functions	
`nd2sys`	Convert a SISO transfer function into a system matrix
`pck`	Create a system from `(A, B, C, D)`
`pss2sys`	Convert `[A B;C D]` into a μ-**Tools** system matrix
`sys2pss`	Extract `[A B; C D]` from a system
`sysic`	System interconnection program
`unpck`	Extract state-space data `(A,B,C,D)` from a system
`zp2sys`	Convert poles and zeros to a system matrix

System Matrix Functions	
`reordsys`	Reorder states in a system matrix
`samhld`	Sample-hold approximation of a continuous system
`spoles`	Poles of system matrix
`statecc`	Apply a coordinate transformation
`strans`	Bidiagonal coordinate transformation
`sysrand`	Generate a random system matrix
`szeros`	Transmission zeros of a system matrix
`tustin`	Prewarped continuous to discrete transformation

Model Reduction Functions	
`hankmr`	Optimal Hankel norm approximation of a system
`sdecomp`	Decompose a system matrix into two system matrices
`sfrwtbal`	Frequency weighted balanced realization of a system matrix
`sfrwtbld`	Stable frequency weighted realization of a system matrix
`sncfbal`	Balanced realization of coprime factors of a system matrix
`srelbal`	Stochastic balanced realization of a system matrix
`sresid`	Residualize states of a system matrix
`strunc`	Truncate states of a system matrix
`sysbal`	Balanced realization of a system matrix

System Response Functions	
`cos_tr`	Generate a cosine signal as a varying matrix
`dtrsp`	Discrete time response of a linear system
`frsp`	Frequency response of a system matrix
`sin_tr`	Generate a sine signal as a varying matrix
`siggen`	Generate a signal as a varying matrix
`step_tr`	Generate a step signal as a varying matrix
`trsp`	Time response of a linear system

H_2 and H_∞ Analysis and Synthesis Functions	
`csord`	Order complex Schur form matrices
`h2norm`	Calculate 2-norm of a stable, strictly proper system
`h2syn`	H_2 control design
`hinffi`	H_∞ full information control design
`hinfnorm`	Calculate ∞-norm of a stable, proper system
`hinfsyn`	H_∞ control design
`pkvnorm`	Peak norm of varying matrix
`ric_eig`	Solve a Riccati equation via eigenvalue decomposition
`ric_schr`	Solve a Riccati equation via real Schur decomposition

Structured Singular Value (μ) Analysis and Synthesis	
blknorm	Block norm of constant/varying matrices
dypert	Create a rational perturbation from mu data
fitmag	Fit magnitude data with rational transfer function
fitmaglp	Fit magnitude data with rational transfer function
fitsys	Fit frequency response data with transfer function
genphase	Generate a minimum phase frequency response to magnitude data
magfit	Fit magnitude data with rational transfer function (a batch process)
mu	μ-analysis of constant/varying matrices
muftbtch	Batch D scaling rational fit routine
musynfit	Interactive D scaling rational fit routine
musynflp	Interactive D scaling rational fit routine
randel	Generate a random perturbation
sisorat	Fit a frequency point with first order, all-pass, stable transfer function
unwrapd	Construct D scaling from mu
unwrapp	Construct Δ perturbation from mu

Varying Matrix Manipulation	
getiv	Get the independent variable of a varying matrix
indvcmp	Compare the independent variable data
negangle	Calculate angle of elements between 0 and -2π
scliv	Scale the independent variable
sortiv	Sort the independent variable
tackon	String together varying matrices
var2con	Convert a varying matrix to a constant matrix
varyrand	Generate a random varying matrix
vdcmate	Decimate varying matrices
vinterp	Interpolate varying matrices
vpck	Pack a varying matrix
vunpck	Unpack a varying matrix
xtract	Extract portions of a varying matrix
xtracti	Extract portions of a varying matrix

