Model Predictive Control Toolbox

For Use with MATLAB®

Manfred Morari
N. Lawrence Ricker

Computation

Visualization

Programming

User's Guide

Version 1

The MathWorks

How to Contact The MathWorks:

`www.mathworks.com`	Web
`comp.soft-sys.matlab`	Newsgroup
`support@mathworks.com`	Technical support
`suggest@mathworks.com`	Product enhancement suggestions
`bugs@mathworks.com`	Bug reports
`doc@mathworks.com`	Documentation error reports
`service@mathworks.com`	Order status, license renewals, passcodes
`info@mathworks.com`	Sales, pricing, and general information
508-647-7000	Phone
508-647-7001	Fax
The MathWorks, Inc. 3 Apple Hill Drive Natick, MA 01760-2098	Mail

For contact information about worldwide offices, see the MathWorks Web site.

Model Predictive Control Toolbox User's Guide

Printing History:	January 1995	First printing
	October 1998	(Online only)

Contents

Preface

1 Tutorial

2 MPC Based on Step Response Models

3 MPC Based on State-Space Models

4 Command Reference

Index

Preface

Acknowledgments

The toolbox was developed in cooperation with: Douglas B. Raven and Alex Zheng

The contributions of the following people are acknowledged: Yaman Arkun, Nikolaos Bekiaris, Richard D. Braatz, Marc S. Gelormino, Evelio Hernandez, Tyler R. Holcomb, Iftikhar Huq, Sameer M. Jalnapurkar, Jay H. Lee, Yusha Liu, Simone L. Oliveira, Argimiro R. Secchi, and Shwu-Yien Yang

We would like to thank Liz Callanan, Jim Tung and Wes Wang from the MathWorks for assisting us with the project, and Patricia New who did such an excellent job putting the manuscript into LATEX.

About the Authors

Manfred Morari

Manfred Morari received his diploma from ETH Zurich in 1974 and his Ph.D. from the University of Minnesota in 1977, both in chemical engineering. Currently he is the McCollum-Corcoran Professor and Executive Officer for Control and Dynamical Systems at the California Institute of Technology. Morari's research interests are in the areas of process control and design. In recognition of his numerous contributions, he has received the Donald P. Eckman Award of the Automatic Control Council, the Allan P. Colburn Award of the AIChE, the Curtis W. McGraw Research Award of the ASEE, was a Case Visiting Scholar, the Gulf Visiting Professor at Carnegie Mellon University and was recently elected to the National Academy of Engineering. Dr. Morari has held appointments with Exxon R&E and ICI and has consulted internationally for a number of major corporations. He has coauthored one book on Robust Process Control with another on Model Predictive Control in preparation.

N. Lawrence Ricker

Larry Ricker received his B.S. degree from the University of Michigan in 1970, and his M.S. and Ph.D. degrees from the University of California, Berkeley, in 1972/78. All are in Chemical Engineering. He is currently Professor of Chemical Engineering at the University of Washington, Seattle. Dr. Ricker has over 80 publications in the general area of chemical plant design and operation. He has been active in Model Predictive Control research and teaching for more than a decade. For example, he published one of the first nonproprietary studies of the application of MPC to an industrial process, and is currently involved in a large-scale MPC application involving more than 40 decision variables.

1

Tutorial

Introduction

The Model Predictive Control (MPC) Toolbox is a collection of functions (commands) developed for the analysis and design of model predictive control (MPC) systems. Model predictive control was conceived in the 1970s primarily by industry. Its popularity steadily increased throughout the 1980s. At present, there is little doubt that it is the most widely used multivariable control algorithm in the chemical process industries and in other areas. While MPC is suitable for almost any kind of problem, it displays its main strength when applied to problems with:

- A large number of manipulated and controlled variables
- Constraints imposed on both the manipulated and controlled variables
- Changing control objectives and/or equipment (sensor/actuator) failure
- Time delays

Some of the popular names associated with model predictive control are Dynamic Matrix Control (DMC), IDCOM, model algorithmic control, etc. While these algorithms differ in certain details, the main ideas behind them are very similar. Indeed, in its basic unconstrained form MPC is closely related to linear quadratic optimal control. In the constrained case, however, MPC leads to an optimization problem which is solved on-line in real time at each sampling interval. MPC takes full advantage of the power available in today's control computer hardware.

This software and the accompanying manual are not intended to teach the user the basic ideas behind MPC. Background material is available in standard textbooks like those authored by Seborg, Edgar and Mellichamp (1989)[1], Deshpande and Ash (1988)[2] and the monograph devoted solely to this topic authored by Morari and coworkers (Morari et al., 1994)[3]. This section provides a basic introduction to the main ideas behind MPC and the specific form of implementation chosen for this toolbox. The algorithms used here are consistent with those described in the monograph by Morari et al. Indeed, the software is meant to accompany the monograph and vice versa. The routines included in the MPC Toolbox fall into two basic categories: routines which use

1. D.E. Seborg, T.F. Edgar, D.A. Mellichamp; *Process Dynamics and Control;* JohnWiley & Sons, 1989
2. P.B. Deshpande, R.H. Ash; *Computer Process Control with Advanced Control Applications*, 2nd ed., ISA, 1988
3. M. Morari, C.E. Garcia, J.H. Lee, D.M. Prett; *Model Predictive Control*; Prentice Hall, 1994 (In the process of being written.)

a step response model description and routines which use a state-space model description. In addition, simple identification tools are provided for identifying step response models from plant data. Finally, there are also various conversion routines which convert between different model formats and analysis routines which can aid in determining the stability of the unconstrained system, etc. All MPC Toolbox commands have a built-in usage display. Any command called with no input arguments results in a brief description of the command line. For example, typing `mpccon` at the command line gives the following:

```
usage: Kmpc = mpccon(model,ywt,uwt,M,P)
```

The following sections include several examples. They are available in the tutorial programs `mpctut.m`, `mpctutid.m`, `mpctutst.m`, and `mpctutss.m`. You can copy these demo files from the `mpctools/mpcdemos` source into a local directory and examine the effects of modifying some of the commands.

Target Audience for the MPC Toolbox

The package is intended for the classroom and for the practicing engineer. It can assist in communicating the concepts of MPC to a student in an introductory control course. At the same time it is sophisticated enough to allow an engineer in industry to evaluate alternate control strategies in simulation studies.

System Requirements

The MPC Toolbox assumes the following operating system requirements:

- MATLAB® is running on your system.
- If nonlinear systems are to be simulated, Simulink® is required for the functions `nlcmpc` and `nlmpcsim`.
- If the *theta* format from the System Identification Toolbox is to be used to create models in the MPC *mod* format (using the MPC Toolbox function, `th2mod`), then the System Identification Toolbox function `polyform` and the Control System Toolbox function `append` are required.

The MPC Toolbox analysis and simulation algorithms are numerically intensive and require approximately 1MB of memory, depending on the number of inputs and outputs. The available memory on your computer may limit the size of the systems handled by the MPC Toolbox.

Note: there is a pack command in MATLAB that can help free memory space by compacting fragmented memory locations. For reasonable response times, a computer with power equivalent to an 80386 machine is recommended unless only simple tutorial example problems are of interest.

2

MPC Based on Step Response Models

Step Response Models

Step response models are based on the following idea. Assume that the system is at rest. For a linear time-invariant single-input single-output (SISO) system let the output change for a unit input change Δv be given by

$$\{0, s_1, s_2, \ldots, s_n, s_n, \ldots\}$$

Here we assume that the system settles exactly after n steps. The step response $\{s_1, s_2, \ldots, s_n\}$ constitutes a complete model of the system, which allows us to compute the system output for any input sequence:

$$y(k) = \sum_{i=1}^{n} s_i \Delta v(k-i) + s_n v(k-n-1)$$

Step response models can be used for both stable and integrating processes. For an integrating process it is assumed that the slope of the response remains constant after n steps, i.e.,

$$s_n - s_{n-1} = s_{n+1} - s_n = s_{n+2} - s_{n+1} = \ldots$$

For a multi-input multi-output (MIMO) process with n_v inputs and n_y outputs, one obtains a series of step response coefficient matrices

$$S_i = \begin{bmatrix} s_{1,1,i} & s_{1,2,i} & \cdots & s_{1,n_v,i} \\ s_{2,1,i} & & & \\ \vdots & & & \\ s_{n_y,1,i} & s_{n_y,2,i} & \cdots & s_{n_y,n_v,i} \end{bmatrix}$$

where $s_{l,m,i}$ is the i^{th} step response coefficient relating the m^{th} input to the l^{th} output.

The MPC Toolbox stores step response models in the following format:

$$
\text{plant} = \begin{bmatrix} & S_1 & & & \\ & S_2 & & & \\ & \vdots & & & \\ & S_n & & & \\ \text{nout}(1) & 0 & \dots & 0 & \\ \text{nout}(2) & 0 & \dots & 0 & \\ \vdots & \vdots & & & \vdots \\ \text{nout}(\text{n}_\text{y}) & 0 & \dots & 0 & \\ n_y & 0 & \dots & 0 & \\ \text{delt2} & 0 & \dots & 0 & \end{bmatrix}_{(n \cdot n_y + n_y + 2) \times n_v}
$$

where `delt2` is the sampling time and the vector `nout` indicates if a particular output is integrating or not:

`nout(i) = 0` if output `i` is integrating.
`nout(i) = 1` if output `i` is stable.

The step response can be obtained directly from identification experiments, or generated from a continuous or discrete transfer function or state-space model. For example, if the discrete system description (sampling time $T = 0.1$) is

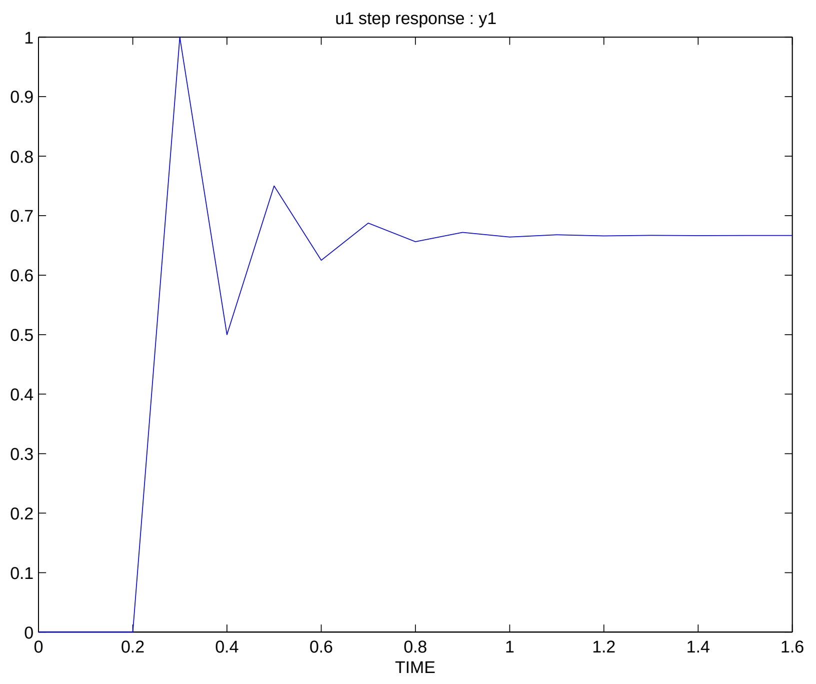
$$y(k) = -0.5y(k-1) + v(k-3)$$

then the transfer function is

$$g(z) = \frac{z^{-3}}{1 + 0.5z^{-1}}$$

The following commands (see mpctut.m) generate the step response model for this system and plot it:

```
num = 1;
den = [1 0.5];
delt1 = 0.1;
delay = 2;
g = poly2tfd(num,den,delt1,delay);
% Set up the model in tf format
tfinal = 1.6;
delt2 = delt1;
nout = 1;
plant = tfd2step(tfinal,delt2,nout,g);
% Calculate the step response
plotstep(plant) % Plot the step response
```



Alternatively, we could first generate a state-space description applying the command `tf2ss` and then generate the step response with `ss2step`. In this case, we need to pad the numerator and denominator polynomials to account for the time delay.

```
num = [0 0 0 num];
den = [den 0 0];
[phi,gam,c,d] = tf2ss(num,den); % Convert to state-space
plant = ss2step(phi,gam,c,d,tfinal,delt1,delt2,nout);
% Calculate step response
```

We can get some information on the contents of a matrix in the MPC Toolbox via the command `mpcinfo`. For our example, `mpcinfo(plant)` returns:

```
This is a matrix in MPC Step format.
   sampling time = 0.1
   number of inputs = 1
   number of outputs = 1
number of step response coefficients = 16
All outputs are stable.
```

Model Identification

The identification routines available in the MPC Toolbox are designed for multi-input single-output (MISO) systems. Based on a historical record of the output $y_l(k)$ and the inputs $v_1(k)$; $v_2(k)$, . . . , $v_{n_v}(k)$,

$$yy_l = \begin{bmatrix} y_l(1) \\ y_l(2) \\ y_l(3) \\ \vdots \end{bmatrix} \qquad v = \begin{bmatrix} v_1(1) & v_2(1) & \ldots & v_{n_v}(1) \\ v_1(2) & v_2(2) & \ldots & v_{n_v}(2) \\ v_1(3) & v_2(3) & \ldots & v_{n_v}(3) \\ \vdots & \vdots & & \end{bmatrix}$$

the step response coefficients

$$\begin{bmatrix} s_{l,1,1} & s_{l,2,1} & \cdots & s_{l,n_v,1} \\ s_{l,1,2} & s_{l,2,2} & \cdots & s_{l,n_v,2} \\ \vdots & & & \\ s_{l,1,i} & s_{l,2,i} & \cdots & s_{l,n_v,i} \\ \vdots & \vdots & & \vdots \end{bmatrix}$$

are estimated. For the estimation of the step response coefficients we write the SISO model in the form

$$\Delta y(k) = \sum_{i=1}^{n} h_i \Delta v(k-i)$$

where

$$\Delta y(k) = y(k) - y(k-1)$$

and

$$h_i = s_i - s_{i-1}$$

h_i are the impulse response coefficients. This model allows the designer to present all the input (v) and output(y_l) information in deviation form, which is often desirable. If the particular output is integrating, then the model

$$\Delta(\Delta y(k)) = \sum_{i=1}^{n} \Delta h_i \Delta v(k-i)$$

where

$$\Delta(\Delta y(k)) = \Delta y(k) - \Delta y(k-1)$$

$$\Delta h_i = h_i - h_{i-1}$$

should be used to estimate Δh_i, and thus h_i and s_i are given by

$$h_i = \sum_{k=1}^{i} \Delta h_k$$

$$s_i = \sum_{j=1}^{i} h_j = \sum_{j=1}^{i} \sum_{k=1}^{j} \Delta h_k$$

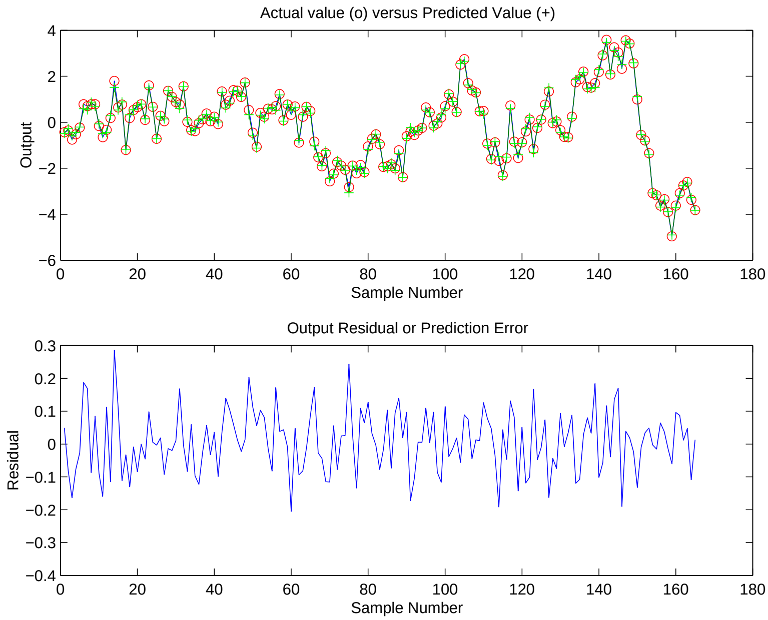
For parameter estimation it is usually recommended to scale all the variables such that they are the same order of magnitude. This may be done via the MPC Toolbox functions `autosc` or `scal`. Then the data has to be arranged into the form

$$Y = X\Theta$$

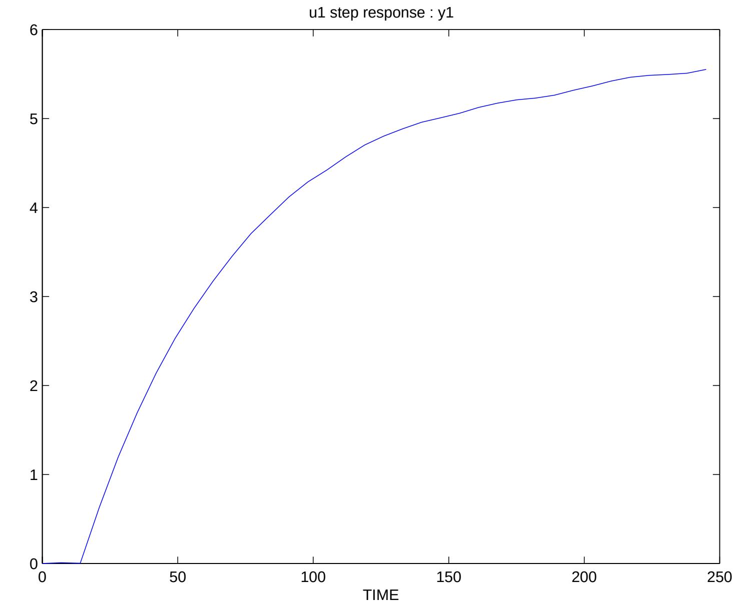
where Y contains all the output information ($\Delta y(k)$ for stable and $\Delta(\Delta y(k))$ for integrating outputs) and X all the input information ($\Delta v(k)$) appropriately arranged. Θ is a vector including all the parameters to be estimated (h_i for stable and Δh_i for integrating outputs). This rearrangement of the input and output information is handled by `wrtreg`. The parameters Θ can be estimated via multivariable least squares regression (`mlr`) or partial least squares (`plsr`). Finally, the step response model is generated from the impulse response coefficients via `imp2step`. The following example (see `mpctutid`) illustrates this procedure.

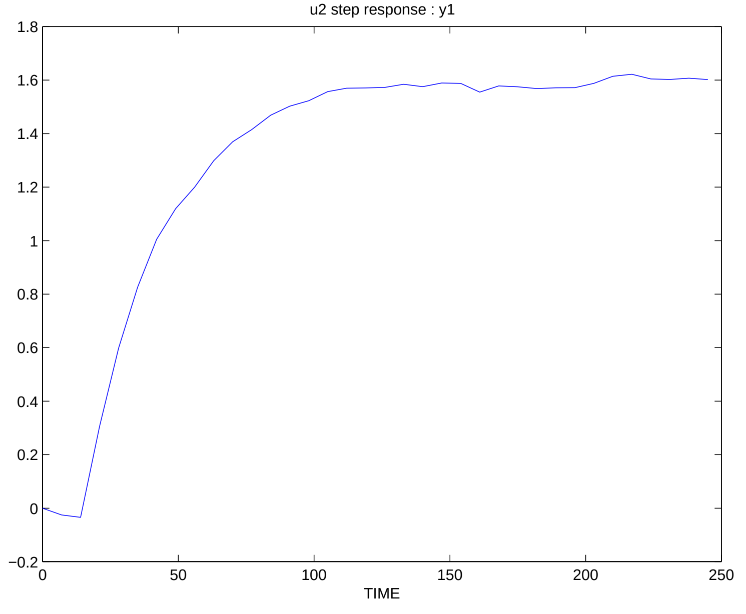
Example:

```
% Load the input and output data. The input and output
% data are generated from the following transfer
% functions and random zero-mean noises.
% TF from input 1 to output 1: g11=5.72exp(-14s)/
% (60s+1)
% TF from input 2 to output 1: g12=1.52exp(-15s)/
% (25s+1)
% Sampling time of 7 minutes was used.
%
% load mlrdat
%
% Determine the standard deviations for input data using
% autosc.
[ax,mx,stdx]=autosc(x);
%
% Scale the input data by their standard deviations only.
mx=0*mx;
sx=scal(x,mx,stdx);
%
% Put the input and output data in a form such that they
% can be used to determine the impulse response
% coefficients. 35 coefficients (n) are used.
n=35;
[xreg,yreg]=wrtreg(sx,y,n);
%
% Determine the impulse response coefficients via mlr.
% By specifying plotopt=2, two plots—plot of predicted
% output and actual output, and plot of the output
% residual (or prediction error)—are produced.
ninput=2; plotopt=2;
[theta,yres]=mlr(xreg,yreg,ninput,plotopt);
```



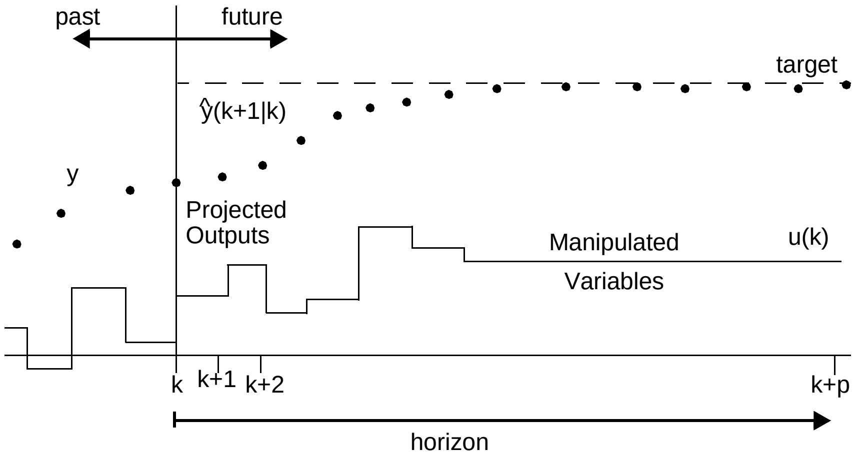

```
% Scale theta based on the standard deviations used in
% scaling the input.
theta=scal(theta,mx,stdx);
%
% Convert the impulse model to a step model to be used
% in MPC design.
% Sampling time of 7 minutes was used in determining
% the impulse model.
% Number of outputs (1 in this case) must be specified.
nout=1;
delt=7;
model=imp2step(delt,nout,theta);
%
% Plot the step response.
plotstep(model)
```


u1 step response : y1
0
1
2
3
4
5
6
0
50
100
150
200
250
TIME


u2 step response : y1
−0.2
0
0.2
0.4
0.6
0.8
1
1.2
1.4
1.6
1.8
0
50
100
150
200
250
TIME

Unconstrained Model Predictive Control

The MPC control law can be most easily derived by referring to the following figure.



For any assumed set of present and future control moves $\Delta u(k)$, $\Delta u(k+1)$, . . . , $\Delta u(k+m-1)$ the future behavior of the process outputs $y(k+1 \mid k)$, $y(k+2 \mid k)$, . . . , $y(k+p \mid k)$ can be predicted over a horizon p. The m present and future control moves ($m \le p$) are computed to minimize a quadratic objective of the form

$$\min_{\Delta u(k) \ldots \Delta u(k+m-1)} \quad \sum_{l=1}^{p} \left\| \Gamma_l^y ([y(k+l \mid \mathrm{k}) - r(k+l)]) \right\|^2 + \sum_{l=1}^{m} \left\| \Gamma_l^u [\Delta u(k+l-1)] \right\|^2$$

Here Γ_l^y and Γ_l^u are weighting matrices to penalize particular components of y or u at certain future time intervals. $r(k+l)$ is the (possibly time-varying) vector of future reference values (setpoints). Though m control moves are calculated, only the first one ($\Delta u(k)$) is implemented. At the next sampling interval, new values of the measured output are obtained, the control horizon is shifted forward by one step, and the same computations are repeated. The resulting control law is referred to as “moving horizon” or “receding horizon.”

The predicted process outputs $y(k+1\,|\,k), \ldots, y(k+p\,|\,k)$ depend on the current measurement $\hat{y}(k)$ and assumptions we make about the unmeasured disturbances and measurement noise affecting the outputs. The MPC Toolbox assumes that the unmeasured disturbances for each output are steps passing through a first order lag with time constant `tfilter(2,:)`.[1] For rejecting measurement noise, the time constant of an exponential filter `tfilter(1,:)` can be specified by the user. (It can be shown that this procedure is optimal for white noise disturbances passed through an integrator and a first order lag, and white measurement noise). For conventional Dynamic Matrix Control (DMC) the disturbance time constant is assumed to be zero (`tfilter(2,:) = zeros(1,ny)`), i.e., the unmeasured disturbances have the form of steps, and the noise filter time constant is also set to zero (`tfilter(1,:) = zeros(1,ny)`), i.e., there is no measurement noise filtering for doing the prediction.

Under the stated assumptions, it can be shown that a linear time-invariant feedback control law results

$$\Delta u(k) = K_{MPC}\, E_p(k+1\,|\,k)$$

where $E_p(k+1\,|\,k)$ is the vector of predicted future errors over the horizon p which would result if all present and future manipulated variable moves were equal to zero $\Delta u(k) = \Delta u(k+1) = \ldots = 0$.