Many of the MATLAB matrix functions have analogous μ-Tools functions which are identical on CONSTANT matrices, but operate on VARYING matrices on a matrix by matrix basis. These functions are

```
vabs   vceil   vdet   vdiag   veig    veval
vexpm  vfloor  vinv   vimag   vnorm   vpinv
vpoly  vrcond  vreal  vroots  vschur  vsvd
```

Additional μ-Tools functions which extend MATLAB functionality to VARYING matrices are

```
vconj  vcjt  vebe  vfft  vifft  vldiv  vrdiv  vrho  vtp
```

The functions veval and vebe perform a named operation on VARYING matrices. vebe performs MATLAB or user defined functions on the elements of a VARYING matrix (for example: sin, tan ...). veval can perform any function including those with multiple input and output arguments. vebe and veval allow the evaluation of any MATLAB matrix function on VARYING matrices. Several of these functions are illustrated below.

MATLAB Matrix Function	μ-Tools VARYING Function
A + B + ... + H	madd(A,B,...,H)
A / B ; A \ B	vrdiv(A,B) ; vldiv(A,B)
A .* B	veval('.*',A,B)
A ^ b	veval('^',A,b)
A'	cjt(A) (or vcjt(A))
sin(A)	vebe('sin',A)
max(abs(eig(A))	vrho(A)

4.1 Special features

μ-Tools has several features that help make it user friendly. In addition to standard MATLAB help facilities, whenever a command is typed without any arguments, or with an incorrect number of arguments, a usage line is returned. For example, if the user has forgotten the inputs to mu, simply type

```
>> mu
usage: [bnds,row_d,sens,row_p] = mu(matin,blk,opt)
```

For creating systems, in addition to nd2sys, zp2sys, sysic, and other programs that take command line arguments, drawmag is an interactive mouse-based sketch and fitting tool. drawmag creates a VARYING file of points at the locations where the mouse is clicked and a stable, minimum-phase system which approximately fits the points. This is particularly useful for creating weights for μ-synthesis and candidate loopshapes for loopshaping designs.

The μ-Tools commands for handling VARYING matrices are particularly useful for studying parametrized systems. Suppose we want to design a controller for the system

$$p(s) = \frac{Ks}{Ts+1}$$

where $K \in [1,10]$ and $T \in [.1,1]$. We can easily form a VARYING matrix with the 4 extreme corner systems with the following commands:

```
>>p1 = nd2sys(1,[1 1]);
>>p2 = mscl(p1,10);
>>p4 = nd2sys(1,[.1 1]);
>>p3 = mscl(p4,10);
>>pstack = vpck([p1;p2;p3;p4;p1],[1;2;3;4;1]);
```

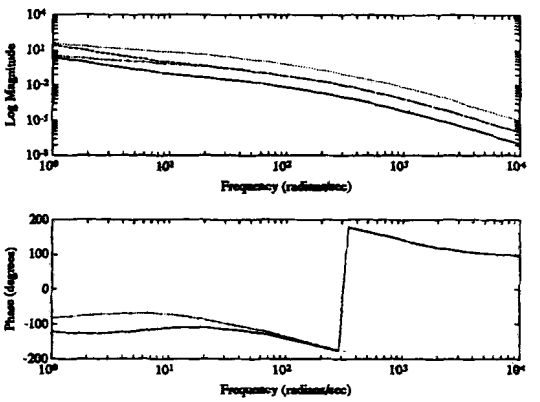
The VARYING matrix pstack contains 5 systems, with the last one a repeat of the first. This is done to provide good Nichols chart plots. The commands

```
>>c = mmult(nd2sys(1,[1 0]),nd2sys([1 4],[.01 1]),...
>>nd2sys(1,[1e-3 1]));
>>cstack = vpck([c;c;c;c;c],[1;2;3;4;1]);
>>lstack = veval('mmult',cstack,pstack);
```

form a 3rd order controller and multiply it by each plant to form a VARYING matrix of loop transfer functions. We can then take the frequency response and plot it in several ways.

```
>>lvar = veval('frsp',lstack,omega);
>>lnic = veval('vunpck',lvar);
>>lbode = veval('vunpck',swapiv(lvar,[2 1]));
>>lbode = sel(lbode,1:4,1);

>>vplot('nic',lnic); grid;
>>vplot('bode',lbode);
```