For open-loop stable plants, nominal stability of the closed-loop system depends only on K_{MPC} which in turn is affected by the horizon p, the number of moves m and the weighting matrices Γ_l^y and Γ_l^u. No precise conditions on m, p, Γ_l^y and Γ_l^u exist which guarantee closed-loop stability. In general, decreasing m relative to p makes the control action less aggressive and tends to stabilize a system. For $p = 1$, nominal stability of the closed-loop system is guaranteed for any finite m, and time-invariant input and output weights. More commonly, Γ_l^u is used as a tuning parameter. Increasing Γ_l^u always has the effect of making the control action less aggressive.

The noise filter time constant `tfilter(1,:)` and the disturbance time constant `tfilter(2,:)` do not affect closed-loop stability or the response of the system to setpoint changes or measured disturbances. They do, however, affect the robustness and the response to unmeasured disturbances.

1. See `cmpc` in the “Reference” section for details on how to specify `tfilter`.

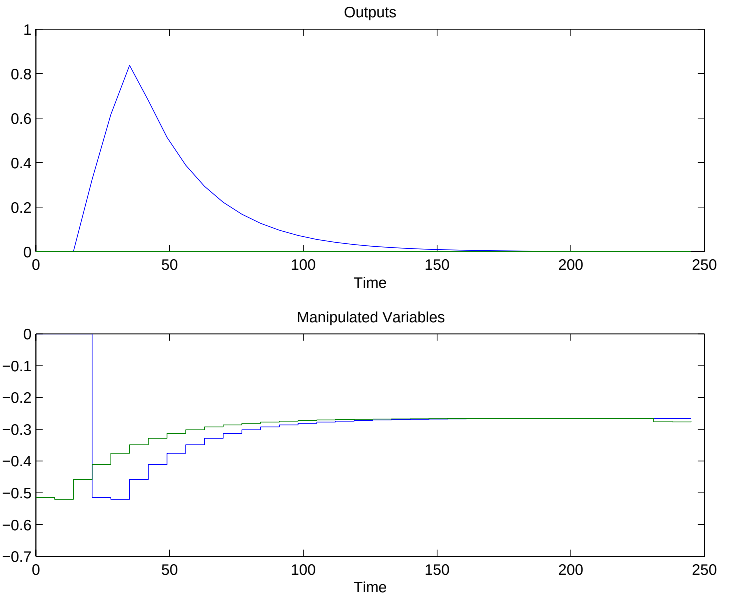
Increasing the noise filter time constant makes the system more robust and the unmeasured disturbance response more sluggish. Increasing the disturbance time constant increases the lead in the loop, making the control action somewhat more aggressive, and is recommended for disturbances which have a slow effect on the output.

All controllers designed with the MPC Toolbox track steps asymptotically error-free (Type 1). If the unmeasured disturbance model or the system itself is integrating, ramps are also tracked error-free (Type 2).

Example: (see `mpctutst.m`)

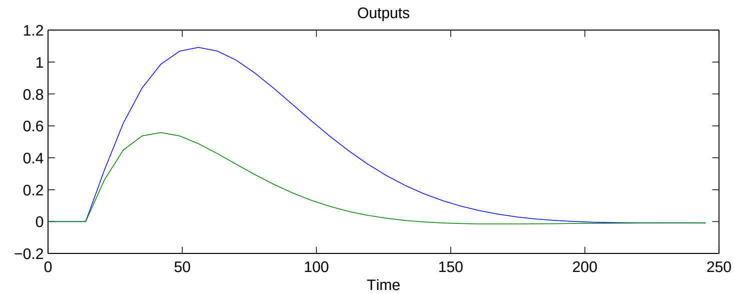
```
% Plant transfer function: g=5.72exp(-14s)/(60s+1)
% Disturbance transfer function: gd=1.52exp(-15s)/
% (25s+1)
%
% Build the step response models for a sampling period
% of 7.
delt1=0;
delay1=14;
num1=5.72;
den1=[60 1];
g=poly2tfd(num1,den1,delt1,delay1);
tfinal=245;
delt2=7;
nout1=1;
plant=tfd2step(tfinal,delt2,nout1,g);
delay2=15;
num2=1.52;
den2=[25 1];
gd=poly2tfd(num2,den2,delt1,delay2);
delt2=7;
nout2=1;
dplant=tfd2step(tfinal,delt2,nout2,gd);
%
% Calculate the MPC controller gain matrix for
% No plant/model mismatch,
% Output Weight=1, Input Weight=0
% Input Horizon=5, Output Horizon=20
model=plant;
ywt=1; uwt=0;
```

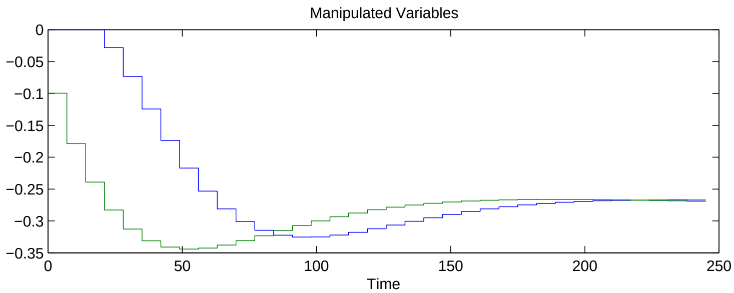
```
M=5; P=20;
Kmpc1=mpccon(model,ywt,uwt,M,P);
%
% Simulate and plot response for unmeasured and measured
% step disturbance through dplant.
tend=245;
r=[  ]; usat=[  ]; tfilter=[  ];
dmodel=[  ];
dstep=1;
[y1,u1]=mpcsim(plant,model,Kmpc1,tend,r,usat,tfilter,...
   dplant,dmodel,dstep);
dmodel=dplant; % measured disturbance
[y2,u2]=mpcsim(plant,model,Kmpc1,tend,r,usat,tfilter,...
   dplant,dmodel,dstep);
plotall([y1,y2],[u1,u2],delt2);
pause; % Perfect rejection for measured disturbance case.
```



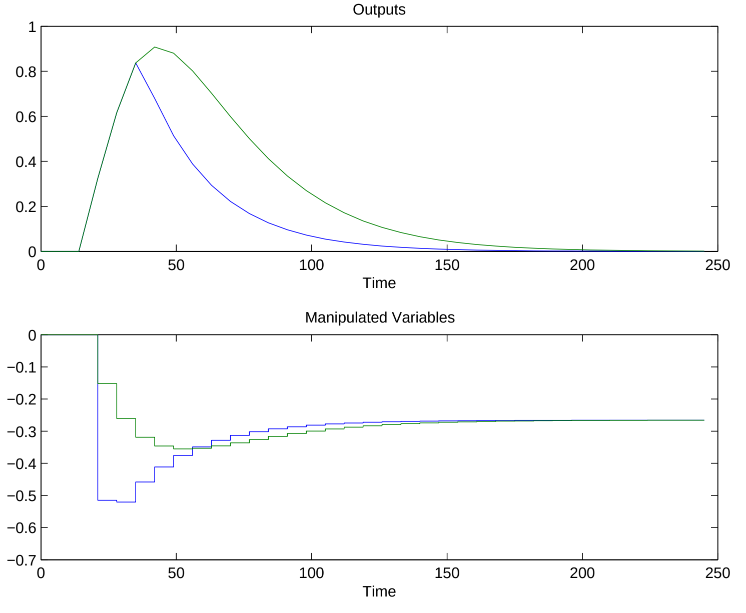

```
%
% Calculate a new MPC controller gain matrix for
% No plant/model mismatch,
% Output Weight=1, Input Weight=10
```

```
% Input Horizon=5, Output Horizon=20
model=plant;
ywt=1; uwt=10;
M=5; P=20;
mpc2=mpccon(model,ywt,uwt,M,P);
%
% Simulate and plot response for unmeasured and measured
% step disturbance through dplant.
tend=245;
r=[  ]; usat=[  ]; tfilter=[  ];
dmodel=[  ];
dstep=1;
[y3,u3]=mpcsim(plant,model,Kmpc2,tend,r,usat,tfilter,...
   dplant,dmodel,dstep);
dmodel=dplant; % measured disturbance
[y4,u4]=mpcsim(plant,model,Kmpc2,tend,r,usat,tfilter,...
   dplant,dmodel,dstep);
plotall([y3,y4],[u3,u4],delt2);
pause;
```

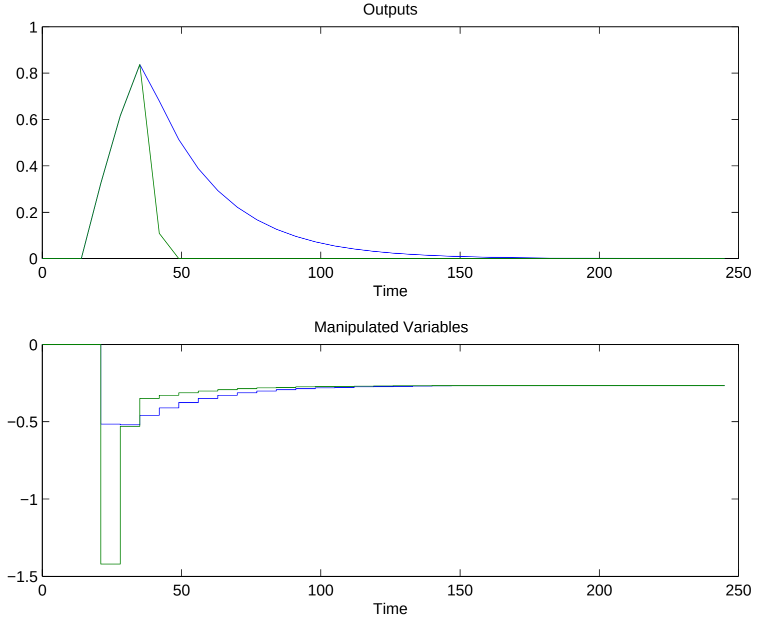




```
%
% Simulate and plot response for unmeasured
% step disturbance through dplant with uwt=0,
% with and without noise filtering.
tend=245;
r=[  ]; usat=[  ]; dmodel=[  ];
tfilter=[  ];
dstep=1;
[y5,u5]=mpcsim(plant,model,Kmpc1,tend,r,usat,tfilter,...
   dplant,dmodel,dstep);
tfilter=20; % noise filtering time constant=20
[y6,u6]=mpcsim(plant,model,Kmpc1,tend,r,usat,tfilter,...
   dplant,dmodel,dstep);
plotall([y5,y6],[u5,u6],delt2);
pause;
```



```
%
% Simulate and plot response for unmeasured
% step disturbance through dplant with uwt=0,
% with and without unmeasured disturbance time constant
% being specified.
tend=245;
r=[  ]; usat=[  ]; dmodel=[  ];
tfilter=[  ];
dstep=1;
[y7,u7]=mpcsim(plant,model,Kmpc1,tend,r,usat,tfilter,...
   dplant,dmodel,dstep);
tfilter=[0;25]; % unmeasured disturbance time constant=25
[y8,u8]=mpcsim(plant,model,Kmpc1,tend,r,usat,tfilter,...
   dplant,dmodel,dstep);
plotall([y7,y8],[u7,u8],delt2);
pause;
```



Closed-Loop Analysis

Apart from simulation, other tools are available in the MPC Toolbox to analyze the stability and performance of a closed-loop system. We can obtain the state-space description of the closed-loop system with the command `mpccl` and then determine the pole locations with `smpcpole`.

Example: (`mpctutst.m`)

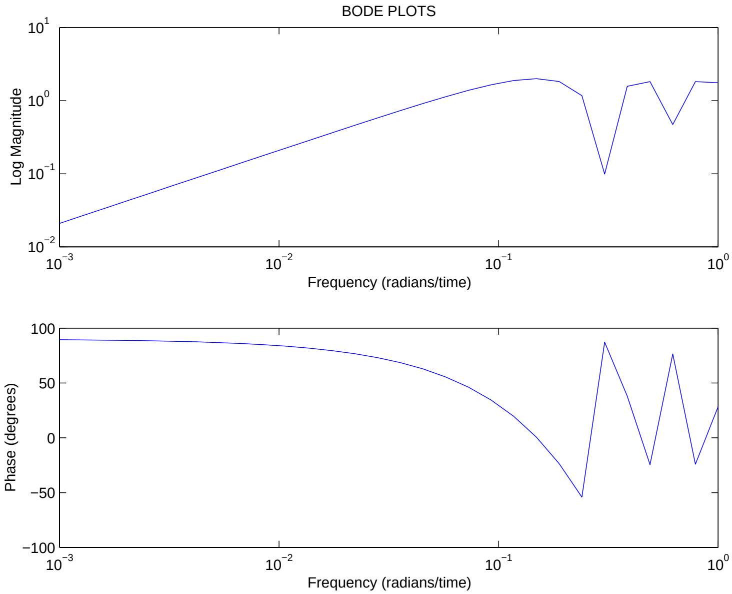
```
% Construct a closed-loop system for no disturbances
% and uwt = 0. Determine the poles of the system.
clmod = mpccl(plant,model,Kmpc1);
poles = smpcpole(clmod);
maxpole = max(poles)
Result is: maxpole = 1.0
```

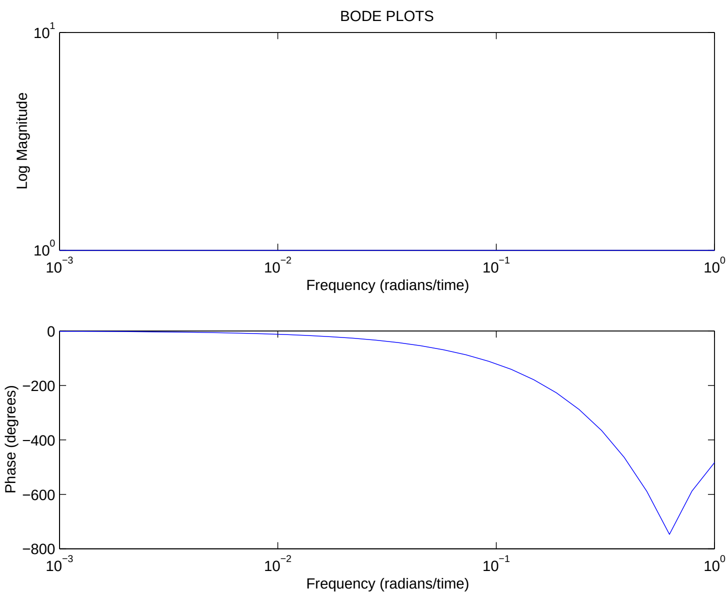
The closed-loop system is stable if all the poles are inside or on the unit-circle. Furthermore we can examine the frequency response of the closed-loop system. For multivariable systems, singular values as a function of frequency can be obtained using `svdfrsp`.

Example: (`mpctutst.m`)

```
% Calculate and plot the frequency response of the
% sensitivity and complementary sensitivity functions.
freq = [ -3,0,30];
ny = 1;
in = [1:ny]; % input is r for comp. sensitivity
out = [1:ny]; % output is yp for comp. sensitivity
[frsp,eyefrsp] = mod2frsp(clmod,freq,out,in);
plotfrsp(eyefrsp); % sensitivity
pause;

plotfrsp(frsp); % complementary sensitivity
pause; % Magnitude = 1 for all frequencies chosen.
```


BODE PLOTS
Log Magnitude
Phase (degrees)
Frequency (radians/time)


BODE PLOTS
Log Magnitude
Phase (degrees)
Frequency (radians/time)

Constrained Model Predictive Control

The control action can also be computed subject to hard constraints on the manipulated variables and the outputs.

Manipulated variable constraints:

$$u_{min}(l) \le u(k+l) \le u_{max}(l)$$

Manipulated variable rate constraints:

$$|\Delta u(k+l)| \le \Delta u_{max}(l)$$

Output variable constraints:

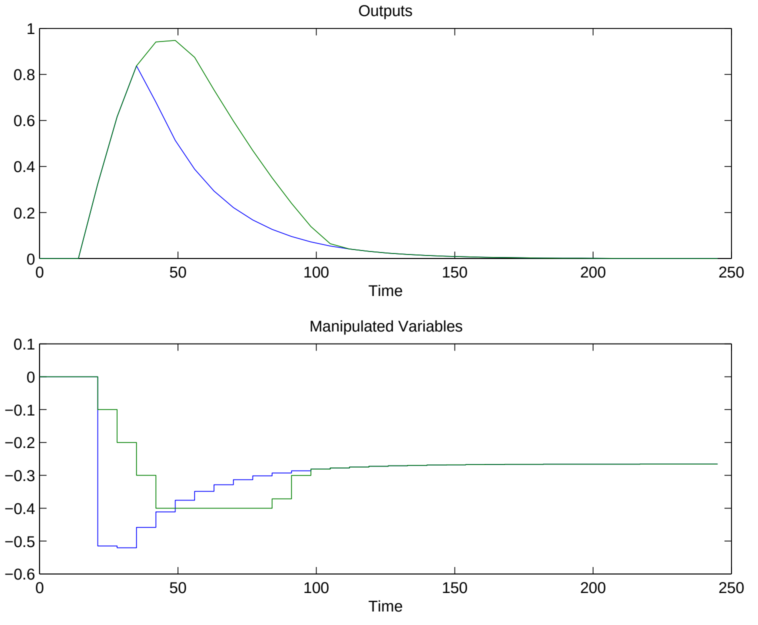
$$y_{min}(l) \le y(k+1|k) \le y_{max}(l)$$

When hard constraints of this form are imposed, a quadratic program has to be solved at each time step to determine the control action and the resulting control law is generally nonlinear. The performance of such a control system has to be evaluated via simulation.

Example: (mpctutst.m)

```
% Simulate and plot response for unmeasured step
% disturbance through dplant with and without
% input constraints.
% No plant/model mismatch,
% Output Weight = 1, Input Weight = 0
% Input Horizon = 5, Output Horizon = 20
% Minimum Constraint on Input = -0.4
% Maximum Constraint on Input = inf
% Delta Constraint on Input = 0.1
model = plant;
ywt = 1; uwt = 0;
M = 5; P = 20;
tend = 245;
r = 0;
ulim = [  ];
ylim = [  ]; tfilter = [  ]; dmodel = [  ];
dstep = 1;
[y9,u9] = cmpc(plant,model,ywt,uwt,M,P,tend,r,...
   ulim,ylim,tfilter,dplant,dmodel,dstep);
ulim = [ -0.4, inf, 0.1]; % impose constraints
```

```
[y10,u10] = cmpc(plant,model,ywt,uwt,M,P,tend,r,...
   ulim,ylim,tfilter,dplant,dmodel,dstep);
plotall([y9,y10],[u9,u10],delt2);
```



Application: Idle Speed Control

Process Description

An idle speed control[2] system should maintain the desired engine rpm with automatic transmission in neutral or drive. Despite sudden load changes due to the actions of air conditioning, power steering, etc., the control system should maintain smooth stable operation. Because of varying operating conditions and engine-to-engine variability inevitable in mass production, the system dynamics may change. The controller must be designed to be *robust* with respect to these changes. Two control inputs, bypass valve (or throttle) opening and spark advance, are used to maintain the engine rpm at a desired idle speed level. For safe operation, spark advance should not change by more than 20 degrees. Also, spark advance should be at the design point at steady-state for fuel economy. Thus, spark advance is viewed both as a manipulated input and a controlled output.

Control Problem Formulation

Here we consider two different operating conditions (transmission in neutral and drive positions) and the models for the two plants are taken from the paper by Hrovat and Bodenheimer.[3] The goal is to design a model predictive controller such that the closed loop performance at both operating conditions is good in the presence of the input constraint specified above. There is no synthesis method available which systematically generates a controller design which guarantees *robust* performance (or even just robust stability) in the presence of constraints. Thus, we must rely on both design and simulation tools to determine achievable performance objectives when there are both constraints and robustness requirements. The toolbox helps us toward this objective.

Consider the following system:

$$\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} G_{11} & G_{21} \\ 0 & 1 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} + \begin{bmatrix} G_d \\ 0 \end{bmatrix} w$$

2. More detail on the problem formulation can be found in the paper by Williams et al., “Idle Speed Control Design Using an H_∞ Approach,” *Proceedings of American Control Conference,* 1989, pages 1950–1956.

3. D. Hrovat and B. Bodenheimer, “Robust Automotive Idle Speed Control Design Based on μ-Synthesis,” *Proceedings of American Control Conference,* 1993, pages 1778–1783.

where y_1 is engine rpm, y_2 and u_2 are spark advance, u_1 is bypass valve, w is torque load (unmeasured disturbance), and G_{11}, G_{21} and Gd are the corresponding transfer functions. After scaling, the constraints on spark advance become ±0.7, i.e., $|u_2| \le 0.7$.

Plant #1 corresponds to operation in drive at 800 rpm and a load of 30 Nm and the transfer functions are given by

$$G_{11} = \frac{9.62e^{-0.16s}}{s^2 + 2.4s + 5.05}$$

$$G_{21} = \frac{15.9(s+3)e^{-0.04s}}{s^2 + 2.4s + 5.05}$$

$$G_d = \frac{-19.1(s+3)}{s^2 + 2.4s + 5.05}$$

Plant #2 corresponds to operation at 800 rpm in neutral with zero load and the transfer functions are given by

$$G_{11} = \frac{20.5e^{-0.16s}}{s^2 + 2.2s + 12.8}$$

$$G_{21} = \frac{47.6(s+3.5)e^{-0.04s}}{s^2 + 2.2s + 12.8}$$

$$G_d = \frac{-19.1(s+3.5)}{s^2 + 2.2s + 12.8}$$

The goal is to design a model predictive controller such that the closed-loop performance is good for plants #1 and #2 when subjected to an unmeasured torque load disturbance.

Simulations

Since the toolbox handles only discrete-time systems, the models are discretized using a sampling time of 0.1. We approximate each of the discrete transfer functions with 40 step response coefficients. The function `cmpc` is used to generate the controller and to simulate the closed-loop response because it determines *optimal* changes of the manipulated variables subject to constraints. For comparison (Simulation # 4), we also use the functions `mpccon` for controller design and `mpcsim` for simulating closed-loop responses. On-line computations are simpler, but the resulting controller is linear and the constraints are not handled in an optimal fashion. The following additional functions from the toolbox are also used: `tfd2step` and `plotall`. The MATLAB code for the following simulations can be found in the file `idlectr.m` in the directory `mpcdemos`.

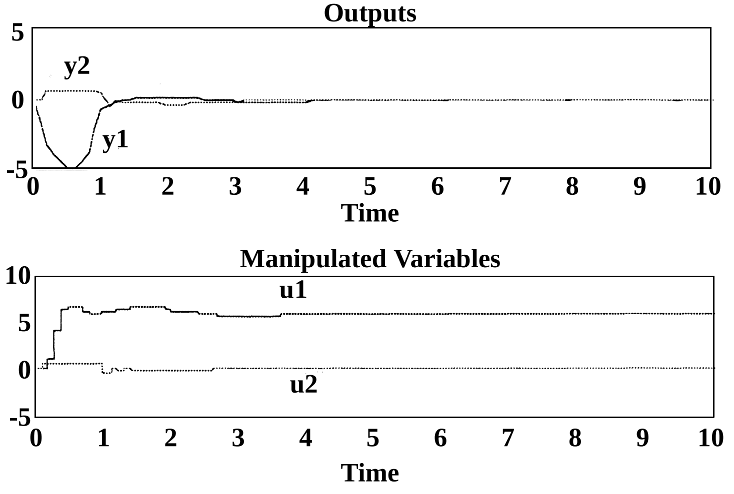


Figure 2-1 Responses to a Unit Torque Disturbance for Plant #1 (no model/ plant mismatch)

Simulation #1. No model/plant mismatch. The following parameters are used:

```
M = 10, P = inf, ywt = [5 1], uwt = [0.5 0.5],
tfilter = [  ]
```

The larger weight on the first output (engine rpm) is to emphasize that controlling engine rpm is more important than controlling spark advance. Figure 2-1 and Figure 2-2 show the closed-loop response for a unit step torque load change. No model/plant mismatch is introduced, i.e., we use Plant #1 and Plant #2 as the nominal model for simulating the closed loop response for Plant #1 and Plant #2, respectively.

As we can see, both controllers perform well for their respective plants. Because of the infinite output horizon, i.e., `P = inf`, nominal stability is guaranteed for both systems. In some sense, the performance observed in Figure 2-1 and Figure 2-2 is the best which can be expected, when the spark advance constraint is invoked and there is no model/plant mismatch. Obviously, if we want to control Plant #1 and Plant #2 with the same controller the nominal performance for each plant will deteriorate.

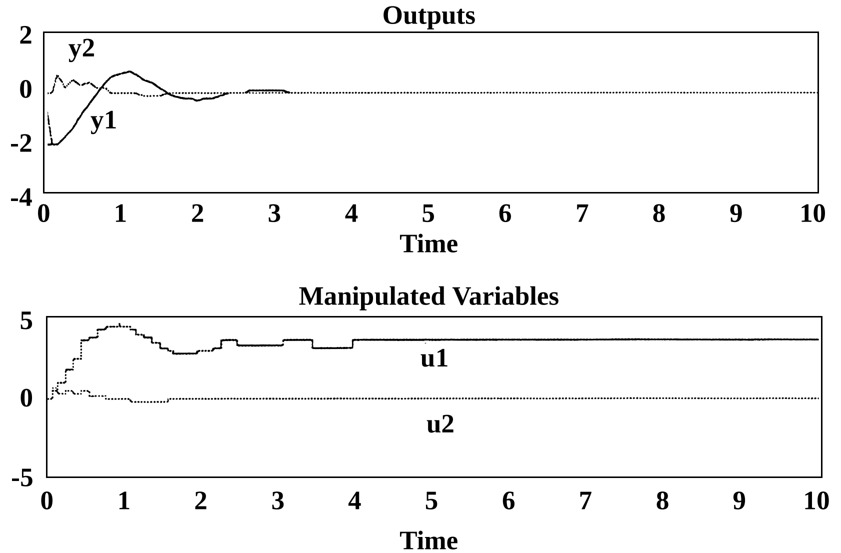


Figure 2-2 Responses to a Unit Torque Disturbance for Plant #2 (no model/plant mismatch)

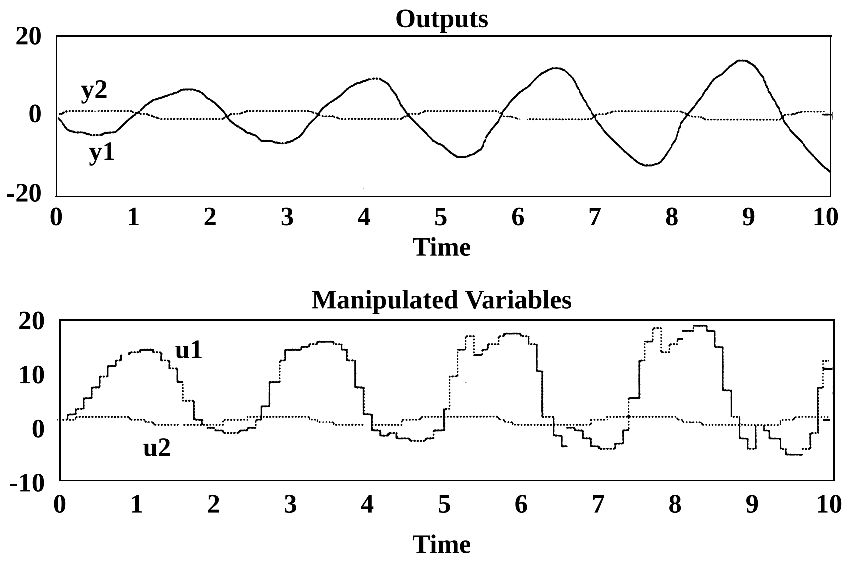


Figure 2-3 Responses to a Unit Torque Disturbance for Plant #1 (nominal model = Plant #2)

Simulation #2. Model/plant mismatch. All parameters are kept the same as in Simulation #1. Shown in Figure 2-3 is the response to a unit torque disturbance for Plant #1 using Plant #2 as the nominal model. Figure 2-4 depicts the response to a unit torque disturbance for Plant #2 using Plant #1 as the nominal model. As one can see, both systems are unstable. Therefore, the controllers must be detuned to improve robustness if one wants to control both plants with the same controller.

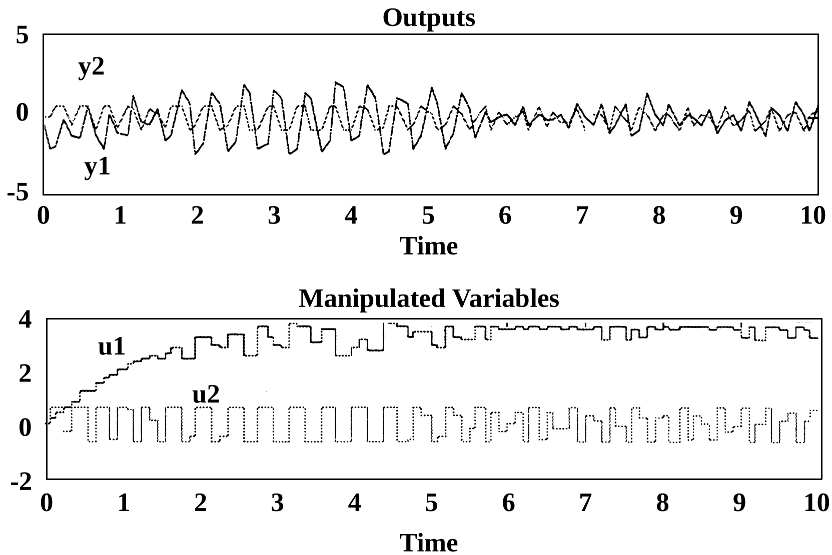


Figure 2-4 Responses to a Unit Torque Disturbance for Plant #2 (nominal model = Plant #1)

Simulation #3. The input weight is increased to [10 20] to improve robustness. All other parameters are kept the same as in Simulation #1. Plant #1 is used as the nominal model. The simulation results depicted in Figure 2-5 and Figure 2-6 seem to indicate that with an input weight of [10 20] the controller stabilizes both plants. However, we must point out that the design procedure guarantees global asymptotic stability only for the nominal system, i.e., Plant # 1. Because of the input constraints, the system is nonlinear. The observed stability for Plant # 2 in Figure 2-6 should not be mistaken as an indication of global asymptotic stability.

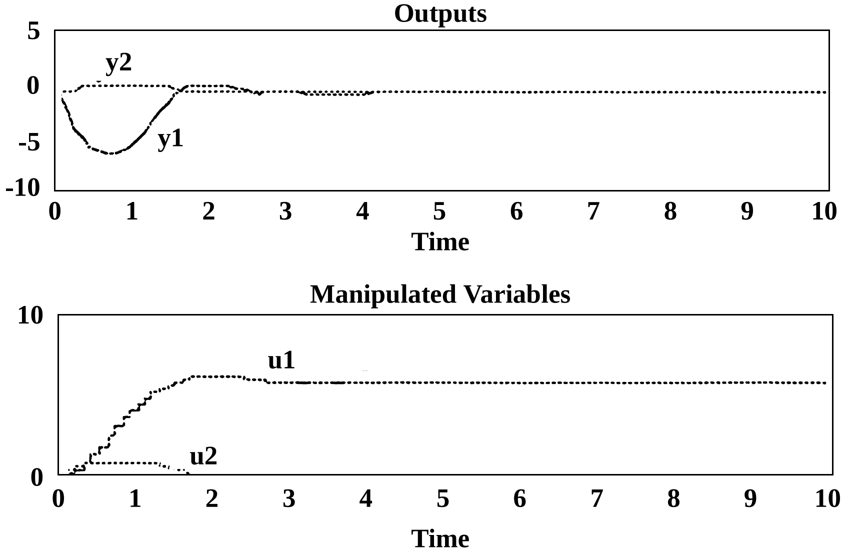


Figure 2-5 Responses to a Unit Torque Disturbance for Plant #1 (nominal model = Plant #1)

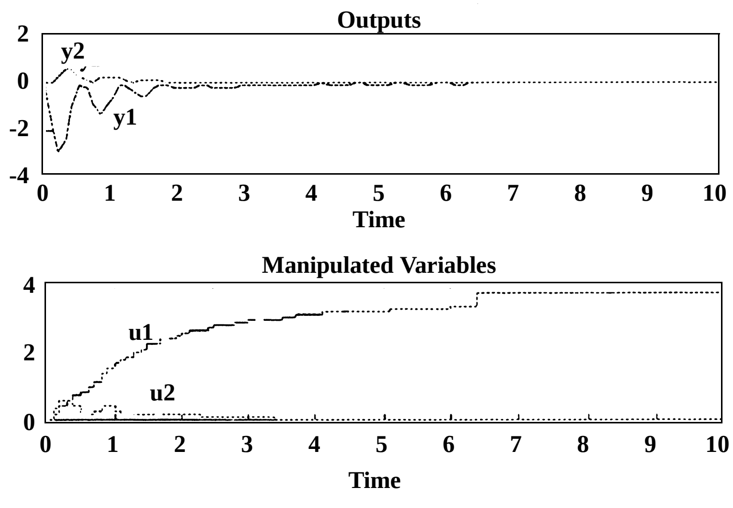


Figure 2-6 Responses to a Unit Torque Disturbance for Plant #2 (nominal model = Plant #1)

As expected, the nominal performance for both Plant #1 and Plant #2 has deteriorated when compared to the simulations shown in Figure 2-1 and Figure 2-2. A similar effect would be observed if we had detuned the controller which uses Plant #2 as the nominal model.

Simulation #4. The parameter values are the same as in Simulation #3. Instead of using `cmpc`, we use `mpccon` and `mpcsim` for simulating the closed loop responses. Figure 2-2 compares the responses for Plant #1 using `mpccon` and `mpcsim`, and `cmpc`. As we can see, for this example and these tuning parameters, the improvement obtained through the on-line optimization in `cmpc` is small. However, the difference could be large, especially for ill-conditioned systems and other tuning parameters. For example, by reducing the output horizon to P = 80 while keeping the other parameters the same, the responses for Plant # 1 found with `mpccon` and `mpcsim` are significantly slower than those obtained with `cmpc` (Figure 2-8).

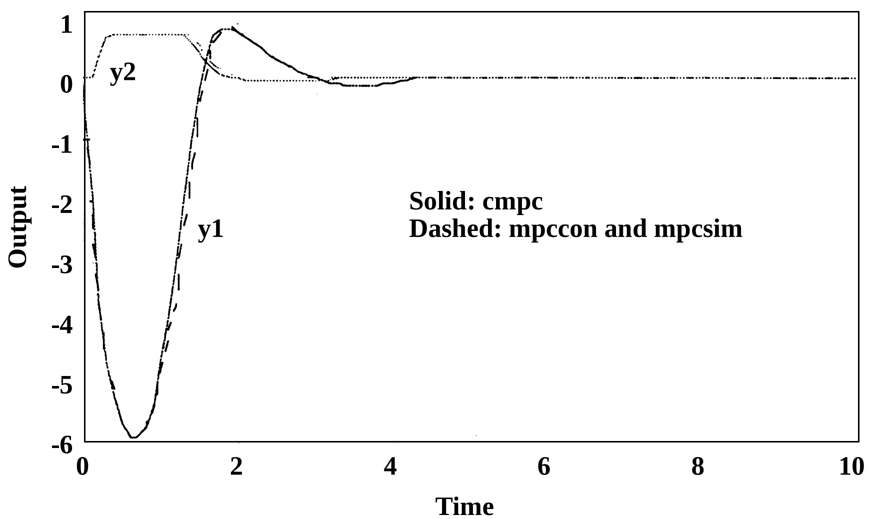


Figure 2-7 Comparison of Responses From `cmpc`**, and** `mpccon` **and** `mpcsim` **for Plant #1** `P = inf`

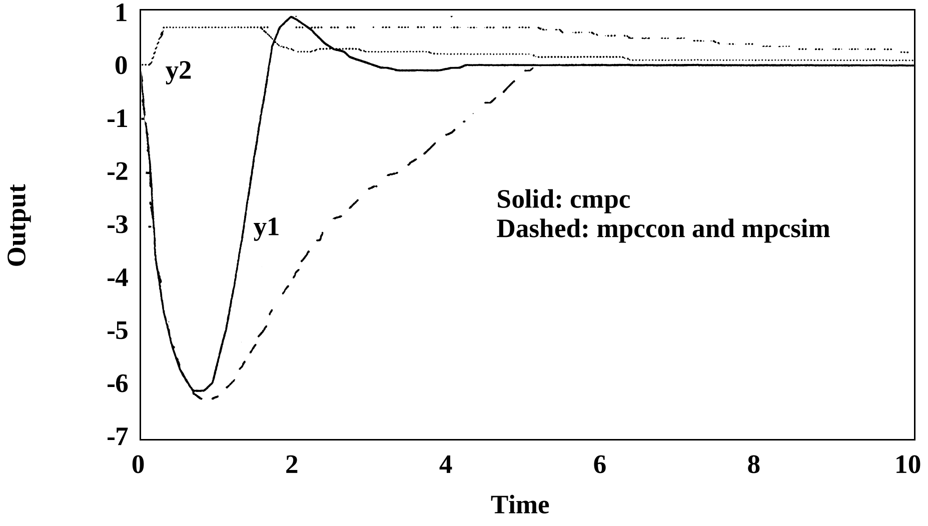


Figure 2-8 Comparison of Responses From `cmpc`**, and** `mpccon` **and** `mpcsim` **for Plant #1** (`P = 80`)

Application: Control of a Fluid Catalytic Cracking Unit

Process Description

Fluid Catalytic Cracking Units (FCCUs) are widely used in the petroleum refining industry to convert high boiling oil cuts (of low economic value) to lighter more valuable hydrocarbons including gasoline. Cracking refers to the catalyst enhanced thermal breakdown of high molecular weight hydrocarbons into lower molecular weight materials. A schematic of the FCCU studied[4] is given in Figure 2-9. Fresh feed is contacted with hot catalyst at the base of the riser and travels rapidly up the riser where the cracking reactions occur. The desirable products of reaction are gaseous (lighter) hydrocarbons which are passed to a fractionator and subsequently to separation units for recovery and purification. The undesirable byproduct of cracking is coke which is deposited on the catalyst particles, reducing their activity. Catalyst coated with coke is transported to the regenerator section where the coke is burned off thereby restoring catalytic activity and raising catalyst temperature. The regenerated catalyst is then transported to the riser base where it is contacted with more fresh feed. Regenerated catalyst at the elevated temperature provides the heat required to vaporize the fresh feed as well as the energy required for the endothermic cracking reaction.

4. A detailed problem description and the model used for this study can be found in the paper by McFarlane et al., "Dynamic Simulator for a Model IV Fluid Catalytic Cracking Unit," *Comp. & Chem. Eng.*, 17(3), 1993, pages 275–300

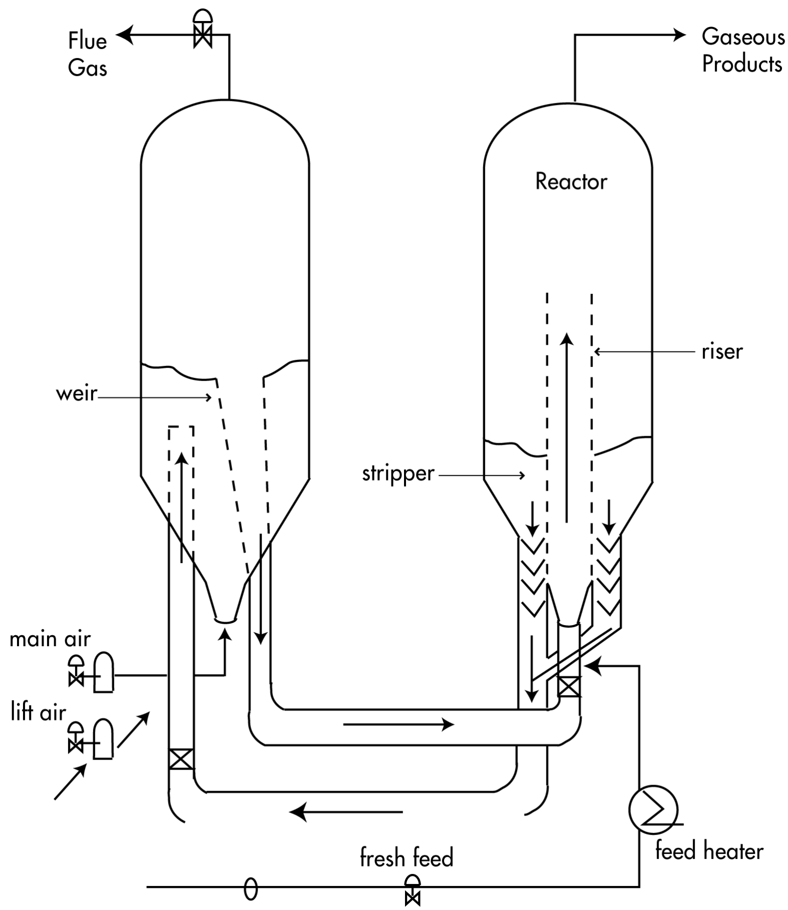


Figure 2-9 Fluid Catalytic Cracking Unit Schematic

Product composition, and therefore the economic viability of the process, is determined by the cracking temperature. The bulk of the combustion air in the regenerator section is provided by the main air compressor which is operated at full capacity. Additional combustion air is provided by the lift air compressor, the throughput of which is adjustable by altering compressor speed. By maintaining excess flue gas oxygen concentration, it is possible to ensure essentially complete coke removal from the catalyst.

Control Problem Formulation

The open loop system is modeled as follows:

$$y = Gu + G_d d$$

where

$$u = [V_{fg} V_{lift}]^T \quad y = [C_{O_{2,sg}} T_r \Delta F_{la}]^T \quad d = [d_1 d_2 d_3]^T$$

G is the plant model and G_d is the disturbance model. The variables are:

- Controlled variables
  - Cracking temperature — (T_r)
  - Flue gas oxygen concentration — ($C_{O_{2,sg}}$)
- Associated variable
  - Lift Air Compressor Surge Indicator — (ΔF_{la})
- Manipulated variables
  - Lift air compressor speed — (V_{lift})
  - Flue gas valve opening — (V_{fg})
- Modeled disturbances
  - Variations in ambient temperature affect compressor throughput — (d_1)
  - Fluctuations in heavy oil feed composition to the FCCU — (d_2)
  - Pressure upset in down stream units propagating back to the FCCU — (d_3)

In addition to the controlled variables there are many process variables that need not be maintained at specific setpoints but which need to be within bounds for safety or economic reasons. These variables are called *associated variables*. For example, compressors must not surge during process upsets i.e., the suction flow rate must be greater than some minimum flow rate (surge flow rate).

The control objective is to maintain the controlled variables (cracking temperature and flue gas oxygen concentration) at pre-determined setpoints in the presence of typical process disturbances while maintaining safe plant operation.

Simulations

State-space realizations of the plant and disturbance models are available in the MATLAB file `fcc_dat.mat` in the directory `mpcdemos`. A MATLAB script detailing the simulations is also included (`fcc_demo.m`). The following table gives the parameters used for controller design and examination of the closed loop response:

Table 2-1 FCCU Simulation Parameters

Simulation Time(s)	`tend = 2500`
# Step Response Coefficients	`60`
Process Sampling Time	`delt2 = 100`
Output Weights	`ywt = [3 3 0]`
Input Weights	`uwt = [0 2]`
Prediction Horizon (steps)	`P = 12`
# manipulated variable moves	`M = 3`
input constraints	$u_i \in [-1,1]$, $i = 1,2$
output constraints	$y_i \in [-1,1]$, $i = 1,2$ y_3 Š -1 (hard constraint)

Step Response Model

Figure 2-10A shows the plant open loop step response to a unit step in V_{fg}. Although the plant is stable the settling time is large (1 day). The time scale of interest for control purposes is on the order of one hour — which corresponds to the initial plant response, Figure 2-10B. For time scales of one hour, the process can be approximated by an integrating system. In deriving the step response model, the plant is therefore assumed to be an integrating process.

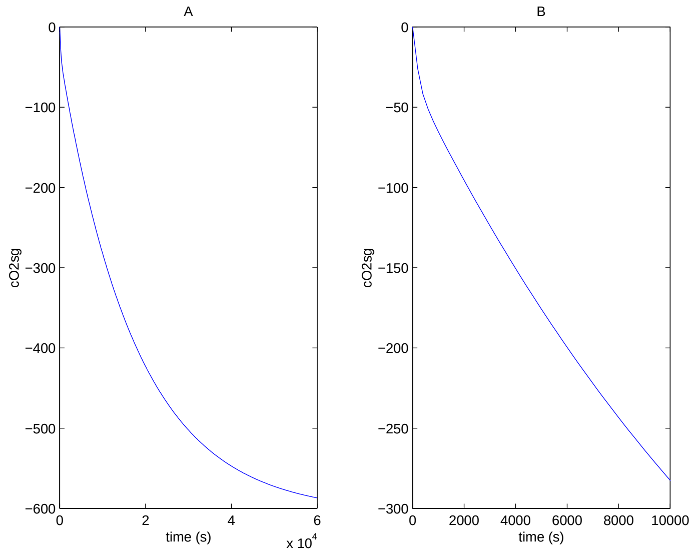


Figure 2-10 Open Loop Step Response to `u = [1 0]`

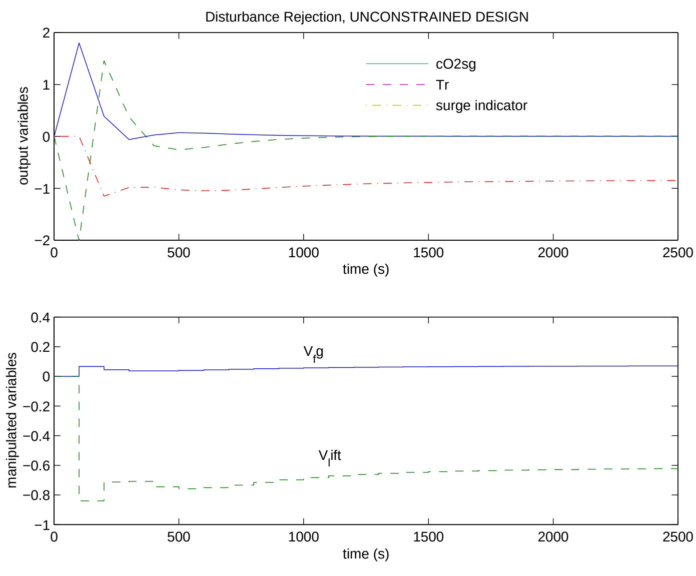


Figure 2-11 Unconstrained Closed Loop Response to `d = [0 -0.5 -0.75]`

Associated Variables

As mentioned previously, associated variables need not be at any setpoint as long as they are within acceptable bounds. Deviations of associated variables from the nominal value do not appear in overall objective function to be minimized and the output weight corresponding to the associated variable is set to zero in Table 2-1.

Unconstrained Control Law

Figure 2-11 shows the closed loop response to a disturbance `d = [0 -0.5 -0.75] at t = 0`. The controller gain matrix is derived using `mpccon` and the closed loop response is examined using `mpcsim`. Note the following:

- At the first time step (t = 100s) the controlled variables are outside their allowed limits. The onset of the disturbance at t = 0 is unknown to the controller at t = 0 since there is no disturbance feedforward loop. Thus from t = 0 to t = 100s there is no control action and the process response is the open loop response with no control action. Only after t = 100s is corrective action implemented.
- At t = 200s (2^{nd} time step) riser temperature is outside the allowed limits.
- The lift air compressor surges during the interval $200s \le t \le 800s$ which is unacceptable. Compressor surging will result in undesirable vibrations in the compressor leading to rapid wear and tear.

Constrained Control Law

It is clear that the unconstrained control law generated using `mpcsim` is physically unacceptable since hard output constraints are violated. Figure 2-12 shows the closed loop response of the nominal plant to the same disturbance taking process constraints explicitly into account. The closed loop response is determined using the command `cmpc`.

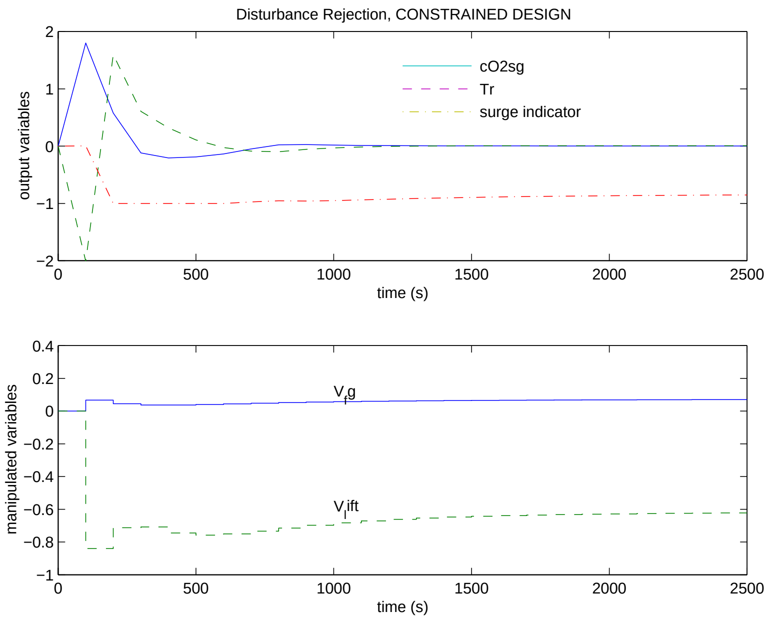


Figure 2-12 Constrained Closed Loop Response to d = [0 -0.5 -0.75]

The output limit vector is:

$$ylim = \begin{bmatrix} -1 & -1 & -1 & 1 & 1 & Inf \\ -1 & -1 & -1 & 1 & 1 & Inf \\ -Inf & -Inf & -Inf & Inf & Inf & Inf \end{bmatrix}$$

Note the following:

- At the first time step (t = 100s) the controlled variables are outside their allowed limits. In fact the outputs are identical to the outputs for the unconstrained case at t = 100s. This should be expected as there is no control action from t = 0 to t = 100s for both constrained and unconstrained designs.
- At t = 200s (2^{nd} time step) riser temperature (y_2) is still outside the allowed limits. This is because the constrained QP solved at t = 100s assumes that disturbances are constant for t > 100s which is not the case for this process. Thus while the manipulated variable move made at t = 100s ensures that the predicted y_2 = 1 at t = 200s, the actual output at t = 200s exceeds one.
- The lift air compressor does not surge during the disturbance transient, Figure 2-13.

The constrained control law therefore ensures safe operation while rejecting process disturbances. If no constraints are violated, `mpccon` and `mpcsim`, and `cmpc` will give identical closed loop responses. Note that the disturbance `d = [0 -0.5 -0.75]` was specifically chosen to illustrate the possibility of constraint violations during disturbance transient periods.