5 Robust Performance Design Example

This section contains an example of μ-synthesis as applied to a paper design for the linearized pitch axis controller of an experimental highly maneuverable airplane, using a model taken from data for the HIMAT vehicle. Because of space constraints, only minimal engineering motivation will be give for this problem; it is presented only to illustrate the software. The **μ-Tools software** comes with six M-files, **himat_x1** through **himat_x6**, which go through this example in greater detail. The problem is posed as a **robust performance problem**, with **multiplicative plant uncertainty** at the plant input and **plant output weighted sensitivity function** as the performance criterion. The design procedure involves several steps:

1. Specification of closed loop feedback structure.
2. Specification of model uncertainty and performance objectives in terms of frequency-dependent weighting matrices.
3. Construction of open-loop interconnection for control synthesis routines.
4. $\mathcal{H}_\infty$ optimal controller design for the open-loop interconnection.
5. Analysis of ROBUST PERFORMANCE properties of the resulting closed-loop systems using the structured singular value, μ (μ-analysis).
6. Use of frequency dependent similarity scalings, obtained in the μ-analysis step, to scale the open loop interconnection, and redesign $\mathcal{H}_\infty$ controller (iterating on steps 5, 6, and 7 constitutes an approach to μ-synthesis).

The state vector consists of the vehicle's basic rigid body variables:

$$x^T = (\delta v, \alpha, q, \theta)$$

representing the forward velocity, angle-of-attack, pitch rate, and pitch angle, respectively. The control inputs are the elevon (δ_e) and the canard (δ_c). The variables to be measured are α and θ.

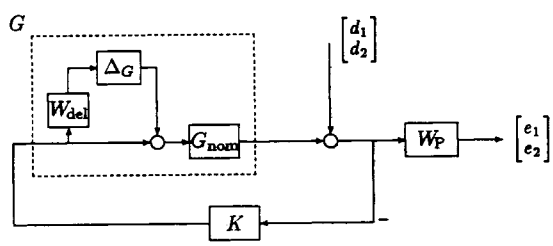


Figure 1: HIMAT Closed-loop Interconnection Structure

The control design objective is to design a stabilizing controller K such that for all stable perturbations $\Delta_G(s)$, with $\|\Delta_G\|_\infty < 1$, the perturbed closed-loop system remains stable, and the perturbed weighted sensitivity transfer function,

$$S(\Delta_G) := W_p(I + P(I + \Delta_G W_{del})K)^{-1}$$

has $\|S(\Delta_G)\|_\infty < 1$ for all such perturbations.

5.1 Models and performance objectives

Sources of uncertainty include uncertainty in the canard and the elevon actuators, in the forces and moments generated on the aircraft due to their deflections, uncertainty in the linear and angular accelerations produced by the aerodynamically generated forces and moments, and many others. In this example, we choose not to model the uncertainty in a detailed manner, but rather to lump all of these effects together into 1 **full-block uncertainty** at the input of a 4-state, nominal model of the aircraft rigid body. This nominal model has no (i.e., perfect) actuators and only **quasi-steady dynamics**. The nominal model for the airplane is loaded from the **mutools/subs** directory.

```
>> mkhimat;
>> minfo(himat)
>> seesys(himat,'%9.1e')
-2.3e-02 -3.7e+01 -1.9e+01 -3.2e+01  |  0.0e+00  0.0e+00
 0.0e+00 -1.9e+00  9.8e-01  0.0e+00  | -4.1e-01  0.0e+00
 1.2e-02 -1.2e+01 -2.6e+00  0.0e+00  | -7.8e+01  2.2e+01
 0.0e+00  0.0e+00  1.0e+00  0.0e+00  |  0.0e+00  0.0e+00
 ------------------------------------|------------------
 0.0e+00  5.7e+01  0.0e+00  0.0e+00  |  0.0e+00  0.0e+00
 0.0e+00  0.0e+00  0.0e+00  5.7e+01  |  0.0e+00  0.0e+00
```