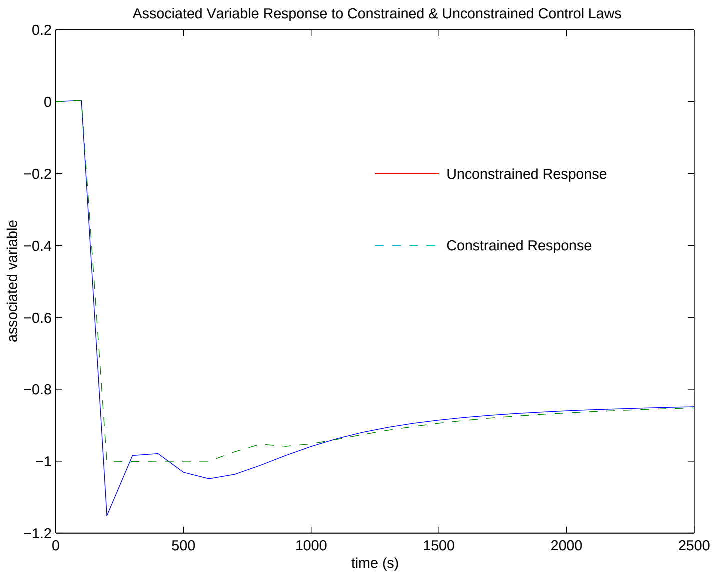
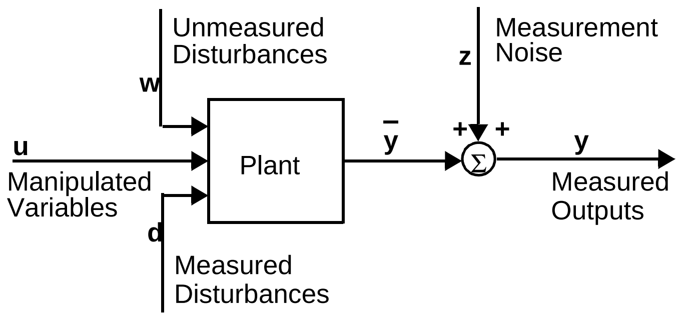


Figure 2-13 Comparison of Constrained and Unconstrained Response of ΔF_{la} to `d = [0 -0.5 -0.75]`

3

MPC Based on State-Space Models

State-Space Models



Consider the process shown in the above block diagram. The general discrete-time linear time invariant (LTI) state-space representation used in the MPC Toolbox is:

$$x(k+1) = \Phi x(k) + \Gamma_u u(k) + \Gamma_d d(k) + \Gamma_w w(k)$$

$$y(k) = \bar{y}(k) + z(k)$$

$$= Cx(k) + D_u u(k) + D_d d(k) + D_w w(k) + z(k)$$

where x is a vector of n state variables, u represents the n_u manipulated variables, d represents n_d measured but freely-varying inputs (i.e., measured disturbances), w represents n_w unmeasured disturbances, y is a vector of n_y plant outputs, z is measurement noise, and Φ, Γ_u, etc., are constant matrices of appropriate size. The variable $\bar{y}(k)$ represents the plant output before the addition of measurement noise. Define:

$$\Gamma = \begin{bmatrix} \Gamma_u & \Gamma_d & \Gamma_w \end{bmatrix}$$

$$D = \begin{bmatrix} D_u & D_d & D_w \end{bmatrix}$$

In many applications, all outputs are measured. In some cases, however, one has n_{ym} *measured* and n_{yu} *unmeasured outputs* in y, where $n_{ym} + n_{yu} = n_y$. If so, the MPC Toolbox assumes that the y vector and the C and D matrices are arranged such that the measured outputs come first, followed by the unmeasured outputs.

Mod Format

The MPC Toolbox works with state-space models in a special format, called the **mod** format. The **mod** format is a single matrix that contains the state-space Φ, Γ, C, and D matrices plus some additional information (see `mod format` in the "Command Reference" chapter for details). The MPC Toolbox includes a number of commands that make it easy to generate models in the **mod** format. The following sections illustrate the use of these commands.

SISO Continuous-Time Transfer Function to Mod Format

The MPC Toolbox uses a format called the `tf` format. Let the continuous-time transfer function be

$$G(s) = \frac{b_0 s^n + b_1 s^{n-1} + \ldots + b_n}{a_0 s^n + a_1 s^{n-1} + \ldots + a_n} e^{-T_d s}$$

where T_d is the time delay. The `tf` format is a matrix consisting of three rows:

row 1:	The n coefficients of the numerator polynomial, b_0 to b_n.
row 2:	The n coefficients of the denominator polynomial, a_0 to a_n.
row 3:	column 1: The sampling period. *This must be zero for a continuous system*. (It must be positive for discrete transfer functions — see next section).
	column 2: The time delay in time units. It must satisfy $Td \geq 0$.

The `tf` matrix will always have at least two columns, since that is the minimum width of the third row.

You can either define a model in the `tf` format directly or use the command `poly2tfd`. The general form of this command is

```
g = poly2tfd(num,den,delt,delay)
```

For example, consider a SISO system modeled by the transfer function

$$G(s) = \frac{-13.6s + 1}{54.3s^2 + 113.5s + 1} e^{-5.3s}$$

To create the `tf` format directly you could use the command

```
G = [0 -13.6 1; 54.3 113.5 1; 0 5.3 0];
```

which defines a matrix consisting of three rows and three columns. Note that all rows must have the same number of columns so you must be careful to insert zeros where appropriate. The `poly2tfd` command is more convenient since it does that for you automatically:

```
G = poly2tfd([-13.6 1],[54.3 113.5 1],0,5.3);
```

Either command would define a variable `G` in your workspace, containing the matrix

```
G=
        0     -13.6000     1.0000
  54.3000     113.5000     1.0000
        0       5.3000          0
```

To convert this to the **mod** format, use the command `tfd2mod`, which has the form

```
model = tfd2mod(delt,ny,g1,g2,g3,...,gN)
```

where:

`delt`	The sampling period. `tfd2mod` will convert your continuous time transfer function(s) `g1, ..., gN` to discrete-time using this sampling period.
`ny`	is the number of output variables in the plant you are modeling.
`g1,g2,...,gN`	A sequence of N transfer functions in the *tf* format, where $N \geq 1$. `tfd2mod` assumes that these are the individual elements of a transfer-function matrix: $$\begin{bmatrix} g_{1,1} & g_{1,2} & \cdots & g_{1,n_u} \\ g_{2,1} & g_{2,2} & \cdots & g_{2,n_u} \\ \vdots & \vdots & \ddots & \vdots \\ g_{n_y,1} & g_{n_y,2} & \cdots & g_{n_y,n_u} \end{bmatrix}$$ Thus N must be an integer multiple (n_u) of the number of outputs, n_y. You supply the transfer functions in *column-wise* order. In other words, you first give the n_y transfer functions for input 1 ($g_{1,1}$ to $g_{n_y,1}$), then the n_y transfer functions for input 2 ($g_{1,2}$ to $g_{n_y,2}$), etc.

Suppose you want to convert the SISO model defined above to the **mod** format with a sampling period of 2.1 time units. The appropriate command would be

```
mod = tfd2mod(2.1,1,G);
```

This would define a variable called `mod` in your workspace that would contain the discrete-time state-space description of your system.

SISO Discrete-Time Transfer Function to Mod Format

Suppose you have a transfer function in discrete-time format (in terms of the forward-shift operator, z):

$$G(q) = \frac{b_0 + b_1 z^{-1} + \ldots + b_n z^{-n}}{a_0 + a_1 q^{-1} + \ldots + a_n z^{-n}} z^{-d}$$

where d is an *integer* (≥ 0) and represents the sampling periods of pure delay. The corresponding *tf* format is the same as for the continuous-time case *except* for the definition of the third row:

column 1	is the sampling period for which $G(z)$ was created. It must be positive (in contrast to the continuous-time case described above).
column 2	is the *periods* of pure delay, d, which must be an *integer* ≥ 0. Contrast this to the continuous case, where the delay is given in time units.

As in the previous section, you can use `poly2tfd` followed by `tfd2mod` to get such a transfer function in **mod** format. For example, the discrete-time representation of the SISO system considered in the previous section is

$$G(z) = \frac{-0.1048 + 0.1215 z^{-1} + 0.0033 z^{-2}}{1 - 0.9882 z^{-1} + 0.0082 z^{-2}} z^{-3}$$

If you had this to begin with, you could convert it to the **mod** format as follows:

```
G = poly2tfd([-0.1048 0.1215 0.0033],[1 -0.9882 0.0082], 2.1,3);
mod = tfd2mod(2.1,1,G);
```

Note that both the `poly2tfd` and `tfd2mod` commands specify the same sampling period (`delt=2.1`). This would be the usual case, but you have the option of converting a discrete-time model in the *tf* format to a different sampling period in the **mod** format.

MIMO Transfer Function Description to Mod Format

Suppose you have a transfer-function matrix description of your system in the form

$$\begin{bmatrix} g_{1,1} & g_{1,2} & \cdots & g_{1,n_u} \\ g_{2,1} & g_{2,2} & \cdots & g_{2,n_u} \\ \vdots & \vdots & \ddots & \vdots \\ g_{n_y,1} & g_{n_y,2} & \cdots & g_{n_y,n_u} \end{bmatrix}$$

where $g_{i,j}$ is the transfer function of the i^{th} output with respect to the j^{th} input. If all n_y outputs are measured and all n_u inputs are manipulated variables, the default mode of `tfd2mod` will give you the correct **mod** format. For example, consider the 2-output, 3-input system:

$$\begin{bmatrix} y_1(s) \\ y_2(s) \end{bmatrix} = \begin{bmatrix} \dfrac{12.8e^{-s}}{16.7s+1} & \dfrac{-18.9e^{-3s}}{21.0s+1} & \dfrac{3.8e^{-8s}}{14.9s+1} \\ \dfrac{6.6e^{-7s}}{10.9s+1} & \dfrac{-19.4e^{-3s}}{14.4s+1} & \dfrac{4.9e^{-3s}}{13.2s+1} \end{bmatrix} \begin{bmatrix} u_1(s) \\ u_2(s) \\ u_3(s) \end{bmatrix}$$

The following sequence of commands would convert this to the equivalent **mod** format with a sampling period of $T = 4$:

```
g11 = poly2tfd(12.8,[16.7 1],0,1);
g21 = poly2tfd(6.6,[10.9 1],0,7);
g12 = poly2tfd(-18.9,[21.0 1],0,3);
g22 = poly2tfd(-19.4,[14.4 1],0,3);
g13 = poly2tfd(3.8,[14.9 1],0,8);
g23 = poly2tfd(4.9,[13.2 1],0,3);
pmod = tfd2mod(4,2,g11,g21,g12,g22,g13,g23);
```

Suppose, however, that the third input were actually an unmeasured disturbance, i.e., the system were

$$\begin{bmatrix} y_1 s \\ y_2 s \end{bmatrix} = \begin{bmatrix} \dfrac{12.8e^{-s}}{16.7s+1} & \dfrac{-18.9e^{-3s}}{21.0s+1} \\ \dfrac{6.6e^{-7s}}{10.9s+1} & \dfrac{-19.4e^{-3s}}{14.4s+1} \end{bmatrix} \begin{bmatrix} u_1 s \\ u_2 s \end{bmatrix} + \begin{bmatrix} \dfrac{3.8e^{-8s}}{14.9s+1} \\ \dfrac{4.9e^{-3s}}{13.2s+1} \end{bmatrix} w(s)$$

In this case you would need to override the default mode of `tfd2mod` by specifying the number of inputs in each of the three categories described at the beginning of this section, i.e., manipulated variables, measured disturbances, and unmeasured disturbances. This and other information about the system is contained in the first 7 columns of row 1 of the **mod** format, as follows:

column	1	T, the sampling period.
	2	n, the number of states.
	3	n_u, the number of manipulated variable inputs.
	4	n_d, the number of measured disturbances.
	5	n_w, the number of unmeasured disturbances.
	6	n_{ym}, the number of measured outputs.
	7	n_{yu}, the number of unmeasured outputs.

For example, if you had defined `pmod` using the default mode of `tfd2mod` as shown above, the contents of row 1, columns 1 to 7 of `pmod` would be:

```
4 13 3 0 0 2 0
```

You could override this to set $n_u = 2$ and $n_w = 1$ as follows:

```
pmod(1,3)=2;
pmod(1,5)=1;
```

Note that in the original transfer function matrix description, the first n_u columns must be for the manipulated variables, the next n_d for the measured disturbances (if any), and the last n_w for the unmeasured disturbances (if any). Similarly, the first n_{ym} outputs must be measured and the last n_{yu} (≥ 0) unmeasured.

Continuous or Discrete State-Space to Mod Format

If you have a continuous-time state-space model, you may convert it to **mod** format by first using the function `c2dmp` (continuous to discrete-time state-space), followed by `ss2mod`(discrete-time state-space to **mod** format). Of course, if you are starting with a discrete-time state-space model you can skip the first step.

For example, suppose *a*, *b*, *c*, and *d* are matrices describing a continuous-time system. To convert to the **mod** format using a sampling period of $T = 1.5$, you could use the following commands:

```
[phi,gam] = c2dmp(a,b,1.5);
mod = ss2mod(phi,gam,c,d,1.5);
```

If your system is complicated, i.e., it contains disturbance inputs and/or unmeasured outputs, you will need to override the default mode of `ss2mod`. See the "Command Reference" section for more details.

Identification Toolbox ("Theta") Format to Mod Format

The System Identification Toolbox identifies discrete-time transfer-function models from input/output data. The result is a model in a special form called the *theta* format (see the *System Identification Toolbox User's Guide* for details). In general, each *theta* model describes the response of a single output to one or more inputs (MISO model).

The MPC Toolbox function, `th2mod`, converts one or more such models to the **mod** format. Suppose, for example, that

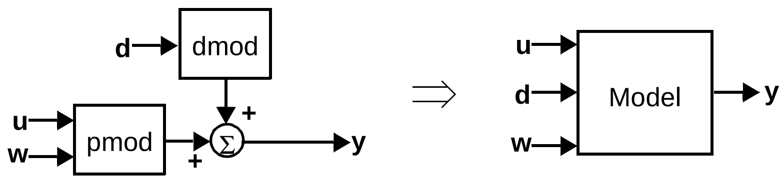
`th1`	is the *theta* model describing the response of output y_1 to inputs u_1 and u_2.
`th2`	is the *theta* model describing the response of output y_2 to inputs u_1 and u_2.

Then the following command would provide the equivalent **mod** format with $n_y = 2$ and $n_u = 2$:

```
mod = th2mod(th1,th2);
```

Combination of Models in Mod Format

The functions addmod, addmd, addumd, appmod, paramod, and sermod allow you to combine simple models in the **mod** format to generate more complex plant structures. For example, addmd includes the effect of one or more measured disturbances in an existing model, as shown in the following schematic:



pmod gives the effect of one or more manipulated variables, $u(k)$, and optional unmeasured disturbance(s), $w(k)$, on the output(s), $y(k)$. dmod gives the effect of the measured disturbance(s), $d(k)$, on the same outputs. Once you have defined pmod and dmod (e.g., starting from transfer functions as illustrated above), you can use the command addmd to generate the composite, model:

```
model = addmd(pmod,dmod);
```

Please see Chapter 4, “Command Reference” for more details on the various model-building commands.

Converting Mod Format to Other Model Formats

The function mod2ss converts a model in the **mod** format to the standard discrete-time state-space format:

```
[phi,gam,c,d,minfo] = mod2ss(mod);
```

Here, phi, gam, c, and d are the coefficient matrices of

$$x(k+1) = \Phi x(k) + \Gamma u(k)$$

$$y(k) = Cx(k) + Du(k)$$

The vector minfo contains the first 7 columns of the first row in mod. The section "MIMO Transfer Function Description to Mod Format" gives the significance of this information.

Once you have phi, gam, c, and d, you can use d2cmp, ss2tf2, and other functions to convert from discrete state-space to other model forms.

The function mod2step uses a model in the **mod** format to generate a step-response model in the **step** format as required by the functions mpccon, mpcsim, etc., discussed in Chapter 2, "MPC Based on Step Response Models". See the Chapter 4, "Command Reference" for details on the use of mod2step.

Unconstrained MPC Using State-Space Models

Once you have described your system by generating state-space models in the **mod** format you can use the commands:

`smpccon`	to calculate the unconstrained controller gain matrix.
`smpcest`	to design a state estimator (optional).
`smpcsim`	to simulate the response of the closed-loop system to one or more specified inputs.
`plotall`	(or `ploteach`) to plot the response(s).

In addition, you can analyze certain properties of the closed-loop system using the commands:

`smpccl`	to generate a model of the closed-loop system (plant plus controller).
`smpcgain`	to calculate the closed-loop gain matrix.
`smpcpole`	to calculate the closed-loop poles.
`mod2frsp`	(and `plotfrsp`) to calculate and plot the closed-loop frequency response.
`svdfrsp`	to calculate the singular values of the frequency response.

Note: `smpcgain`, `smpcpole` and `mod2frsp` also work with open-loop models in the **mod** format.

Example: (see `mpctutss.m`)

The following example (`mpctutss.m`) illustrates the basic procedures. The example process has 2 measured outputs, 2 manipulated variables, and an unmeasured disturbance:

$$\begin{bmatrix} y_1 s \\ y_2 s \end{bmatrix} = \begin{bmatrix} \dfrac{12.8e^{-s}}{16.7s+1} & \dfrac{-18.9e^{-3s}}{21.0s+1} \\ \dfrac{6.6e^{-7s}}{10.9s+1} & \dfrac{-19.4e^{-3s}}{14.4s+1} \end{bmatrix} \begin{bmatrix} u_1 s \\ u_2 s \end{bmatrix} + \begin{bmatrix} \dfrac{3.8e^{-8s}}{14.9s+1} \\ \dfrac{4.9e^{-3s}}{13.2s+1} \end{bmatrix} w(s)$$

We first define the model in the **mod** format. The following commands use a sampling period of $T = 2$ time units (chosen arbitrarily):

```
delt = 2;
ny = 2;
g11 = poly2tfd(12.8,[16.7 1],0,1);
g21 = poly2tfd(6.6,[10.9 1],0,7);
g12 = poly2tfd(-18.9,[21.0 1],0,3);
g22 = poly2tfd(-19.4,[14.4 1],0,3);
umod = tfd2mod(delt,ny,g11,g21,g12,g22);
% Defines the effect of u inputs
g13 = poly2tfd(3.8,[14.9 1],0,8);
g23 = poly2tfd(4.9,[13.2 1],0,3);
dmod = tfd2mod(delt,ny,g13,g23);
% Defines the effect of w input
pmod = addumd(umod,dmod); % Combines the two models.
```

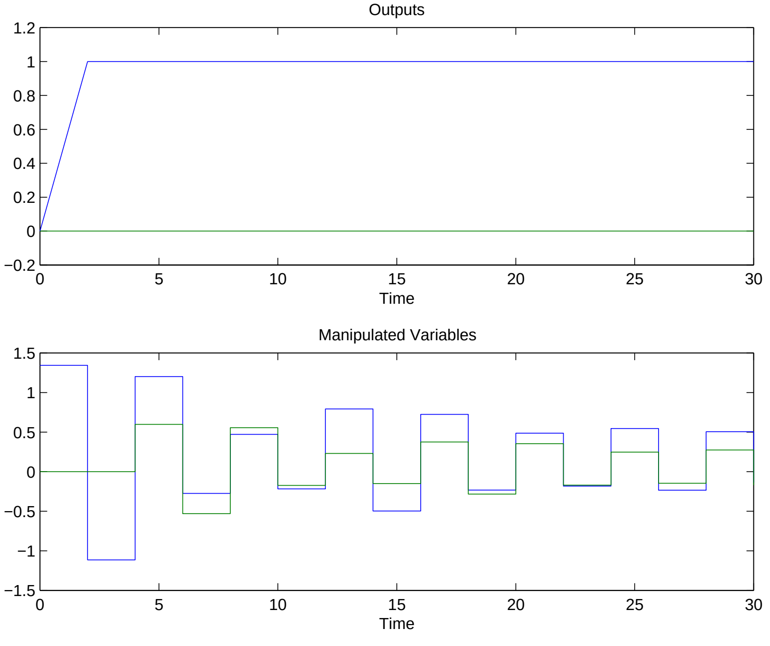
We now design an unconstrained MPC controller. The design parameters are essentially the same as for the functions based on step response models (see Chapter 2). In this case, start by choosing design parameters such that we get the *perfect controller*:

```
imod = pmod;% assume perfect modeling
ywt = [ ]; % default (unity) weights on both outputs
uwt = [ ]; % default (zero) weights on both inputs
P = 5; % prediction horizon
M = P; % control horizon
Ks = smpccon(imod,ywt,uwt,M,P);
```

We check the design by running a simulation for a step increase in the setpoint of output y_1:

```
tend=30; % time period for simulation.
r = [1 0]; % setpoints for the two outputs.
[y,u] = smpcsim(pmod,imod,Ks,tend,r);
plotall(y,u,delt)
```

Note that there is no model error since we used the same model to represent the *plant* (pmod) as that used to design the controller (imod). The results are:



Note that we get perfect tracking of the specified setpoint change ($y_1 = 1$, $y_2 = 0$), but the manipulated variables are *ringing*. You could have anticipated this by calculating the poles of the controller:

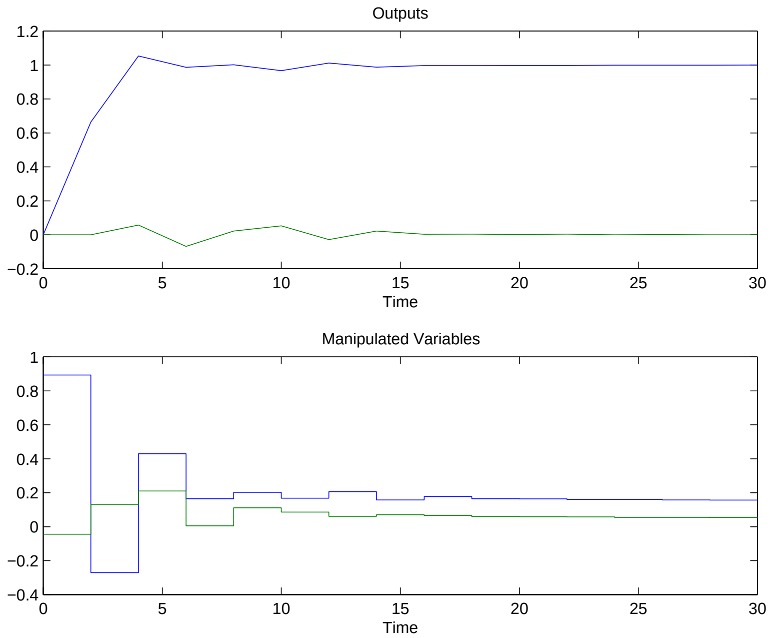
```
[clmod,cmod] = smpccl(pmod,imod,Ks);
smpcpole(cmod)
```

The result shows that one of the poles is at -0.9487 and another is at -0.9223. In general, such negative-real poles cause ringing.

One way to minimize ringing is to make the prediction horizon significantly larger than the control horizon:

```
P = 10;
M = 3;
Ks = smpccon(imod,ywt,uwt,M,P);
[y,u] = smpcsim(pmod,imod,Ks,tend,r);
plotall(y,u,delt)
```

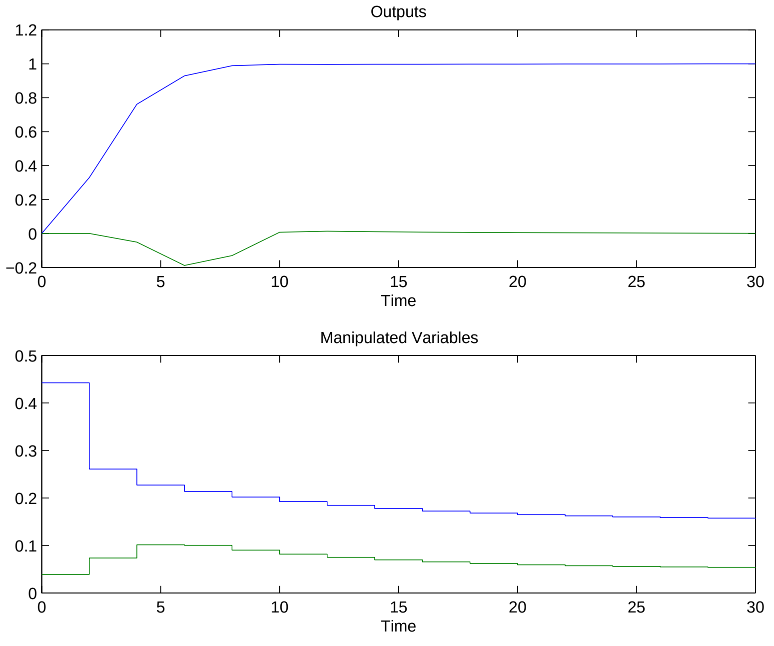
This results in the following improved responses:



Another (often more effective) way is to use *blocking*. In the case of blocking, each element of the vector M indicates the number of steps over which $\Delta u = 0$ during the optimization. For example, $M = [2\ 3]$ indicates that $u(k+1) = u(k)$ or $\Delta u(k+1) = 0$ and $u(k+4) = u(k+3) = u(k+2)$ (or $\Delta u(k+3) = \Delta u(k+4) = 0$):

```
M = [2 3 4]; % Defines 3 blocks of control moves
Ks = smpccon(imod,ywt,uwt,M,P);
[y,u] = smpcsim(pmod,imod,Ks,tend,r);
plotall(y,u,delt)
pause
```

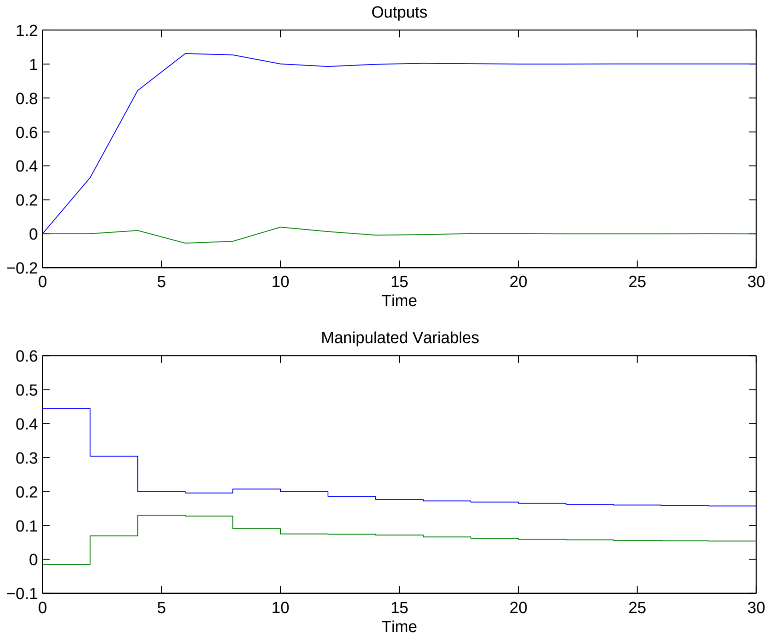
This completely eliminates the ringing, as shown in the following responses, at the expense of a more sluggish servo response and a larger disturbance in y_2.