The partitioned matrix represents the **[A B; C D]** state space data For this example, $W_{del} := w_{del}(s)I_2$, with $w_{del}(s) = \frac{50(s+100)}{s+10000}$, and $W_p(s) = w_p(s)I_2$, where $w_p(s) = \frac{0.5(s+3)}{s+0.03}$. The engineering motivation for a performance specification like this would most naturally come from the desire to be able to have independent tracking of the angle of attack and pitch angle. W_{del} can be formed as follows:

```
>> wdel = nd2sys([1 100],[1 10000],50);
>> wdel = daug(wdel,wdel);
>> wp = nd2sys([0.5,1.5],[1,0.03]); wp = daug(wp,wp);
```

The phrases **robust stability**, **nominal performance**, and **robust performance** are used in this framework extensively. For this problem, they mean:

Nominal Performance:

$$\left\|W_p(I + G_{nom}K)^{-1}\right\|_\infty < 1$$

Robust Stability:

$$\left\|W_{del}KG_{nom}(I + KG_{nom})^{-1}\right\|_\infty < 1$$

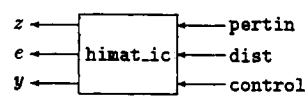
Robust Performance:

$$\left\|W_p(I + GK)^{-1}\right\|_\infty < 1$$

is satisfied for every $G \in \mathcal{G}$. The property of **robust performance** is equivalent to a **structured singular value** test.

5.2 Building the open-loop interconnection with sysic

A 6-input, 6-output SYSTEM matrix, **himat_ic**, (also referred to as $P(s)$)



has internal structure shown in Figure 2.

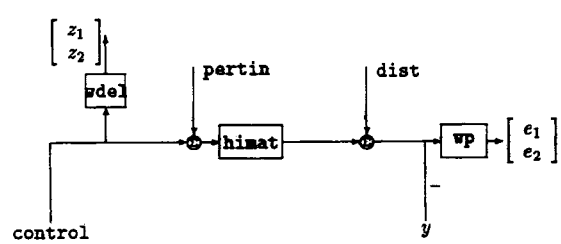


Figure 2: HIMAT Open-loop Interconnection Structure

This can be produced with nine MATLAB commands, listed below. The first 8 lines describe the various aspects of the interconnection,

and may appear in any order. The last command, **sysic**, produces the final interconnection. The commands can be placed in an M-file, or executed at the command line.

```
≫ systemnames = ' himat wp wdel ';
≫ inputvar = '[ pertin(2) ; dist(2) ; control(2) ]';
≫ outputvar = '[ wdel ; wp ; -himat - dist ]';
≫ input_to_himat = '[ control + pertin ]';
≫ input_to_wdel = '[ control ]';
≫ input_to_wp = '[ himat + dist ]';
≫ sysoutname = 'himat_ic';
≫ cleanupsysic = 'yes';
≫ sysic;
```

5.3 μ-synthesis and $D-K$ iteration

For notational purposes, let $P(s)$ denote the transfer function of the six-input, six-output open-loop interconnection, **himat_ic**. Define a block structure $\mathbf{\Delta}$ as

$$\mathbf{\Delta} := \left\{ \begin{bmatrix} \Delta_1 & 0 \\ 0 & \Delta_2 \end{bmatrix} : \Delta_1 \in \mathbf{C}^{2\times 2}, \Delta_2 \in \mathbf{C}^{2\times 2} \right\} \subset \mathbf{C}^{4\times 4}.$$

The first block of this structured set corresponds to the full-block uncertainty Δ_G used in section 5.1 to model the uncertainty in the airplane's behavior. The second block, Δ_2 is a fictitious uncertainty block, used to incorporate the $\mathcal{H}_\infty$ performance objectives on the weighted output sensitivity transfer function into the μ-framework.