A third approach is to increase the weights on the manipulated variables:

```
uwt = [1 1]; % increase input weighting
P = 5; % original prediction horizon
M = P; % original control horizon
Ks = smpccon(imod,ywt,uwt,M,P);
[y,u] = smpcsim(pmod,imod,Ks,tend,r);
plotall(y,u,delt)
```

for which the response is:

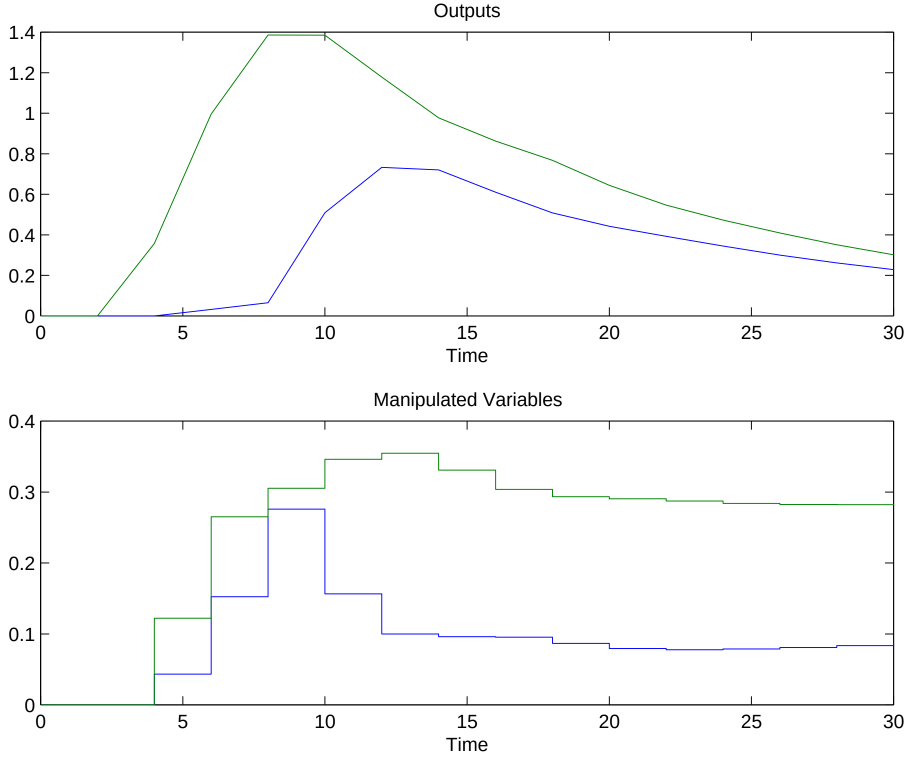


In general, you must choose the horizons and weights by trial-and-error, using simulations to judge their effectiveness.

The servo-response of the last controller looks good. Let's see how it responds to a unit-step in the unmeasured disturbance, $w(k)$:

```
ulim = [ ]; % default (no) constraints on u variables.
Kest = [ ]; % default (DMC) state estimator.
r = [0 0]; % Both output setpoints at zero.
z = [ ]; % default (zero) measurement noise.
v = [ ]; % default (zero) measured disturbances.
w = [1]; % unit-step in unmeasured disturbance.
[y,u] = smpcsim(pmod,imod,Ks,tend,r,ulim,Kest,z,v,w);
plotall(y,u,delt)
pause
```

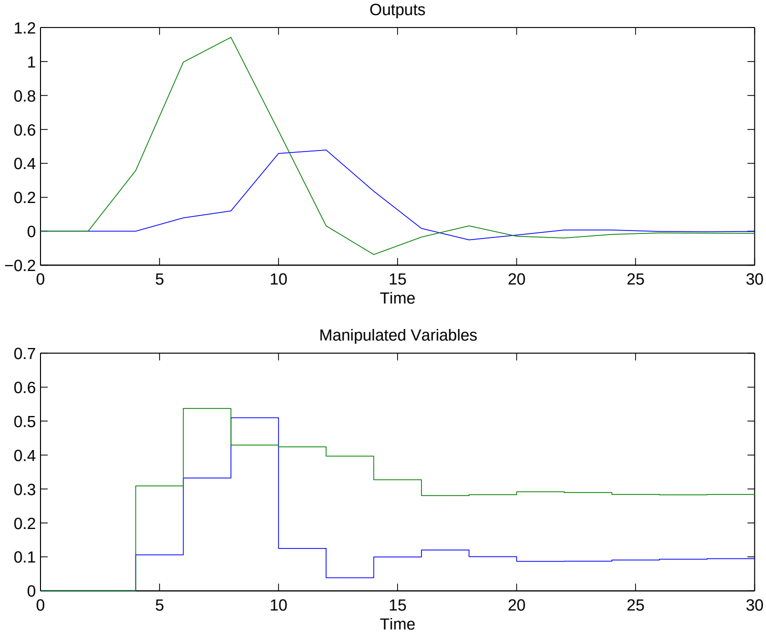
Note that both setpoints are zero. The resulting response is:



The regulatory response has a rather long transient. Let's see if we can improve it by using a state estimator other than the default (DMC) estimator:

```
[Kest,newmod] = smpcest(imod,[15 15],[3 3]);
Ks = smpccon(newmod,ywt,uwt,M,P);
[y,u] = smpcsim(pmod,newmod,Ks,tend,r,ulim,Kest,z,v,w);
plotall(y,u,delt)
```

See the detailed description of the `smpcest` function for a discussion of the estimator design parameters. The results show that the controller now compensates for the disturbance much more rapidly:



State-Space MPC with Constraints

The function scmpc handles problems with inequality constraints on the manipulated variables and/or outputs. The recommended procedure is to first use the tools described in the section section Unconstrained MPC Using State-Space Models to find values of the prediction horizon, P, control horizon, M, input and output weights, and a state-estimation strategy that work well for the unconstrained version of your problem. Then define the constraints and solve the problem using scmpc. The following example illustrates the use of scmpc.

Example: (see mpctutss.m)

We use the same example process as in the previous section, but use a sampling period of 1 and omit the unmeasured disturbance input:

```
T = 1;
g11 = poly2tfd(12.8,[16.7 1],0,1);
g21 = poly2tfd(6.6,[10.9 1],0,7);
g12 = poly2tfd(-18.9,[21.0 1],0,3);
g22 = poly2tfd(-19.4,[14.4 1],0,3);
imod = tfd2mod(2,T,g11,g21,g12,g22);
```

The following statements specify parameters required in both the constrained and unconstrained cases, and calculate the gain for the unconstrained controller.

```
nhor = 10; % Prediction horizon.
ywt = [ ]; % Unity weighting on output tracking errors
% (default).
uwt = [ ]; % Zero weighting on man. variable moves
% (default).
blks = [2 3 5]; % Allows 3 moves of manipulated variables.
K = [ ]; % DMC-type state estimation (default).
Ks = smpccon(imod,ywt,uwt,blks,nhor);
```

Let's first verify that the constrained and unconstrained solutions will be the same when the constraints are *loose* i.e. inactive. The following statements define upper and lower bounds on $u(k)$ at $-\infty$ and ∞, respectively, and bounds on $\Delta u(k)$ at 10 (both u_1 and u_2).[1] Also, bounds on $y(k)$ are set at the default values of $\pm\infty$.

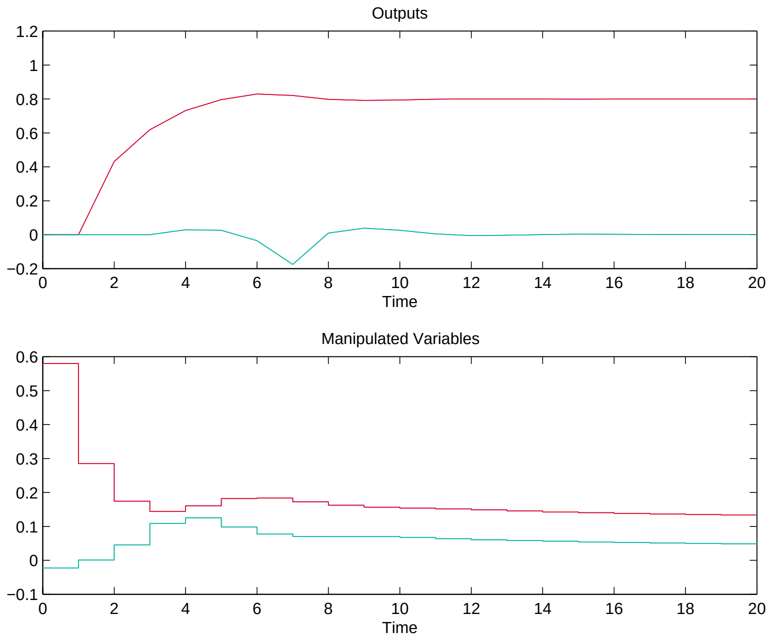
```
ulim = [-inf -inf inf inf 10 10];
ylim = [ ]; % Default -- no limits on outputs.
```

For the simulation we will make a step change of 0.8 in the setpoint for y_1. We will also assume a perfect model, i.e., use the same model for the plant as was used to design the controller.

```
setpts = [0.8 0]; % Define the step in the setpoint.
plant = imod;
tend = 20; % Duration of the simulation
[y1,u1] = smpcsim(plant,imod,Ks,tend,setpts,ulim,K);
[y,u] = scmpc(plant,imod,ywt,uwt,blks,nhor,tend,...
       setpts,ulim,ylim,K);
plotall([y y1],[u u1],T)
```

The above `plotall` command plots the results from `smpcsim` and `scmpc` on the same graph. Since the constraints were loose, there should be no difference. In the following plots, you can only distinguish two curves, i.e., the two simulations give the same values of y and u, as expected.

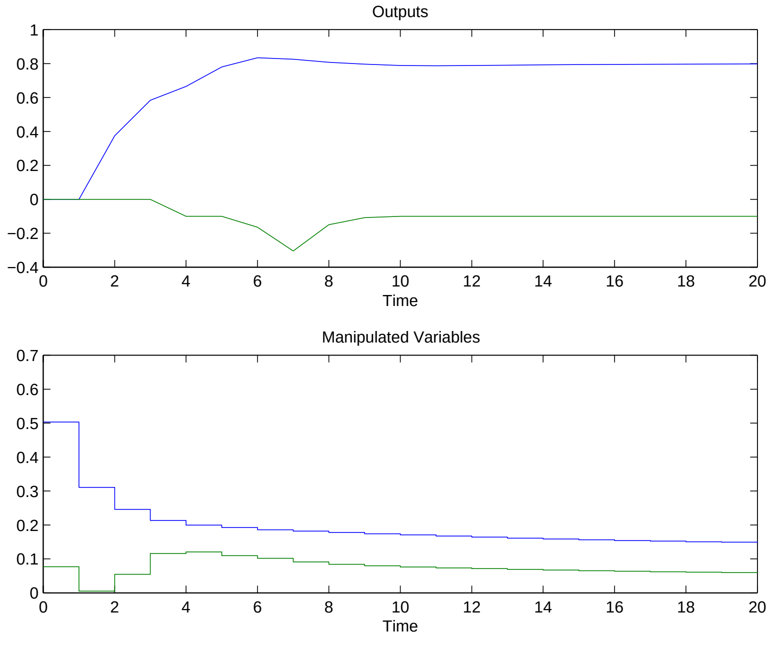
1. Finite bounds on Δu are required by `scmpc`. Here they are chosen large enough so that they have no effect.



Now let's add some constraints to the problem. Suppose we want the maximum value of y_2 to be -0.1. In the previous case it goes slightly above zero (dashed line in the *Outputs* plot). The following statements define a hard upper limit of y_2 = -0.1, *starting at the 4th step in the prediction horizon*. This accounts for the minimum delay of 3 sampling periods before y_2 can be affected by either u_1 or u_2, i.e., it is important to leave y_2 unconstrained for the first 3 steps in the prediction horizon. In this case, since the initial condition is $y_2 = 0$, it is *impossible* to make $y_2 \leq -0.1$ prior to $t = 4$. If you were to attempt to do so, you would get an error message stating that the problem is infeasible. Note also that the upper bound on y_2 supersedes the setpoint, which is still specified as zero. The controller thus maximizes the value of y_2 at steady state.

```
ylim = [-inf -inf inf inf
          -inf -inf inf inf
          -inf -inf inf inf
          -inf -inf inf -0.1];
[y,u] = scmpc(plant,imod,ywt,uwt,blks,nhor,tend,...
       setpts,ulim,ylim,K);
```

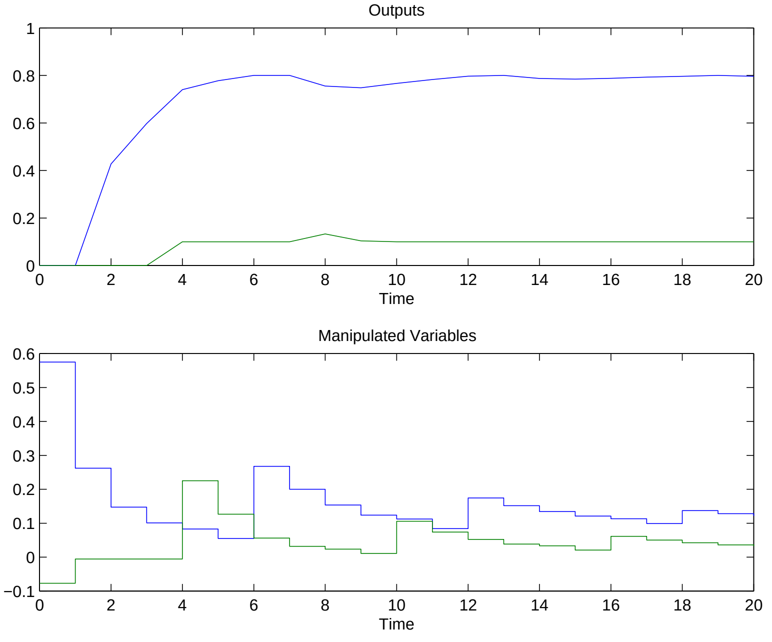
The following plot shows that the constraints are satisfied for $t \geq 4$, as expected.



If you use a long prediction horizon with constraints, the calculations can be time-consuming. You can minimize this by *turning off* constraints that are far out in the prediction horizon. The following example defines bounds on y_1 and y_2, then turns them off beyond the 4th point in the prediction horizon. The calculations are much faster than would be the case if only the first 4 rows of `ylim` had been used (try it). Also, since there is neither model error nor unmeasured disturbances, the solution satisfies all constraints for t 4 in any case. In general, output constraints must be chosen carefully to avoid infeasibilities and maximize the speed of the calculations.

```
ylim = [-inf -inf inf inf
           -inf -inf 0.8 inf
           -inf -inf inf inf
           -inf 0.10 inf inf]
           -inf -inf inf inf
% Turns off remaining bounds.
```

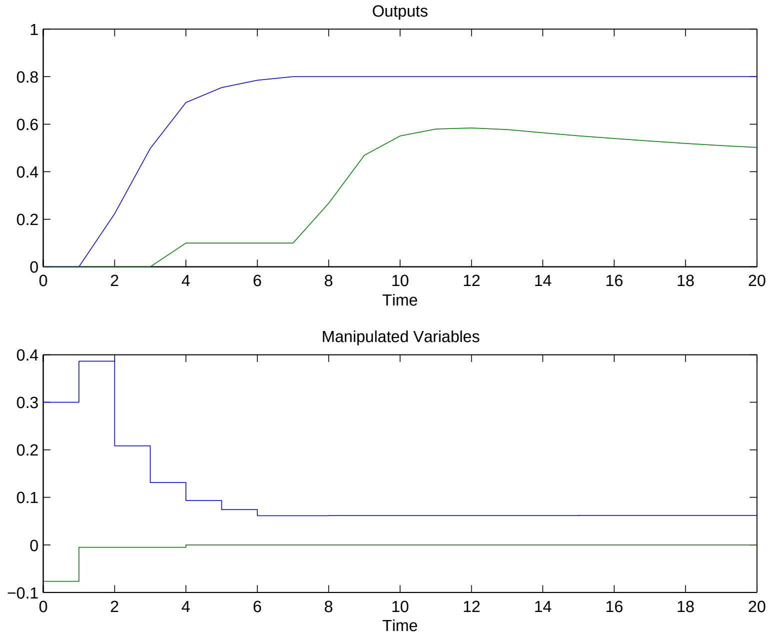
```
[y,u] = scmpc(plant,imod,ywt,uwt,blks,nhor,tend,...
       setpts,ulim,ylim,K);
plotall(y,u,T)
```



As a final example we impose bounds on the manipulated variables:

```
ulim = [-0.5 -0.5 0.5 0 0.3 0.3
          -inf -inf inf inf 0.3 0.3];
[y,u] = scmpc(plant,imod,ywt,uwt,blks,nhor,tend,...
      setpts,ulim,ylim,K);
plotall(y,u,T)
```

Again, to save computer time the constraints apply only for the first block in the prediction horizon, i.e., the constraints are turned off for periods 2 through $P = 10$. The following plot shows that the upper bound of $u_2 \leq 0$ and $\Delta u_1 \leq 0.3$ are the most restrictive. The former prevents y_2 from coming back to the minimum allowed value of 0.1.


Outputs
1
0.8
0.6
0.4
0.2
0
0 2 4 6 8 10 12 14 16 18 20
Time
Manipulated Variables
0.4
0.3
0.2
0.1
0
−0.1
0 2 4 6 8 10 12 14 16 18 20
Time

Application: Paper Machine Headbox Control

Ying et al. (1992)[2] studied the control of composition and liquid level in a paper machine headbox, a schematic of which is shown in Figure 3-1. The process model is given by a set of ordinary differential equations (ODEs) in bilinear form. Using their nomenclature, the states are $x^T = [H_1\ H_2\ N_1\ N_2]$, where H_1 is the liquid level in the feed tank, H_2 is that in the headbox, N_1 is the consistency (percentage of pulp fibers in suspension) in the feed tank, and N_2 is that in the headbox. All states except H_1 are measured, i.e., the measured outputs are $y^T = [H_2\ N_1\ N_2]$. The primary control objective is to hold H_2 and N_2 (y_1 and y_3) at specified setpoints.

There are two manipulated variables: $u^T = [G_p\ G_w]$, where G_p is the flowrate of stock entering the feed tank, and G_w is that of the recycled *white water*. There is a single measured disturbance: $v = [N_p]$, the consistency of the stock entering the feed tank, and a single unmeasured disturbance: $d = [N_w]$, the consistency of the white water. All variables are normalized such that they are zero at the nominal steady state, and variations about the steady-state are of the same order of magnitude. The process is open-loop stable.

2. Ying, Y., M. Rao, and Y. Sun, “Bilinear Control Strategy for Paper-Making Process,” *Chem. Eng. Comm*. 1992, 111, 13–28.

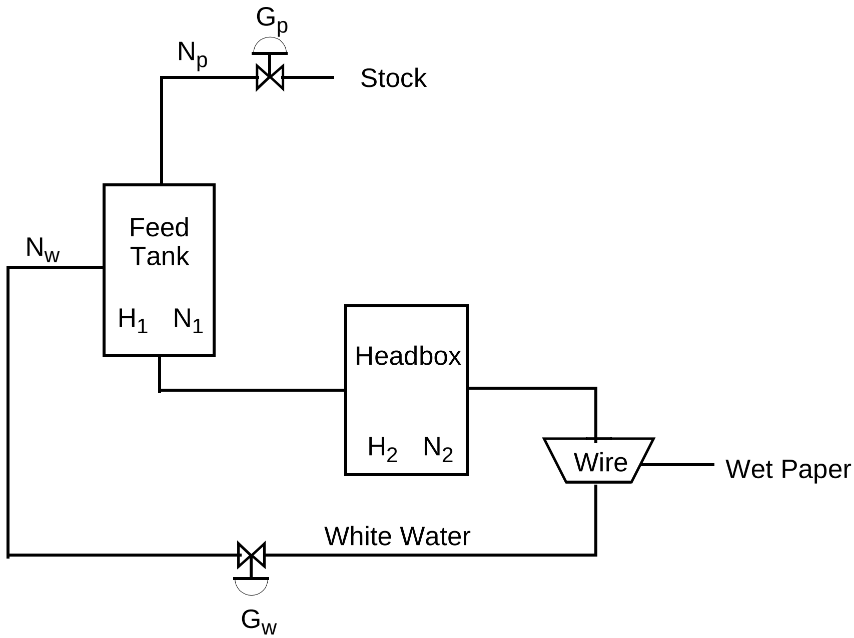


Figure 3-1 Schematic of Paper Machine Headbox Control Problem

MPC Design Based on Nominal Linear Model

The standard MPC design methods require a linear model of the plant. We therefore linearize the bilinear model at the nominal steady-state condition ($x = 0$; $u = 0$; v = 0; $d = 0$). Since the model is simple, one can linearize it analytically to obtain:

$$\dot{x}_m = Ax_m + B_0 u + B_v v + B_d d_m$$

where x_m, y_m, and d_m are the model states, outputs, and disturbances, respectively. The desired closed-loop response time is of the order of 10 minutes, so we choose a sampling period of $T_s = 2$ minutes. The file `pm_lin.m` in the directory `mpcdemos` contains the code for all the computations in this section of the manual. The following commands define the linear model and plot the response of the outputs to a unit step in each manipulated variable:

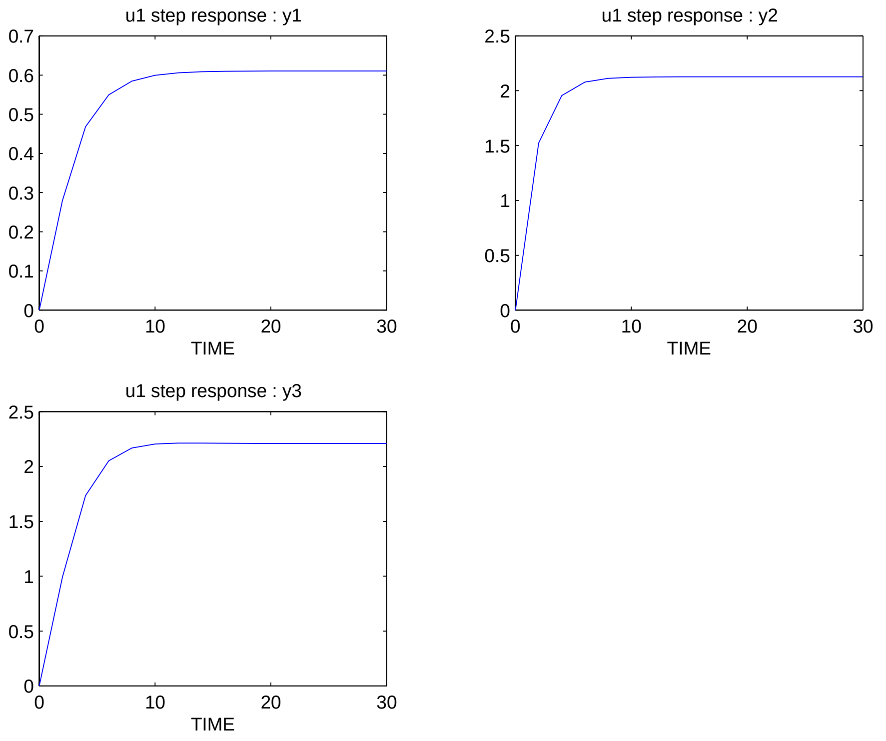


Figure 3-2 Responses of Paper Machine Outputs to Unit Step in u_1

```
% Matrices= of the linearized paper machine model

A = [-1.93 0 0 0; .394 -.426 0 0; 0 0 .63 0; .82 -.784
   .413 -.426];
B = [1.274 1.274 0 0;0 0 0 0;1.34 -.65 .203 .406;0 0 0 0];
C = [0 1 0 0; 0 0 1 0; 0 0 0 1];
D = zeros(3,4);

% Discretize the linear model and save in mod form.

dt = 2;
```

```
[PHI,GAM] = c2dmp(A,B,dt);
minfo = [dt,4,2,1,1,3,0];
imod = ss2mod(PHI,GAM,C,D,minfo);
plotstep(mod2step(imod,30))
```