Recall that a stabilizing controller K achieves closed-loop, **robust performance** if and only if for each frequency $\omega \in [0, \infty]$, the structured singular value

$$\mu_{\mathbf{\Delta}} \left[F_l(P,K)(j\omega) \right] < 1$$

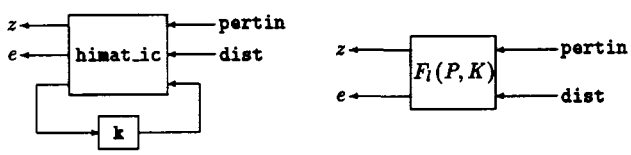
Using the upper bound for μ, (recall that in this case, 2 full blocks, the upper bound is exactly equal to μ) we can attempt to minimize the peak closed-loop μ value by posing the optimization problem

$$\min_{\substack{K \, stabilizing \\ \hat{d}(s) minphase, stable}} \left\| \begin{bmatrix} \hat{d}(s) I_2 & 0 \\ 0 & I_2 \end{bmatrix} F_l(P,K) \begin{bmatrix} \hat{d}^{-1}(s) I_2 & 0 \\ 0 & I_2 \end{bmatrix} \right\|_\infty .$$

An approximate μ-synthesis involves a sequence of minimizations, first over the controller variable K (holding the d variable fixed), and then over the d variable (holding the K variable fixed). This is often referred to as the $D-K$ iteration.

5.4 $\mathcal{H}_\infty$ design on the open-loop interconnection

In this section, we carry out the first step of the $D-K$ iteration, which is an $\mathcal{H}_\infty$ (sub)optimal control design for the open-loop interconnection, **himat_ic**. In terms of the iteration, this amounts to holding the d variable fixed (at 1), and minimizing the $\| \cdot \|_\infty$ norm of $F_l(P,K)$, over the controller variable K, as shown.



The function **hinfsyn** designs a (sub)optimal $\mathcal{H}_\infty$ control law based on the open-loop interconnection structure provided. Syntax, input and output arguments to **hinfsyn** are:

```
≫ [k,clp] = hinfsyn(p,nmeas,ncon,glow,ghigh,tol);
```

The arguments are:

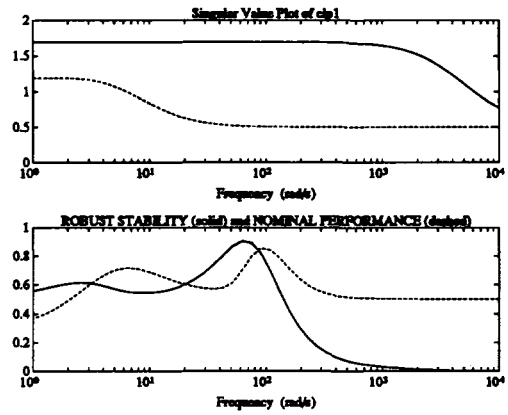
Inputs		
	open-loop interconnection (SYSTEM matrix)	p
	number of measurements	nmeas
	number of controls	ncons
	lower bound on achievable norm	glow
	upper bound on achievable norm	ghigh
	absolute tolerance for bisection method	tol
Outputs		
	controller (SYSTEM matrix)	k
	closed-loop (SYSTEM matrix)	clp

In this example, the calling sequence is

```
≫ [k1,clp1] = hinfsyn(himat_ic,2,2,0.8,6.0,.06);
≫ clp1g = frsp(clp1,omega);
≫ clp1gs = vsvd(clp1g);
≫ subplot(211),vplot('liv,m',clp1gs)
≫ title('Singular Value Plot of clp1')
≫ xlabel('Frequency (rad/s)')
```