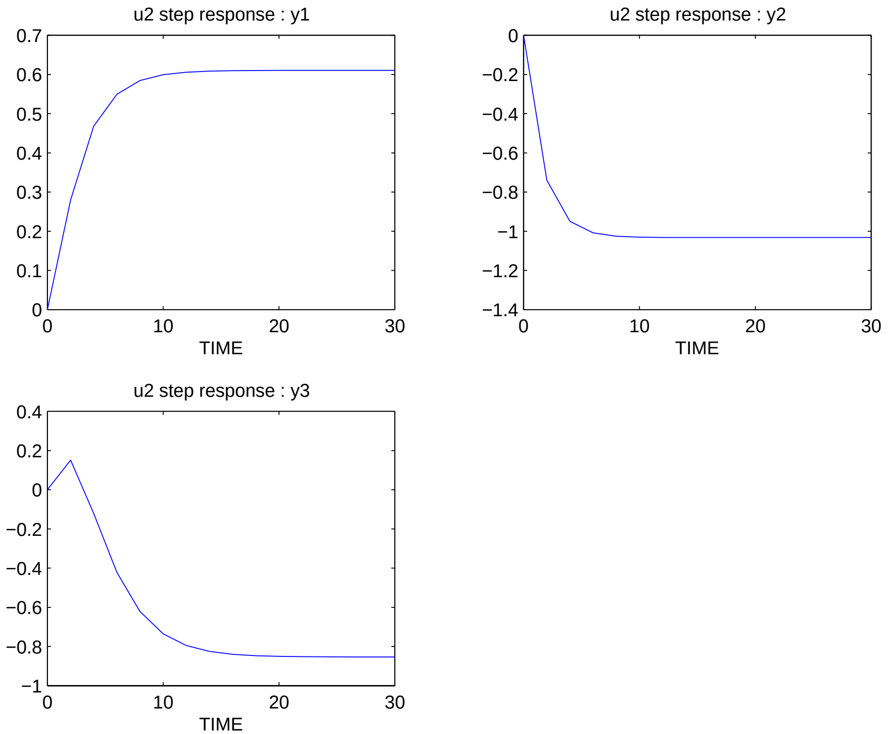


Figure 3-3 Responses of Paper Machine Outputs to Unit Step in u_2

The step responses (Figure 3-2 and Figure 3-3) show that there are large interactions in the open-loop system. Adjustments in u_1 and u_2 have strong effects on both y_1 and y_3. Also, the $u_2 \rightarrow y_3$ step exhibits an inverse response. We begin by attempting a controller design for good response of both y_1 and y_3:

```
% Define controller parameters

P = 10; % Prediction horizon
M = 3; % Control horizon
ywt = [1,0,1]; % Equal weighting of y(1) and y(3),
% no control of y(2)
uwt = 0.6*[1 1]; % Equal weighting of u(1) and u(2).
ulim = [-10*[1 1] 10*[1 1] 2*[1 1]];% Constraints on u
ylim = [ ]; % No constraints on y
Kest = [ ]; % Use default estimator

% Simulation using scmpc -- no model error

pmod=imod; % plant and internal model are identical
setpts = [1 0 0];
% servo response to step in y(1) setpoint
tend = 30; % duration of simulation
[y,u,ym] = scmpc(pmod,imod,ywt,uwt,M,P,tend,...
   setpts,ulim,ylim,Kest);
plotall(y,u,dt)
```

The prediction horizon of 10 sampling periods (20 minutes) extends well past the desired closed-loop response time. Preliminary trials suggested that longer horizons increased the computational load but provided no advantage in setpoint tracking. The use of M < P is not required in this case, but helps to reduce the computational load and inhibit ringing of the manipulated variables. Note the equal penalties on setpoint tracking errors for y_1 and y_3 (ywt variable), reflecting our desire to track both setpoints accurately. There is no penalty on y_2, since it does not have a setpoint. The listed uwt penalties were determined by running several trials. Figure 3-4 shows smooth control of y_1 with the desired 10-minute response time, but there is a noticeable disturbance in y_3.

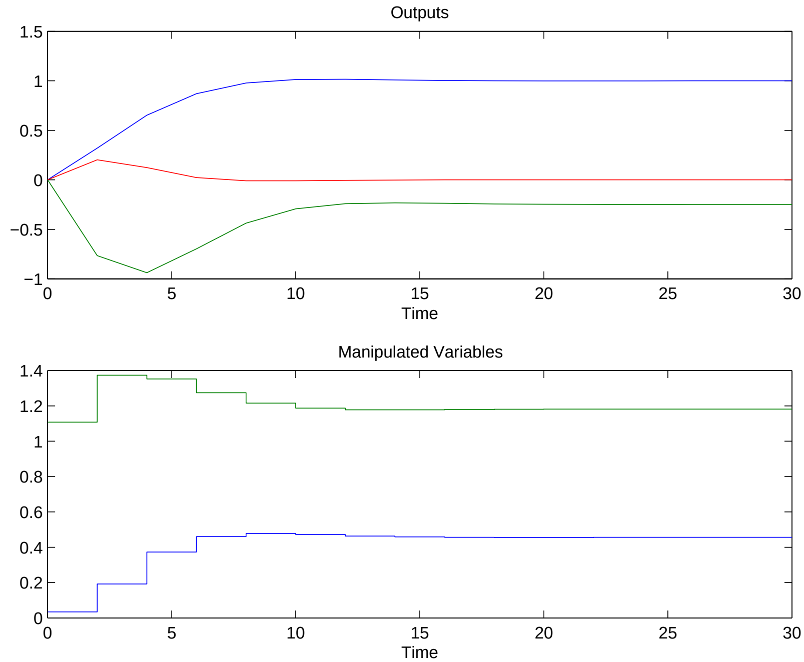


Figure 3-4 Response of closed-loop system to unit step in y_1 setpoint for equal output weighting. Output y_2 is uncontrolled, and the y_3 setpoint is zero

One could repeat the simulation for a step change in the y_3 setpoint as follows:

```
setpts = [0 0 1]; % servo response to step in y(3) setpoint
[y,u,ym] = scmpc(pmod,imod,ywt,uwt,M,P,tend,...
   setpts,ulim,ylim,Kest);
plotall(y,u,dt)
```

Normally, control of consistency (y_3) is more important than control of liquid level, y_1. We can achieve better control of y_3 if we allow larger tracking errors in y_1. For example, an alternative controller design uses unequal weights on the controlled outputs:

```
ywt = [0.2,0,1]; % Unequal weighting of y(1) and y(3),
% no control of y(2)
setpts = [1 0 0];
% servo response to step in y(1) setpoint
[y,u,ym] = scmpc(pmod,imod,ywt,uwt,M,P,tend,...
   setpts,ulim,ylim,Kest);
plotall(y,u,dt)
```

As shown in Figure 3-5, a y_1 setpoint change causes a much smaller disturbance in y_3 than before (compare with Figure 3-4). The disadvantage is that the response time of y_1 has increased from about 8 to 25 minutes. Similarly, a step change in the y_3 setpoint would cause a larger disturbance in y_1 than in the original design. Overall, however, the controller with unequal weighting gives better nominal performance and will be used as the basis for subsequent designs.

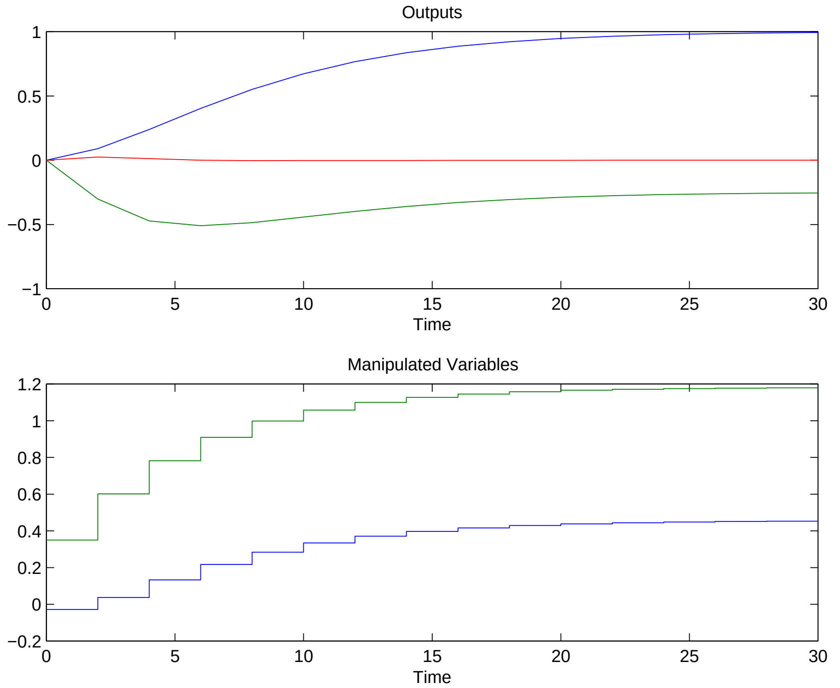


Figure 3-5 Response of closed-loop system to Unit step in y_1 setpoint for unequal output weighting. Output y_2 is uncontrolled, and the y_3 setpoint is zero.

We now evaluate the response of the above controller to a unit step in the measured disturbance, v (i.e., feedforward control). The commands required for this are:

```
setpts = [0 0 0]; % output
setpoints z = [ ]; % measurement noise
v = 1; % measured disturbance
d = 0; % unmeasured disturbance [y,u,ym] =
scmpc(pmod,imod,ywt,uwt,M,P,tend,...
   setpts,ulim,ylim,Kest,z,v,d);
plotall(y,u,dt)
```

As shown in Figure 3-6, both controlled outputs are held near their setpoints, with larger deviations in y_1, as expected.

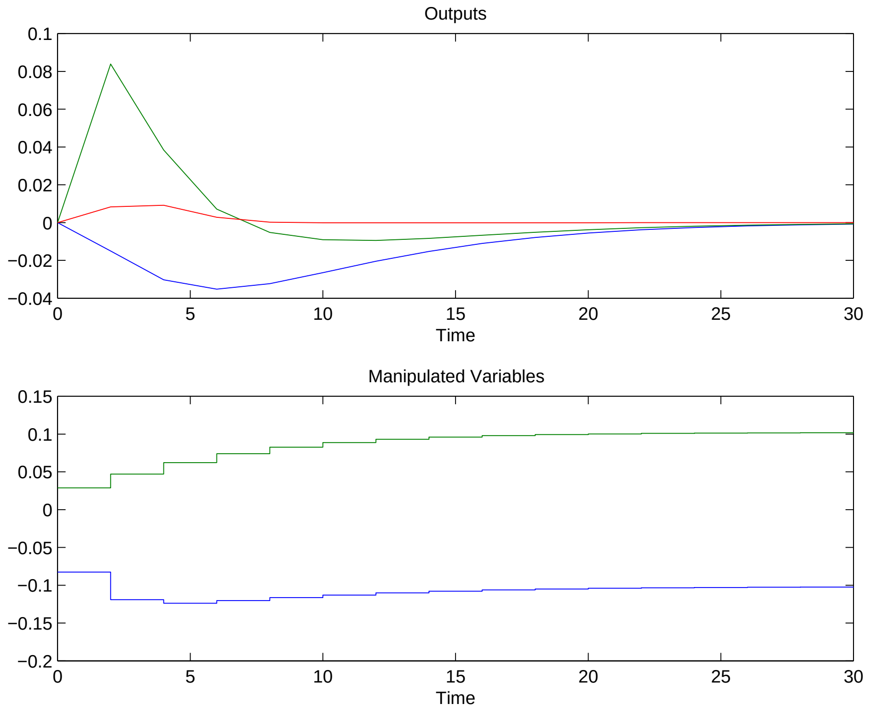


Figure 3-6 Response of closed-loop system to unit step in measured disturbance, *v*. unequal output weighting with y_1 and y_3 setpoints at zero.

Finally, we check the response to a step in the unmeasured disturbance. The required commands are:

```
setpts = [0 0 0]; % output setpoints
v = 0; % measured disturbance
d = 1; % unmeasured disturbance
[y,u,ym] = scmpc(pmod,imod,ywt,uwt,M,P,tend,...
   setpts,ulim,ylim,Kest,z,v,d);
plotall(y,u,dt)
```

As shown in Figure 3-7, the unmeasured disturbance causes significant deviations in both controlled outputs. In fact, the higher-priority output, y_3, exhibits the larger tracking error of the two controlled variables.

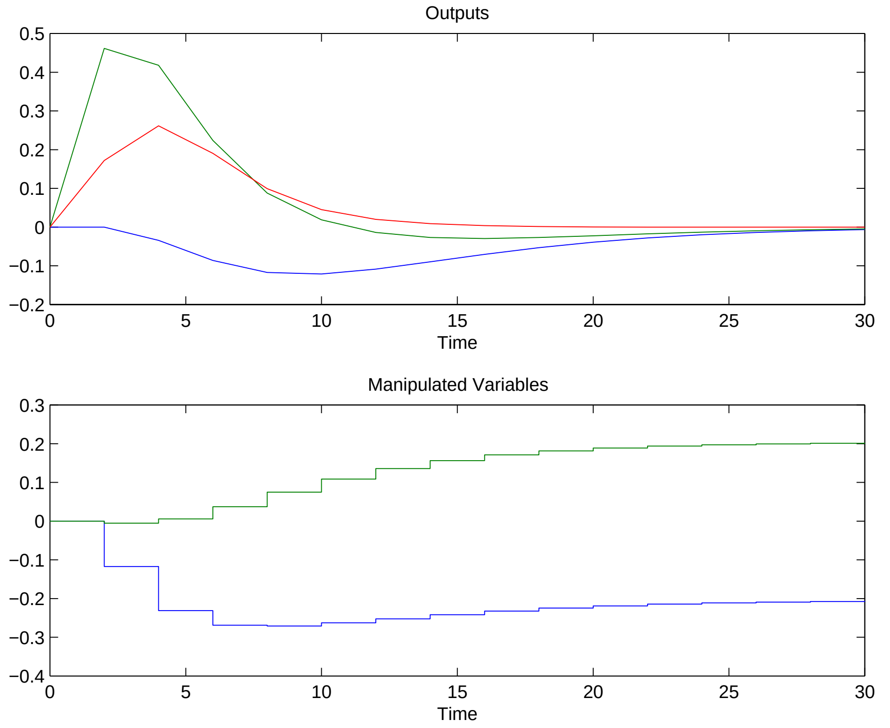


Figure 3-7 Response of closed-loop system (with default state estimator) to unit step in unmeasured disturbance, d. unequal output weighting with y_1 and y_3 setpoints at zero.

The closed-loop response to unmeasured disturbances can often be improved by a change in the state estimator. In the previous trials, we were using the default estimator, which assumes that disturbances are independent, random steps at each output. In fact, the *known* unmeasured disturbance, d, has no effect on y_1, and its effects on y_2 and y_3 are approximately first order with time constants of 3 and 5 minutes, respectively. One way to exploit this knowledge is to specify an expected covariance for d and a measurement noise covariance for y, then use the Kalman gain for the modeled disturbance characteristics.

Consider the following sequence of commands, leading to the responses shown in Figure 3-8:

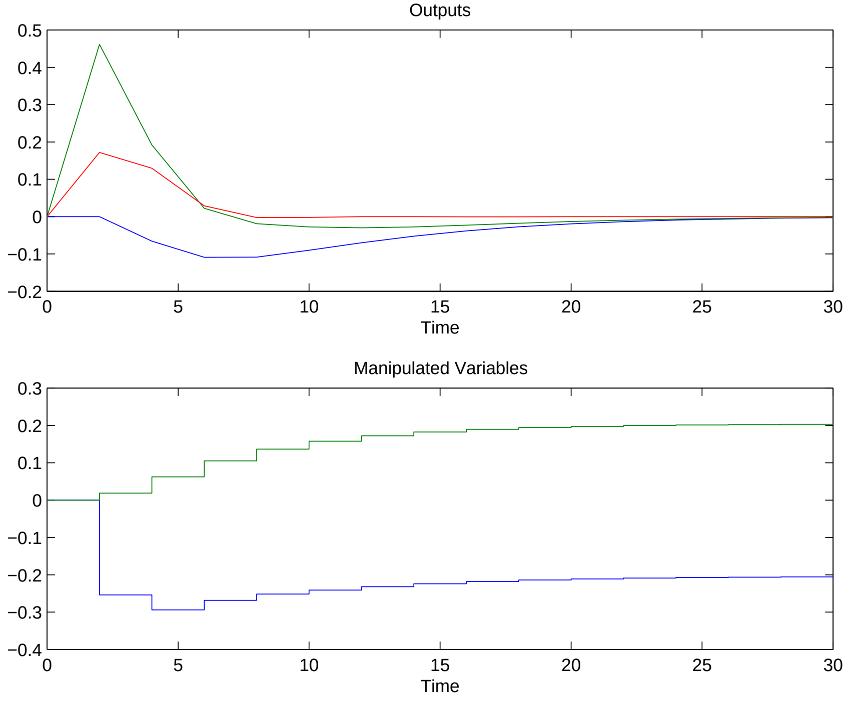


Figure 3-8 Response of closed-loop system to unit step in unmeasured disturbance, d. with Kalman Estimator, unequal output weighting with y_1 and y_3 setpoints at zero.

```
% Estimator design
Q = 30;
R = 1*eye(3);
Kest = smpcest(imod,Q,R);
% Simulation using scmpc -- no model error
setpts = [0 0 0]; % servo response to step in y1 setpoint
d = 1; % unmeasured disturbance
[y,u,ym] = scmpc(pmod,imod,ywt,uwt,M,P,tend,...
  setpts,ulim,ylim,Kest,z,v,d);
plotall(y,u,dt)
```

The resulting Kest matrix is:

```
Kest =
   0   00
   0.0000 -0.00000.0000
   0.0000 0.74460.0732
   -0.0000 0.29850.0851
   0.0000 -0.00000.0000
   -0.0000 0.77770.2934
   0.0000 0.29340.1977
```

We have specified equal measurement noise for each output (`R` is diagonal with a rank equal to the number of outputs, and equal elements on the diagonal). This makes the Kalman estimator give equal weight to each output measurement.[3] The dimension of `Q` must equal the number of elements in d (unity in this case). A relatively large value of `Q(i,i)` signifies an important disturbance mode. In practice, the elements of `Q` and `R` are tuning parameters, and one adjusts the relative magnitudes to achieve the desired balance of fast disturbance rejection (usually promoted by making Q relatively large) and robustness.

For the chosen `Q` and `R`, and the disturbance model in `imod`, the elements of column 1 of `Kest` (shown above) are essentially zero. Thus, the measurement of y_1 provides no information regarding the effect of d on the process states. Output y_2, on the other hand, provides large corrections to the state estimates. If it were not available, rejection of d would degrade.[4]

Figure 3-8 shows that although the revised estimator reduced the disturbance in y_3, it is still significant (compare to Figure 3-7). A key limiting factor is the use of a 2-minute sampling period. As shown in Figure 3-8, the controller does not respond to the disturbance until it is first detected at $t = 2$ minutes. You can verify that reducing the sampling period to 0.25 minutes (holding all other parameters constant) greatly reduces the disturbance in y_3. Such a change would also speed up the setpoint tracking in the nominal case. It may cause robustness problems, however, so we defer further consideration of the sampling period to tests with the nonlinear plant (see next section).

This application is unusual in that the characteristics of the unmeasured disturbance are known. When this is not the case, the *output disturbance* form

3. If a measurement were known to be inaccurate, its `R(i,i)` value should be relatively large.
4. You can see how serious the degradation would be by setting `R(2,2)` to a large value, e.g.,10000.

of the estimator simplifies the design procedure. It requires only a rough idea of the characteristic times for the disturbances, and the signal-to-noise ratio for each output. For example, you can verify that the following design rejects the *d* disturbance almost as well as the *optimal* Kalman design:

```
% Alternative estimator design -- output disturbances
taus = [5 5 5];
signoise = [10 10 10];
[Kest, newmod] = smpcest(imod,taus,signoise);
% Simulation using scmpc -- no model error
[y,u,ym] = scmpc(pmod,newmod,ywt,uwt,M,P,tend,...
    setpts,ulim,ylim,Kest,z,v,d);
plotall(y,u,dt)
```

MPC of Nonlinear Plant

We are now ready to test the controller design on the *real* (nonlinear) plant. A special version of the `scmpc` function (called `scmpcnl`) is available for this purpose. It uses a nonlinear plant model in the S-function format required by Simulink. (See the Simulink documentation for more information on how to write such models.) The model of the paper machine is in the file `pap_mach.m`. Simulations with Simulink involving nonlinear models usually take much longer (by an order of magnitude) than linear simulations of a plant of comparable complexity. This is especially likely if the plant model is in the form of an .M file, as is the case here. If such models are to be used extensively, it may be worthwhile to code them as a `.mex` file (see MATLAB documentation). To see how well the MPC design rejects the *d* disturbance of Figure 3-8, we could use the commands found in the file `pm_nonl.m` in the directory `mpcdemos`. The only differences between these commands and those for the original linear simulation are:

- We have defined the initial values of the plant state and manipulated variables (`x0` and `u0`, respectively).
- A step size for numerical integration has been specified. The value of 0.05 minutes provides reasonable accuracy in this application. In general, one must choose the step size to fit the problem (or use a variable step-size integration method, as provided by Simulink).

You can verify that the results are nearly identical to those shown in Figure 3-8. In other words, the nonlinearities in the plant have caused negligible performance degradation. Very similar results are also obtained for the setpoint change of Figure 3-4.

As the magnitude of the disturbance (or setpoint change) increases, nonlinear effects become significant. For example, Figure 3-9 is for a step in d of 7 units. If the plant were linear, the curves in Figure 3-9 would be the same shape as those in Figure 3-8, but scaled by a factor of 7. Although this is approximately true, there are some qualitative differences. For example, at t = 8 minutes in Figure 3-9, y_2 has gone below y_1, whereas in Figure 3-8, $y_2 > y_1$ at all times.

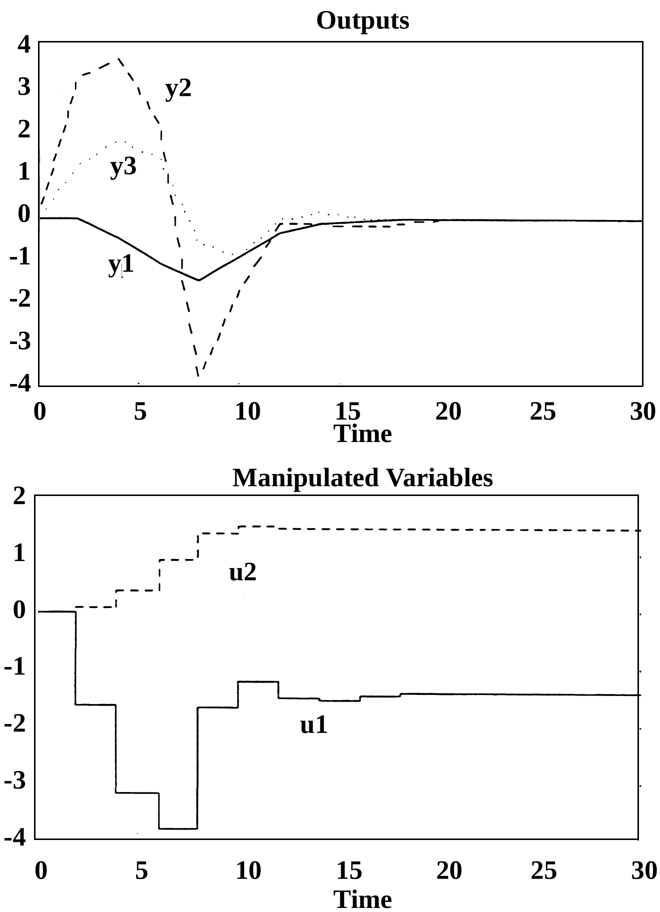


Figure 3-9 As for Figure 3-8, but With Nonlinear Plant, and Step in *d* of 7 Units

If d is increased to 8, control quality degrades dramatically and the maximum tracking error in y_3 goes to about -10^5 (not shown). This is caused by changes in the plant characteristics as it moves away from the nominal state (i.e., causing errors in the MPC's linear model).

Sensitivity to modeling error can often be reduced by *de-tuning* the controller. A common approach is to increase the magnitudes of the `uwt` parameters. When nonlinear effects are severe, however, it may be impossible for *any* time-invariant, linear controller to provide stable, offset-free performance. In that case, if the nonlinear effects are predictable, one might try MPC based on a nonlinear model (e.g., Gattu and Zafiriou, 1992).[5] Scripts for this purpose can be developed using the functions in this toolbox.

As a final test, let's repeat the simulation of Figure 3-8 with a controller sampling period of 0.25 minutes (recall that the original sampling period was 2 minutes). Results appear in Figure 3-10. Compared to Figure 3-8, which had no model error (i.e., linear plant), we reduced the disturbance in y_3 by a factor of 3. Thus, a reduction in sampling period may not lead to robustness problems, and should be tested more thoroughly. You can verify that it works well for other combinations of *small* disturbances and setpoint changes.

5. Gattu, G. and E. Zafiriou, "Nonlinear Quadratic Dynamic Matrix Control with State Estimation," *Ind. Eng. Chem. Research,* 1992, 31, 1096–1104.