The two 2×2 transfer functions associated with **robust stability** and **nominal performance** may be evaluated separately, using the command **sel**.

```
≫ rob_stab = sel(clp1g,[1 2],[1 2]);
≫ nom_perf = sel(clp1g,[3 4],[3 4]);
≫ subplot(212)
≫ vplot('liv,m',vnorm(rob_stab),vnorm(nom_perf))
≫ tmp1 = 'ROBUST STABILITY (solid) and';
≫ tmp2 = ' NOMINAL PERFORMANCE (dashed)';
≫ title([tmp1 tmp2])
≫ xlabel('Frequency (rad/s)')
```



Note that the controlled system achieves both *nominal performance* and *robust stability*, since the peaks of the relevant singular value plots are less than 1.

5.5 Assessing ROBUST PERFORMANCE with μ

The ROBUST PERFORMANCE properties of the closed-loop system can be analyzed using μ-analysis. For a frequency domain μ-analysis of ROBUST PERFORMANCE properties, the block structure should consist of a 2×2 uncertainty block, and a 2×2 performance block. We'll now make this calculation on the closed-loop from the $\mathcal{H}_\infty$ design. The density of the frequency points is increased from 50 to 100 in order to have smoother plots.

```
≫ blk=[2 2 ; 2 2];
≫ omega1 = logspace(-1,4,100);
≫ clp_g1 = frsp(clp1,omega1);
≫ [bnds1,dvec1,sens1,pvec1] = mu(clp_g1,blk);
≫ vplot('liv,m',vnorm(clp_g1),bnds1)
≫ title('Maximum Singular Value and mu Plot')
≫ xlabel('Frequency (rad/s)')
≫ text(.15,.9,'max singular value (solid)','sc')
≫ text(.3,.4,'mu bounds (dashed)','sc')
≫ text(.2,.15,'H-infinity Controller','sc')
```

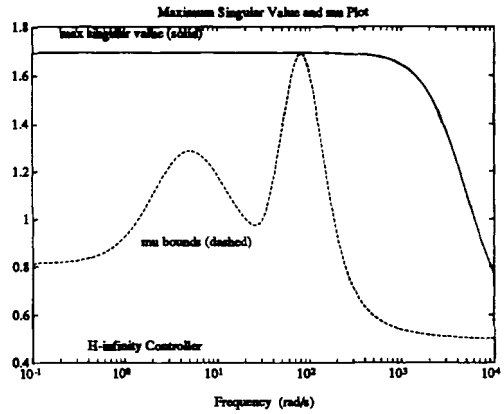


Figure 3: The dashed plot (which is actually two plots) is both the upper and lower bounds for $\mu_\Delta(F_l(P,K)(j\omega))$.

Hence, the controlled system (from $\mathcal{H}_\infty$) does not achieve robust performance. This conclusion follows from the μ plot, which peaks to a value of 1.69, at a frequency of 73.6 rad/s. Since the upper and lower bounds are equal here, this is exact. The worst-case perturbattion can be constructed using dypert.

5.6 One+ iteration of μ-synthesis

Designing an $\mathcal{H}_\infty$ control law was the first computational step in μ-synthesis. The second step involved a μ analysis on the closed-loop system. This calculation produces frequency dependent scaling matrices, called the D-scales. Using `musynfit`, the varying variables in the D-scales can be fit (in magnitude) with stable, minimum phase rational functions and absorbed into the generalized plant for additional iterations. The syntax for `musynfit` is

```
≫ [dsysL,dsysR] = ...
 musynfit(predsysL,Ddata,sens,blk,nmeas,ncont);
```

`predsysL`: the rational D scaling matrix from the previous iteration.