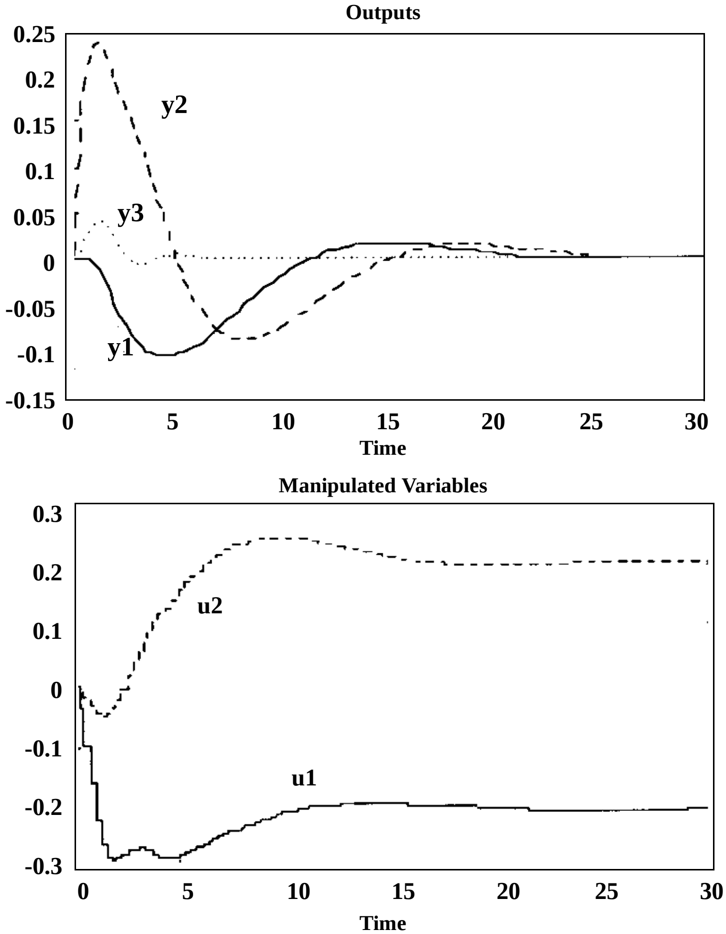


Figure 3-10 As for Figure 3-8, ($d = 1$) but With Nonlinear Plant, Sampling Period of 0.25 Minutes

4

Command Reference

Commands Grouped by Function

Identification	
`autosc`	Automatically scales a matrix by its means and standard deviations.
`imp2step`	Combines MISO impulse response models to form MIMO models in MPC **step** format.
`mlr`	Calculates MISO impulse response model via multi-variable linear regression.
`plsr`	Calculates MISO impulse response model via partial least squares regression.
`rescal`	Converts scaled data back to its original form.
`scal`	Scales a matrix by specified means and standard deviations.
`validmod`	Validates a MISO impulse response model using new data.
`wrtreg`	Writes data matrices used for regression.

Plotting and Matrix Information	
`mpcinfo`	Outputs matrix type and attributes of system representation.
`plotall`	Plots outputs and inputs from a simulation run on one graph.
`plotfrsp`	Plots the frequency response of a system as a Bode plot.
`ploteach`	Makes separate plots of outputs and/or inputs from a simulation run.
`plotstep`	Plots the coefficients of a model in MPC **step** form.

Model Conversions	
`c2dmp`	Converts state-space model from continuous time to discrete-time. (Equivalent to `c2d` in Control System Toolbox)
`cp2dp`	Converts from a continuous to a discrete transfer function in **poly** format.
`d2cmp`	Converts state-space model from discrete-time to continuous time. (Equivalent to `d2c` in Control System Toolbox)
`mod2mod`	Changes sampling period of a model in MPC **mod** format.
`mod2ss`	Converts a model in MPC **mod** format to a state-space model.
`mod2step`	Converts a model in MPC **mod** format to MPC **step** format.
`poly2tfd`	Converts a transfer function in **poly** format to MPC *tf* format.
`ss2mod`	Converts a state-space model to MPC **mod** format.
`ss2step`	Converts a state-space model to MPC **step** format.
`ss2tf2`	Converts state-space model to transfer function. (Equivalent to `ss2tf` in Control System Toolbox)
`tf2ssm`	Converts transfer function to state-space model. (Equivalent to `tf2ss` in Control System Toolbox)
`tfd2mod`	Converts a model in MPC **tf** format to MPC **mod** format.
`tfd2step`	Converts a model in MPC **tf** format to MPC **step** format.
`th2mod`	Converts a model in **theta** format (System Identification Toolbox) into MPC **mod** format.

Model Building — MPC mod format	
`addmd`	Adds one or more measured disturbances to a plant model.
`addmod`	Combines two models such that the output of one adds to the input of the other.
`addumd`	Adds one or more unmeasured disturbances to a plant model.
`appmod`	Appends two models in an unconnected, parallel structure.
`paramod`	Puts two models in parallel such that they share a common output.
`sermod`	Puts two models in series.

Controller Design and Simulation — MPC step format	
`cmpc`	Solves the quadratic programming problem to simulate performance of a closed-loop system with input and output constraints.
`mpccl`	Creates a model in MPC **mod** format of a closed-loop system with an unconstrained MPC controller.
`mpccon`	Calculates the unconstrained controller gain matrix for MPC.
`mpcsim`	Simulates a closed-loop system with optional saturation constraints on the manipulated variables.
`nlcmpc`	Simulink S-function block for MPC controller with input and output constraints (solves quadratic program).
`nlmpcsim`	Simulink S-function block for MPC controller with optional saturation constraints.

Controller Design and Simulation — MPC mod format	
scmpc	Solves the quadratic programming problem to simulate performance of a closed-loop system with input and output constraints.
smpccl	Creates a model in MPC **mod** format of a closed-loop system with an unconstrained MPC controller.
smpccon	Calculates the unconstrained controller gain matrix for MPC.
smpcest	Designs a state estimator for use in MPC.
smpcsim	Simulates a closed-loop system with optional saturation constraints on the manipulated variables.

Analysis	
mod2frsp	Calculates frequency response for a system in MPC **mod** format.
smpcgain	Calculates steady-state gain matrix of a system in MPC **mod** format.
smpcpole	Calculates poles of a system in MPC **mod** format.
svdfrsp	Calculates singular values of a frequency response.

Utility Functions	
`abcdchkm`	Checks dimensional consistency of (A,B,C,D) set. (Equivalent to `abcdchk` in Control System Toolbox)
`dantzgmp`	Solves quadratic programs.
`dareiter`	Solves discrete Riccati equation by an iterative method.
`dimpulsm`	Generates impulse response of discrete-time system. (Equivalent to `dimpulse` in Control System Toolbox)
`dlqe2`	Calculates state-estimator gain matrix for discrete systems.
`dlsimm`	Simulates discrete-time systems. (Equivalent to `dlsim` in Control System Toolbox)
`mpcaugss`	Augments a state-space model with its outputs.
`mpcparal`	Puts two state-space models in parallel.
`nargchkm`	Checks number of M-file arguments. (Equivalent to `nargchk` in Control System Toolbox)
`mpcstair`	Creates the *stairstep* format used to plot manipulated variables.
`vec2mat`	Converts a vector to a matrix.

Purpose

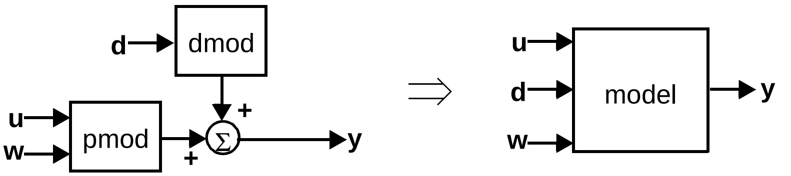
Adds one or more *measured* disturbances to a plant model in the MPC **mod** format. Used to allow for feedforward compensation in MPC.

Syntax

```
model = addmd(pmod,dmod)
```

Description

The disturbance model contained in `dmod` adds to the plant model contained in `pmod` to form a composite, model, with the structure given in the following block diagram:



`pmod`, `dmod` and `model` are in the MPC **mod** format (see `mod` in the online *MATLAB Function Reference* for a detailed description). You would normally create `pmod` and `dmod` using either the `tfd2mod`, `ss2mod` or `th2mod` functions.

`addmd` is a specialized version of `paramod`. Its main advantage over `paramod` is that it assumes all the inputs to `dmod` are to be measured disturbances. This saves you the trouble of designating the input types in a separate step.

Example

See `ss2mod` for an example of the use of this function.

Algorithm

`addmd` converts `pmod` and `dmod` into their state-space form, then uses the `mpcparal` function to build the composite model.

Restrictions

- `pmod` and `dmod` must have been created with equal sampling periods and number of output variables.
- `pmod` must not include measured disturbances, i.e., its **mod** format must specify $n_d = 0$.
- All inputs to `dmod` must be classified as manipulated variables. (They will be reclassified automatically as measured disturbances in `model`.) So the **mod** format of `dmod` must specify $n_d = n_w = 0$ (which is the default for all model creation functions).

See Also

`addmod`, `addumd`, `appmod`, `paramod`, `sermod`

addmod

Purpose

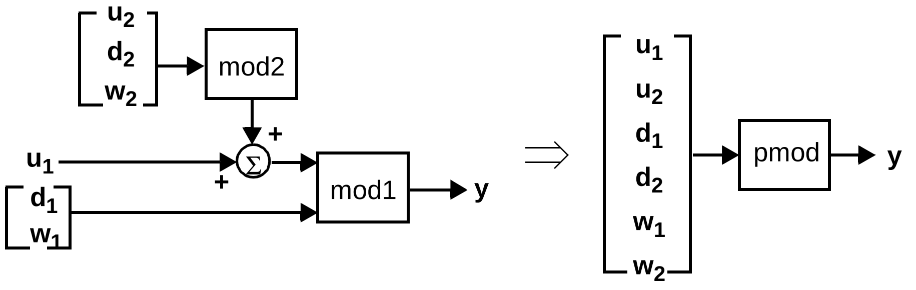
Combines two models in the MPC **mod** format such that the output of one combines with the *manipulated* inputs of the other. This function is specialized and rarely needed. Its main purpose is to build up a model of a complex structure that includes the situation shown in the diagram below.

Syntax

```
pmod = addmod(mod1,mod2)
```

Description

The output(s) of `mod2` add to the manipulated variable(s) of `mod1` to form a composite system, `pmod`, with the structure given in the following block diagram:



`pmod`, `mod1` and `mod2` are in the MPC **mod** format (see `mod` in the online *MATLAB Function Reference* for a detailed description). You would normally create `mod1` and `mod2` using either the `tfd2mod`, `ss2mod` or `th2mod` functions.

The different input *types* associated with `mod1` and `mod2` will be retained in `pmod` and will be ordered as shown in the diagram.

Example

See `mod2ss` for an example of the use of this function.

Restrictions

- `mod1` and `mod2` must have been created with equal sampling periods.
- The number of manipulated variables in `mod1` must equal the number of output variables in `mod2`.

See Also

`addmd`, `addumd`, `appmod`, `paramod`, `sermod`

Purpose

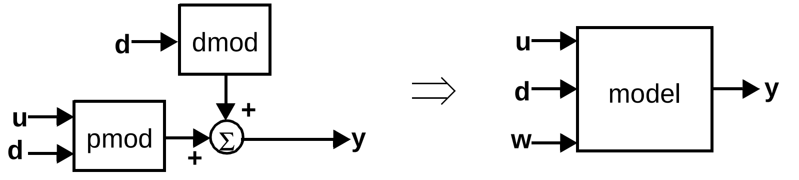
Adds one or more *unmeasured* disturbances to a plant model in MPC **mod** format. Used for simulation of disturbances and for design of state estimators in MPC.

Syntax

```
model = addumd(pmod,dmod)
```

Description

The disturbance model contained in `dmod` adds to the plant model contained in `pmod` to form a composite, `model`, with the structure given in the following block diagram:



`pmod`, `dmod` and `model` are in the MPC **mod** format (see `mod` in the online *MATLAB Function Reference* for a detailed description). You would normally create `pmod` and `dmod` using either the `tfd2mod`, `ss2mod` or `th2mod` functions.

`addumd` is a specialized version of `paramod`. Its main advantage over `paramod` is that it assumes all the inputs to `dmod` are to be unmeasured disturbances. This saves you the trouble of designating the input types in a separate step.

Example

See `ss2mod` for an example of the use of this function.

Algorithm

`addumd` converts `pmod` and `dmod` into their state-space form, then uses the `mpcparal` function to build the composite model.

Restrictions

- `pmod` and `dmod` must have been created with equal sampling periods and number of output variables.
- `pmod` must not include unmeasured disturbances, i.e., its **mod** format must specify $n_w = 0$.
- All inputs to `dmod` must be classified as manipulated variables. (They will be reclassified automatically as unmeasured disturbances in `model`.) So the **mod** format of `dmod` must specify $n_d = n_w = 0$ (which is the default for all model creation functions).

See Also

`addmod`, `addmd`, `appmod`, `paramod`, `sermod`, `smpcest`

appmod

Purpose

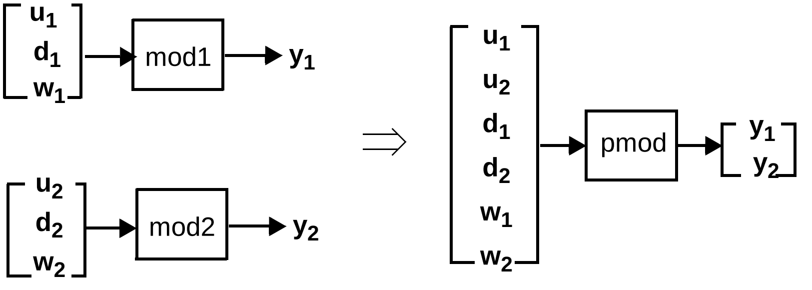
Appends two models to form a composite model that retains the inputs and outputs of the original models. In other words, for models in the MPC **mod** format `appmod` replaces the `append` function of the Control Toolbox.

Syntax

```
pmod = appmod(mod1,mod2)
```

Description

The two input models combine as shown in the following block diagram:



`mod1`, `mod2` and `pmod` are in the MPC **mod** format (see `mod` in the online *MATLAB Function Reference* for a detailed description). You would normally create `mod1` and `mod2` using either the `tfd2mod`, `ss2mod`, or `th2mod` function.

Restriction

`mod1` and `mod2` must have been created with equal sampling periods.

See Also

`addmod`, `addmd`, `addumd`, `paramod`, `sermod`

Purpose

Scales a matrix automatically or by specified mean and standard deviation.

Syntax

```
[ax,mx,stdx] = autosc(x)
sx = scal(x,mx)
sx = scal(x,mx,stdx)
rx = rescal(x,mx)
rx = rescal(x,mx,stdx)
```

Description

`autosc` scales an input matrix or vector `x` by its column means (`mx`) and standard deviations (`stdx`) automatically and outputs `mx` and `stdx` as options. By using `scal`, the input can also be scaled by some specified means and/or standard deviations. rescal converts scaled data back to original data.

Output `mx` is a *row vector* containing the mean value for each column of `x` while `stdx` is a *row vector* containing the standard deviation for each column. Outputs `ax` and `sx` are obtained by dividing the difference of each column of `x` and the mean for the column by the standard deviation for the column, i.e., `ax(:,i) = (x :,i) – mx(i)/stdx(i)`. Output `rx` is determined by multiplying each column of `x` by the corresponding standard deviation and adding the corresponding mean to that product.

If only two arguments are specified in `scal` or `rescal`, `x` is scaled by specified means (`mx`) only.

Example

See `mlr` for an example of the use of these functions.

See Also

`mlr`, `plsr`, `wrtreg`

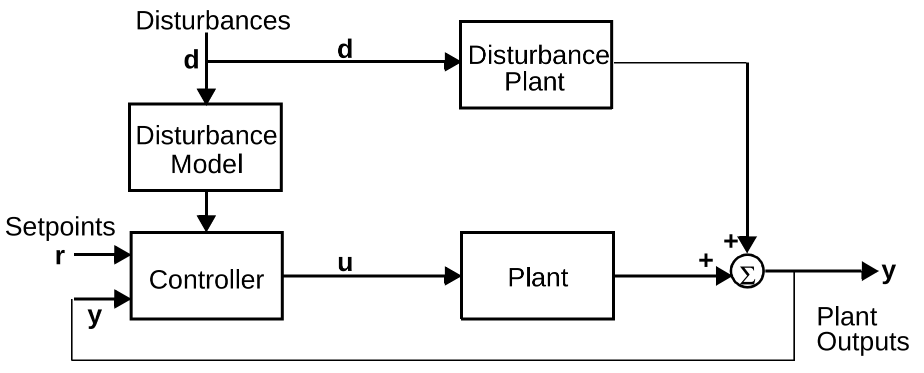
Purpose

Simulates closed-loop systems with hard bounds on manipulated variables and/or outputs using models in the MPC *step* format. Solves the MPC optimization problem by quadratic programming.

Syntax

```
yp = cmpc(plant,model,ywt,uwt,M,P,tend,r)
[yp,u,ym] = cmpc(plant,model,ywt,uwt,M,P,tend,...
     r,ulim,ylim,tfilter,dplant,dmodel,dstep)
```

Description



`cmpc` simulates the performance of the type of system shown in the above diagram when there are bounds on the manipulated variables and/or outputs. Measurement noise can be simulated by treating it as an unmeasured disturbance.

The required input variables are as follows:

`plant`
Is a model in the MPC *step* format that represents the plant.

`model`
Is a model in the MPC *step* format that is to be used for state estimation in the controller. In general, it can be different from `plant` if you want to simulate the effect of plant/controller model mismatch.

ywt

Is a matrix of weights that will be applied to the setpoint tracking errors. If ywt=[ ], the default is equal (unity) weighting of all out- puts over the entire prediction horizon. If ywt¦[ ], it must have n_y columns, where n_y is the number of outputs. All weights must be ≥ 0.

You may vary the weights at each step in the prediction horizon by including up to P rows in ywt. Then the first row of n_y values applies to the tracking errors in the first step in the prediction horizon, the next row applies to the next step, etc. See mpccon for details on the form of the optimization objective function.

If you supply only *nrow* rows, where 1 ≤ *nrow* < P, cmpc will use the last row to fill in any remaining steps. Thus if you want the weighting to be the same for all P steps, you need only specify a single row.

uwt

Same format as ywt, except that uwt applies to the changes in the manipulated variables. If you use uwt = [ ], the default is zero weighting. If uwt ¦ [ ], it must have n_u columns, where n_u is the number of manipulated variables.

M

There are two ways to specify this variable:

If it is a *scalar*, cmpc interprets it as the input horizon (number of moves) as in DMC.

If it is a *row vector* containing n_b elements, each element of the vector indicates the number of steps over which $\Delta u = 0$ during the optimization and cmpc interprets it as a set of n_b blocking factors. There may be $1 \le n_b \le$ P blocking factors, and their sum must be ≤ P

If you set M=[ ] and P ¦ Inf, the default is M=P, which is equivalent to M=ones(1,P). The default value for M is 1 if P=Inf.

P

The number of sampling periods in the prediction horizon. If P=Inf, the prediction horizon is infinite.

tend

Is the desired duration of the simulation (in time units).

r

Is a setpoint matrix consisting of N rows and n_y columns, where n_y is the number of output variables, y:

$$r = \begin{bmatrix} r_1(1) & r_2(1) & \dots & r_{n_y}(1) \\ r_1(2) & r_2(2) & \dots & r_{n_y}(2) \\ \vdots & \vdots & \dots & \vdots \\ r_1(N) & r_2(N) & \dots & r_{n_y}(N) \end{bmatrix}$$

where $r_i(k)$ is the setpoint for output j at time $t = kT$, and T is the sampling period (as specified in the *step* format of `plant` and `model`). If `tend` > NT, the setpoints vary for the first N periods in the simulation, as specified by `r`, and are then held constant at the values given in the last row of `r` for the remainder of the simulation.

In many simulations one wants the setpoints to be constant for the entire time, in which case `r` need only contain a single row of n_y values.

If you set `r=[ ]`, the default is a row of n_y zeros.

The following input variables are optional. In general, setting one of them equal to an empty matrix causes `cmpc` to use the default value, which is given in the description.

ulim

Is a matrix giving the limits on the manipulated variables. Its format is as follows:

$$\text{ulim} = \left[\begin{bmatrix} u_{min,1}(1) & \cdots & u_{min,n_u}(1) \\ u_{min,1}(2) & \cdots & u_{min,n_u}(2) \\ \vdots & \cdots & \vdots \\ u_{min,1}(N) & \cdots & u_{min,n_u}(N) \end{bmatrix} \begin{bmatrix} u_{max,1}(1) & \cdots & u_{max,n_u}(1) \\ u_{max,1}(2) & \cdots & u_{max,n_u}(2) \\ \vdots & \cdots & \vdots \\ u_{max,1}(N) & \cdots & u_{max,n_u}(N) \end{bmatrix} \begin{bmatrix} \Delta u_{max,1}(1) & \cdots & \Delta u_{max,n_u}(1) \\ \Delta u_{max,1}(2) & \cdots & \Delta u_{max,n_u}(2) \\ \vdots & \cdots & \vdots \\ \Delta u_{max,1}(N) & \cdots & \Delta u_{max,n_u}(N) \end{bmatrix}\right]$$

Note that it contains three matrices of N rows. In this case, the limits on N are $1 \le N \le n_b$, where n_b is the number of times the manipulated variables are to change over the input horizon. If you supply fewer than n_b rows, the last row is repeated automatically.

The first matrix specifies the *lower bounds* on the n_u manipulated variables. For example, $u_{min,j}(2)$ is the lower bound for manipulated variable j for the second move of the manipulated variables (where the first move is at the start of the prediction horizon). If $u_{min,j}(k) = -inf$, manipulated variable j will have no lower bound for that move.

The second matrix gives the *upper bounds* on the manipulated variables. If $u_{max,j}(k) = inf$, manipulated variable j will have no upper bound for that move.

The lower and upper bounds may be either positive or negative (or zero) as long as $u_{min,j}(k) \le u_{max,j}(k)$.

The third matrix gives the limits on the rate of change of the manipulated variables. In other words, cmpc will force $|u_j(k) - u_j(k-1)| \le \Delta u_{max,j}(k)$. The limits on the rate of change must be nonnegative and *finite*. If you want it to be unbounded, set the bound to a large number (but not too large — a value of 10^6 should work well in most cases).

The default is $u_{min} = -inf$, $u_{max} = inf$ and $\Delta u_{max} = 10^6$

ylim

Same idea as for ulim, but for the lower and upper bounds of the outputs. The first row applies to the first point in the prediction horizon. The default is $y_{min} = -inf$, and $y_{max} = inf$.

tfilter

Is a matrix of time constants for the noise filter and the unmeasured disturbances entering at the plant output. The first row of n_y elements gives the noise filter time constants and the second row of n_y elements gives the time constants of the lags through which the unmeasured disturbance steps pass. If tfilter only contains one row, the unmeasured disturbances are assumed to be steps. If you set tfilter=[ ] or omit it, the default is no noise filtering and steplike unmeasured disturbances.

dplant

Is a model in MPC *step* format representing all the disturbances (measured and unmeasured) that affect plant in the above diagram. If dplant is provided, then input dstep is also required. For output step disturbances, set dplant=[ ]. The default is no disturbances.

dmodel

Is a model in MPC *step* format representing the measured disturbances. If dmodel is provided, then input dstep is also required. If there are no measured disturbances, set dmodel=[ ]. For output step disturbances, set dmodel=[ ]. If there are both measured and un- measured disturbances, set the columns of dmodel corresponding to the unmeasured disturbances to zero. The default is no measured disturbances.

dstep

Is a matrix of disturbances to the plant. For output step disturbances (`dplant=[ ]` and `dmodel=[ ]`), the format is the same as for `r`. For disturbances through step-response models (`dplant` only or both `dplant` and `dmodel` nonempty), the format is the same as for `r`, except that the number of columns is n_d rather than n_y. The default is a row of zeros.

Notes

- You may use a different number of rows in the matrices `r`, `ulim`, `ylim` and `dstep`, should that be appropriate for your simulation.
- The `ulim` constraints used here are fundamentally different from the `usat` constraints used in the `mpcsim` function. The `ulim` constraints are defined relative to the beginning of the prediction horizon, which moves as the simulation progresses. Thus at each sampling period, k, the `ulim` constraints apply to a block of calculated moves that begin at sampling period k and extend for the duration of the input horizon. The `usat` constraints, on the other hand, are relative to the fixed point $t = 0$, the start of the simulation.

The calculated outputs are as follows (all but `yp` are optional):

yp

Is a matrix containing M rows and n_y columns, where $M = max(fix(tend=T) + 1, 2)$. The first row will contain the initial condition, and row $k - 1$ will give the values of the plant outputs, y (see above diagram), at time $t = kT$.

u

Is a matrix containing the same number of rows as `yp` and n_u columns. The time corresponding to each row is the same as for `yp`. The elements in each row are the values of the manipulated variables, u (see above diagram).

ym

Is a matrix of the same structure as `yp`, containing the values of the predicted output from the state estimator in the controller. These will, in general, differ from those in `yp` if `model¦plant` and/or there are unmeasured disturbances. *The prediction includes the effect of the most recent measurement, i.e.*, it is $\hat{y}(k|k)$.

For unconstrained problems, `cmpc` and `mpcsim` should give the same results. The latter will be faster because it uses an analytical solution of the QP problem, whereas `cmpc` solves it by iteration.