`Ddata`: the frequency varying D-scales from the previous μ-analysis step. These are the "new" D's which need to be absorbed onto the existing scalings.

`sens`: the sensitivity variable from the previous μ analysis step.

`blk`: the uncertainty block structure, same as from the μ-analysis.

`nmeas`: number of measurements, same as in the $\mathcal{H}_\infty$ design.

ncon: number of controls, same as in the $\mathcal{H}_\infty$ design.

The output arguments are two system matrices, `dsysL` and `dsysR`. For the most part, these are the same thing - they are stable, minimum phase, block diagonal approximations to the product of the previous rational D, and the frequency varying D (from μ calculation) which was the adjustment that was to be made to the rational D. The new interconnection structure is formed from the original unscaled `himat_ic` together with `dsysL` and `dsysR`.

```
≫ new_ic = mmult(dsysL,himat_ic,minv(dsysR));
```

`musynfit` runs interactively using the graphics window, but there is batch version as well. For the first iteration a 4th order fit was used. This increases the number of states in the open-loop interconnection structure by 16 states.

```
≫ [dsysL1,dsysR1] = musynfit('first',dvec1,sens1,blk,2,2);
≫ muic1 = mmult(dsysL1,himat_ic,minv(dsysR1));
```

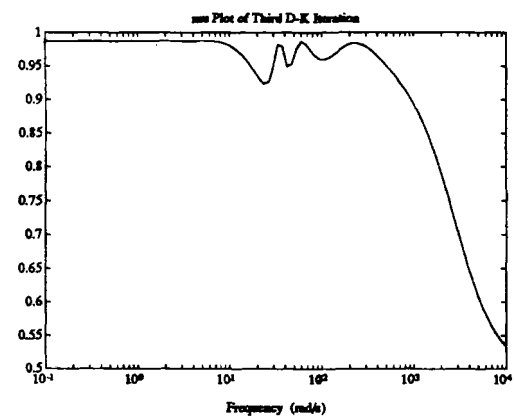
This rational D fitting concludes the first iteration of the $D-K$ iteration approach to μ synthesis. The process is repeated, beginning with a new $\mathcal{H}_\infty$ design on the generalized plant `muic1`. This open-loop interconnection, `muic1`, consists of the rational D-scalings `dsysL1` and `dsysR1` absorbed onto the original open-loop interconnection, `himat_ic`.

The second iteration of $D-K$ iteration is comprised of the following steps listed below. A third order fit was used for the D-scaling.

```
≫ [k2,clp2] = hinfsyn(muic1,2,2,0.8,1.8,0.03);
≫ clp_g2 = frsp(clp2,omega);
≫ [bnds2,dvec2,sens2,pvec2] = mu(clp_g2,blk);
≫ [dsysL2,dsysR2] = musynfit(dsysL1,dvec2,sens2,blk,2,2);
≫ muic2 = mmult(dsysL2,himat_ic,minv(dsysR2));
```

The third iteration of $D-K$ iteration is:

```
≫ [k3,clp3] = hinfsyn(muic2,2,2,0.8,1.1,0.03);
≫ clp_g3 = frsp(clp3,omega);
≫ [bnds3,dvec3,sens3,pvec3] = mu(clp_g3,blk);
≫ vplot('liv,d',bnds3)
≫ title('mu Plot of Third D-K Iteration')
≫ xlabel('Frequency (rad/s)')
```



On the third iteration, the value of μ was less than 1 across frequency. This implies that *robust performance* was achieved for the closed loop system with controller `k3`. Following the steps above, one can carry on as many iterations as desired. In general, the $D-K$ iteration process is continued until either μ is less than 1, indicating the robust performance has been achieved, or the peak μ value of the closed-loop system stops decreasing.

6 Summary

This paper provides a brief description of *The μ Analysis and Synthesis Toolbox* (μ-Tools), an advanced control design toolbox to be used in conjunction with MATLAB.