Examples

Consider the linear system:

$$\begin{bmatrix} y_1(s) \\ y_2(s) \end{bmatrix} = \begin{bmatrix} \dfrac{12.8e^{-s}}{16.7s+1} & \dfrac{-18.9e^{-3s}}{21.0s+1} \\ \dfrac{6.6e^{-7s}}{10.9s+1} & \dfrac{-19.4e^{-3s}}{14.4s+1} \end{bmatrix} \begin{bmatrix} u_1(s) \\ u_2(s) \end{bmatrix}$$

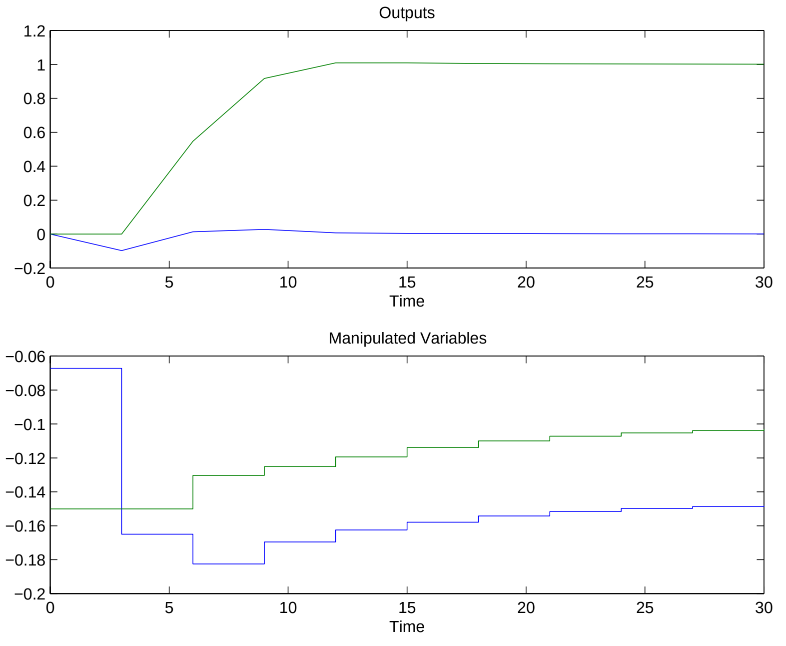
The following statements build the model and set up the controller in the same way as in the mpcsim example.

```
g11=poly2tfd(12.8,[16.7 1],0,1);
g21=poly2tfd(6.6,[10.9 1],0,7);
g12=poly2tfd(-18.9,[21.0 1],0,3);
g22=poly2tfd(-19.4,[14.4 1],0,3);
delt=3; ny=2; tfinal=90;
model=tfd2step(tfinal,delt,ny,g11,g21,g12,g22);
plant=model;
P=6;  M=2;  ywt=[ ]; uwt=[1 1];
tend=30;  r=[0 1];
```

Here, however, we will demonstrate the effect of constraints. First we set a limit of 0.1 on the rate of change of u_1 and a minimum of –0.15 for u_2.

```
ulim=[-inf -0.15 inf inf 0.1 100];
ylim=[ ];
[y,u]=cmpc(plant,model,ywt,uwt,M,P,tend,r,ulim,ylim);
plotall(y,u,delt),pause
```

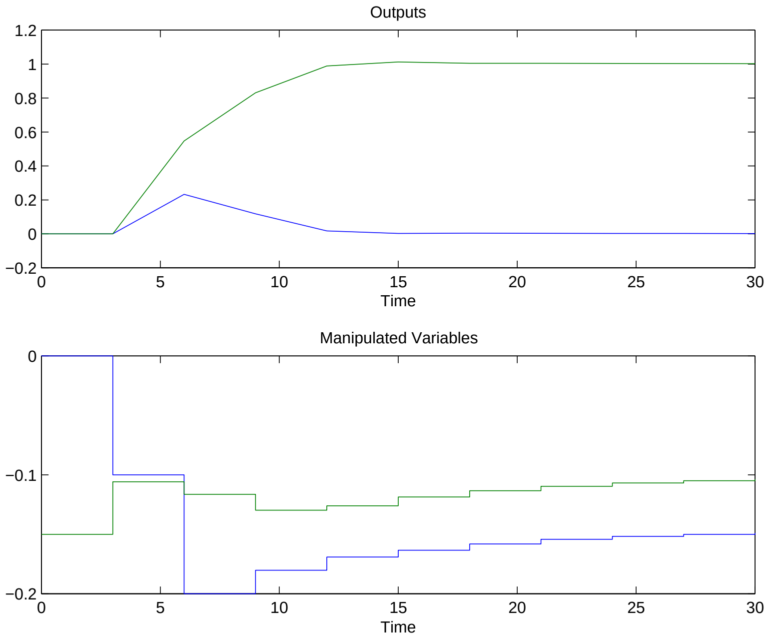
Note that Δu_2 has a large (but finite) limit. It never comes into play.



We next apply a lower bound of zero to both outputs:

```
ulim=[-inf -0.15 inf inf 0.1 100];
ylim=[0 0 inf inf];
[y,u]=cmpc(plant,model,ywt,uwt,M,P,tend,r,ulim,ylim);
plotall(y,u,delt),pause
```

The following results show that no constraints are violated.



Restriction

Initial conditions of zero are used for all the variables. This simulates the condition where all variables represent a deviation from a steady-state initial condition.

Suggestion

Problems with many inequality constraints can be very time consuming. You can minimize the number of constraints by:

- Using small values for `P` and/or `M`.
- Leaving variables unconstrained (limits at ±*inf*) intermittently unless you think the constraint is important.

See Also

`plotall`, `ploteach`, `mpccl`, `mpccon`, `mpcsim`

Purpose

Converts a single-input-single-output, continuous-time transfer function in standard MATLAB polynomial form (including an optional time delay) to a sampled-data transfer function.

Syntax

```
[numd,dend] = cp2dp(num,den,delt)
[numd,dend] = cp2dp(num,den,delt,delay)
```

Description

`num` and `den` are the numerator and denominator polynomials of the continuous-time system (in the standard Control Toolbox polynomial format), `delt` is the sampling period, and `delay` is the (optional) time delay (in time units). If you omit `delay`, `cp2dp` assumes zero delay. The calculated results are numd and dend, the numerator and denominator polynomials of the corresponding discrete-time transfer function. `cp2dp` adds a zero-order hold at the input of the continuous-time system during the conversion to discrete-time.

`cp2dp` accounts properly for the effect of a time delay that is a nonintegral multiple of the sampling period. If `delay=0`, `cp2dp` is equivalent to the MATLAB commands:

```
[a,b,c,d]=tf2ss(num,den);
[phi,gam]=c2dmp(a,b,delt);
[numd,dend]=ss2tf2(phi,gam,c,d,1);
```

Example

See `poly2tfd, poly format` for an example of the use of this function.

Algorithm

`cp2dp` first converts `num` and `den` to the equivalent discrete state-space form. It then accounts for the fractional time delay (if any) using the formulas in Åström and Wittenmark (1984), pages 40–42. Finally, it converts the discrete state-space model to a discrete transfer-function model, simultaneously accounting for the whole periods of delay (if any).

Reference

Åström, K. J.; Wittenmark, B. *Computer Control Systems Theory and Design*, Prentice-Hall, Englewood Cliffs, N.J., 1984.

Restriction

The order of `num` must be ≤ that of `den`.

See Also

`poly2tfd`, `poly format`

Purpose

Solves the discrete Riccati equation by an iterative method to determine the optimal steady-state gain (and optional covariance matrices) for a discrete Kalman filter or state estimator.

Syntax

```
k = dlqe2(phi,gamw,c,q,r)
[k,m,p] = dlqe2(phi,gamw,c,q,r)
```

Description

Filter form:

Consider the state-space description:

$$\begin{aligned} x(k+1) &= \Phi x(k) + \Gamma_u u(k) + \Gamma_d d(k) + \Gamma_w w(k) \\ y(k) &= \bar{y}(k) + z(k) \\ &= Cx(k) + Du(k) + z(k) \end{aligned}$$

where x is a vector of n state variables, u contains n_u known inputs, $\bar{y}$ is a vector of n_y measured outputs, y is the *noise-free output*, w is a vector of n_w unmeasured disturbance inputs, z is a vector of n_y measurement noise inputs, and Φ, Γ_u, Γ_w,C and D are constant matrices. We assume that w and z are stationary random-normal signals (white noise) with covariances

$$E\{w(k)w^T(k)\} = Q$$

$$E\{w(k)z^T(k)\} = R_{12} = 0$$

$$E\{z(k)z^T(k)\} = R$$

The steady-state Kalman filter is

$$\hat{x}(k|k) = \hat{x}(k|k-1) + K[y(k) - C\hat{x}(k|k-1) - Du(k)]$$

$$\hat{x}(k+1|k) = \Phi\hat{x}(k|k) + \Gamma_u u(k)$$

$$\hat{y}(k|k) = C\hat{x}(k|k) + Du(k)$$

where $\hat{x}(k|k)$ is the estimate of $x(k)$ based on the measurements available at period k, $\hat{x}(k|k-1)$is that based on the measurements available at period

$k-1$, etc. Note that $\hat{y}$ is an estimate of the noise-free output, $\bar{y}$. The steady-state Kalman gain, K, is the solution of

$$K = MC^T[R + CMC^T]^{-1}$$

$$M = \Phi P \Phi^T + \Gamma_w Q \Gamma_w^T$$

$$P = M - KCM$$

where M and P may be interpreted as the expected covariance of the errors in the state estimates before and after the measurement update, respectively, i.e.,

$$M = E\{\tilde{x}(k|k-1)\tilde{x}(k|k-1)^T\}$$

$$P = E\{\tilde{x}(k|k)\tilde{x}(k|k)^T\}$$

where, by definition,

$$\tilde{x}(k|k) = x(k) - \hat{x}(k|k)$$

$$\tilde{x}(k|k-1) = x(k) - \hat{x}(k|k-1)$$

The `dlqe2` function takes Φ, Γ_u, C, R, and Q as inputs and calculates K, M, and P. The last two output arguments are optional.

Note that the input and output arguments are identical to those for `dlqe` in the Control Toolbox. The advantage of `dlqe2` is that it can handle a singular state-transition matrix (Φ), e.g., for systems with time delay.

Predictor form:

You can also use `dlqe2` to calculate a state-estimator in the *predictor* form:

$$\hat{x}(k+1|k) = \Phi\hat{x}(k|k-1) + \Gamma_u u(k) + K_p e(k)$$

$$\hat{y}(k|k-1) = C\hat{x}(k|k-1) + Du(k)$$

$$e(k) = y(k) - \hat{y}(k|k-1)$$

The relationship between K_p, the estimator gain for the predictor form, and K as calculated by `dlqe2` is:

$$K_p = \Phi K$$

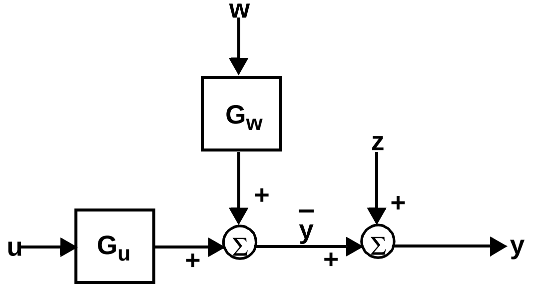
The matrix M calculated by `dlqe2` is the expected covariance of the errors in $\hat{x}(k|k-1)$.

Algorithm

`dlqe2` calls `dareiter`[1], which solves the discrete algebraic Riccati equation using an iterative doubling algorithm.

Example

Consider a system represented by the block diagram:



1. We gratefully acknowledge Kjell Gustafsson, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden, who provided this function.

where G_u and G_w are first-order, discrete-time transfer functions.

$$G_u(z) = \frac{0.20}{1 - 0.8z^{-1}} \qquad G_w(z) = \frac{0.3}{1 - 0.95z^{-1}}$$

and the statistics of the unmeasured inputs are $Q = 2$, $R = 1$.

We use the appropriate MPC Toolbox functions to build a model of the system, then calculate the optimal gain:

```
delt=2; ny=1;
gu=poly2tfd(0.2,[1 -0.8],delt);
Gw=poly2tfd(0.3,[1 -0.95],delt);
[phi,gam,c,d]=mod2ss(tfd2mod(delt,ny,gu,Gw));
k=dlqe2(phi,gam(:,2),c,2,1)
```

The result is:

```
k =      0
    1.0619
```

Note that the gain for the first state is zero since this corresponds to the state of G_u, which is unaffected by the disturbance, w. Also notice that in the composite system, the second column of `gam` is Γ_w. This is because of the order in which `gu` and G_w were specified as inputs to the `tfd2mod` function.

See Also

`smpcest`

imp2step

Purpose

Constructs a multi-input multi-output model in MPC *step* format from multi-input single-output impulse response matrices.

Syntax

```
plant = imp2step(delt,nout,theta1,theta2, ...,
      theta25)
```

Description

Given the impulse response coefficient matrices, `theta1`, `theta2`, etc., a model in MPC *step* format is constructed. Each `thetai` is an n-by-n_u matrix corresponding to the impulse response coefficients for output i. n is the number of the coefficients and n_u is the number of inputs.

`delt` is the sampling interval used for obtaining the impulse response coefficients. `nout` is the output stability indicator. For stable systems, this argument is set equal to number of outputs, n_y. For systems with one or more integrating outputs, this argument is a column vector of length n_y with `nout(i)=0` indicating an integrating output and `nout(i)=1` indicating a stable output.

Example

See `mlr` and `plsr` for examples of the use of this function.

Restriction

The limit on the number of impulse response matrices `thetai` is 25.

See Also

`mlr`, `plsr`

Purpose

Determines impulse response coefficients for a multi-input single-output system via Multivariable Least Squares Regression or Ridge Regression.

Syntax

```
[theta, yres] = mlr(xreg,yreg,ninput)
[theta, yres] = mlr(xreg,yreg,ninput,plotopt,wtheta, ...
      wdeltheta)
```

Description

`xreg` and `yreg` are the input matrix and output vector produced by routines such as `wrtreg`. `ninput` is number of inputs. Least Squares is used to determine the impulse response coefficient matrix, `theta`. Columns of `theta` correspond to impulse response coefficients from each input. Optional output `yres` is the vector of residuals, the difference between the actual outputs and the predicted outputs.

Optional inputs include `plotopt`, `wtheta`, and `wdeltheta`. No plot is produced if `plotopt` is equal to 0 which is the default; a plot of the actual output and the predicted output is produced if `plotopt=1`; two plots — plot of actual and predicted output, and plot of residuals — are produced for `plotopt=2`. Penalties on the squares of `theta` and the changes in `theta` can be specified through the scalar weights `wtheta` and `wdeltheta`, respectively (defaults are 0). `theta` is calculated as follows:

$$theta1 = (X^T X)^{-1} X^T Y$$

where

$$X = \begin{bmatrix} xreg \\ wtheta \times I \\ wdeltheta \times delI \end{bmatrix}$$

$$Y = \begin{bmatrix} yreg \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

where I is identity matrix of dimension $n * n_u$

$$delI = \begin{bmatrix} -1 & 1 & 0 & \dots & 0 \\ 0 & -1 & 1 & \dots & 0 \\ & & \vdots & & \\ 0 & \dots & 0 & -1 & 1 \\ 0 & \dots & 0 & 0 & -1 \end{bmatrix}$$

dimension of $delI$ is $n * n_u$ by $n * n_u$.

then

$$theta = [theta1(1:n) \quad theta1(n+1:2n)\dots$$
$$theta1(nu*(n-1)+1:nu*n)]$$

Example

Consider the following two-input single-output system:

$$y(s) = \begin{bmatrix} \dfrac{5.72e^{-14s}}{60s+1} & \dfrac{1.52e^{-15s}}{25s+1} \end{bmatrix} \begin{bmatrix} u_1(s) \\ u_2(s) \end{bmatrix}$$

Load the input and output data. The input and output data were generated from the above transfer function and random zero-mean noise was added to the output. Sampling time of 7 minutes was used.

```
load mlrdat;
```

Determine the standard deviations for input data using the function autosc.

```
[ax,mx,stdx] = autosc(x);
```

Scale the input data by their standard deviations only.

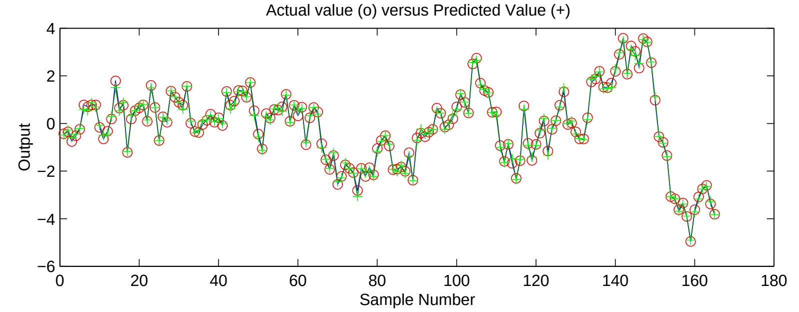
```
mx = [0,0];
sx = scal(x,mx,stdx);
```

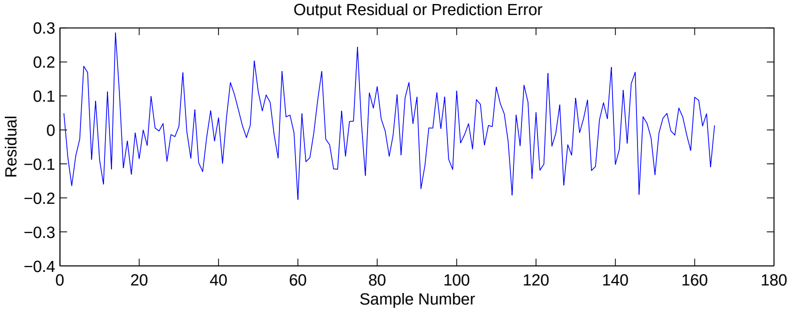
Put the input and output data in a form such that they can be used to determine the impulse response coefficients. 35 impulse response coefficients (n) are used.

```
n = 35;
[xreg,yreg] = wrtreg(sx,y,n);
```

Determine the impulse response coefficients via `mlr`. By specifying `plotopt=2`, two plots — plot of predicted output and actual output, and plot of the output residual (or predicted error) — are produced.

```
ninput = 2;
plotopt = 2;
[theta,yres] = mlr(xreg,yreg,ninput,plotopt);
```





Scale `theta` based on the standard deviations used in scaling the input.

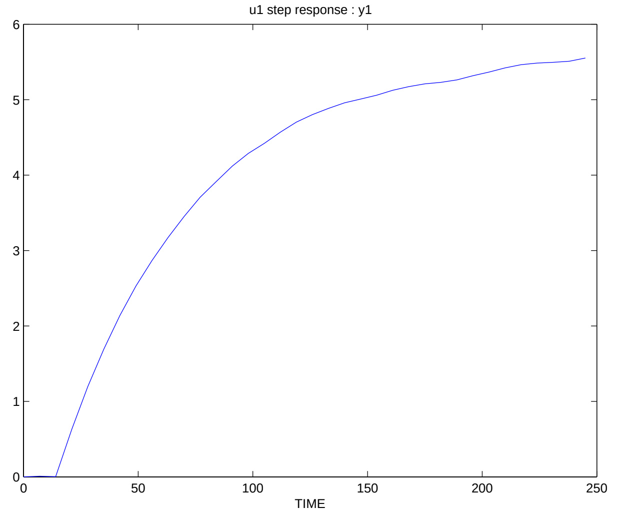
```
theta = scal(theta,mx,stdx);
```

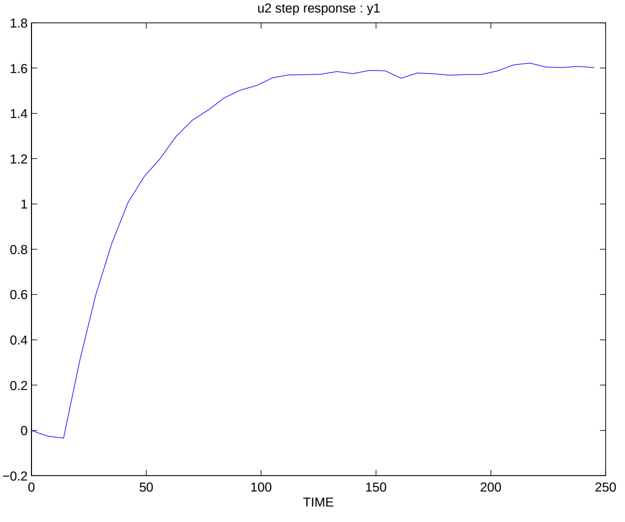
Convert the impulse model to a step model to be used in MPC design. Recall that a sampling time of 7 minutes was used in determining the impulse model. Number of outputs (1 in this case) must be specified.

```
nout = 1;
delt = 7;
model = imp2step(delt,nout,theta);
```

Plot the step response coefficients.

```
plotstep(model)
```





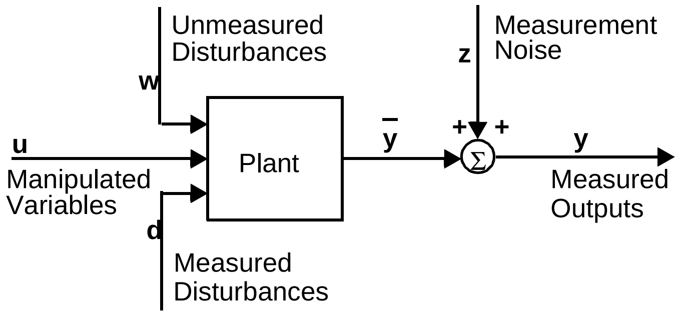
See Also

`plsr`, `validmod`, `wrtreg`

Purpose

The MPC **mod** format is a compact way to store the model of a linear system for subsequent use in the MPC Toolbox functions.

Description



Consider the process shown in the above block diagram. Its *discrete-time* LTI state-space representation is:

$$\begin{aligned} x(k+1) &= \Phi x(k) + \Gamma_u u(k) + \Gamma_d d(k) + \Gamma_w w(k) \\ y(k) &= \bar{y}(k) + z(k) \end{aligned}$$

$$= Cx(k) + D_u u(k) + D_d d(k) + D_w w(k) + z(k)$$

where x is a vector of n state variables, u represents the n_u manipulated variables, d represents n_d measured but freely-varying inputs (i.e., measured disturbances), w represents n_w unmeasured disturbances, y is a vector of n_y plant outputs, z is measurement noise, and Φ, Γ_u, etc., are constant matrices of appropriate size. The variable $\bar{y}(k)$ represents the plant output before the addition of measurement noise. Define:

$\Gamma = [\Gamma_u \; \Gamma_d \; \Gamma_w]$

$D = [D_u \; D_{ud} \; D_w]$

In some cases one would like to include n_{ym} measured and n_{yu} unmeasured outputs in y, where $n_{ym} + n_{yu} = n_y$. If so, the **mod** format assumes that the y vector and the C and D matrices are arranged such that the measured outputs come first, followed by the unmeasured outputs.

mod format

The **mod** format is a single matrix that contains the Φ, Γ, C, and D matrices, plus some additional information. Let M be the **mod** representation of the above system. Its overall dimensions are:

- Number of rows = $n + n_y + 1$
- Number of columns = $max(7, 1 + n + n_u + n_d + n_w)$

The `minfo` vector is the first seven elements of the first row in M. The elements of `minfo` are:

`minfo`	(1)	T, the sampling period used to create the model.
	(2)	n, the number of states.
	(3)	n_u, the number of manipulated variable inputs.
	(4)	n_d, the number of measured disturbances.
	(5)	n_w, the number of unmeasured disturbances.
	(6)	n_{ym}, the number of measured outputs.
	(7)	n_{yu}, the number of unmeasured outputs.

The remainder of M contains the discrete state-space matrices:

Φ in rows 2 to $n + 1$	columns 2 to $n + 1$
Γ in rows 2 to $n + 1$	columns $n + 2$ to $n + n_u + n_d + n_w + 1$
C in rows $n + 2$ to $n + n_y + 1$	columns 2 to $n + 1$
D in rows $n + 2$ to $n + n_y + 1$	columns $n + 2$ to $n + n_u + n_d + n_w + 1$

Notes

Since the `minfo` vector requires seven columns, this is the minimum possible number of columns in the **mod** format, regardless of the dimensions of the state-space matrices.

Also, the first column is reserved for other uses by MPC Toolbox routines. Thus the state-space matrices start in column 2, as described above.

In order for the `mpcinfo` routine to recognize matrices in the MPC **mod** format, the (2,1) element is set to `NaN` (Not-a-Number).

Example See ss2mod for a **mod** format example.

See Also mod2ss, mod2step, step format, mpcinfo, ss2mod, step, tfd2mod, tf format, th2mod, theta format

Purpose

Calculates the complex frequency response in *varying* format of a system in MPC **mod** format.

Syntax

```
frsp = mod2frsp(mod,freq)
[frsp,eyefrsp] = mod2frsp(mod,freq,out,in,balflg)
```

Description

`mod2frsp` calculates the complex frequency response of a system (`mod`) in MPC **mod** format. The desired frequencies are given by the input `freq`, a *row vector* of 3 elements specifying the lower frequency as a power of 10, the upper frequency as a power of 10, and the number of frequency points.

Optional inputs `out` and `in` are *row vector*s that specify the outputs and inputs for which the frequency response is to be generated. If these variables are omitted or empty, the default is to use all outputs and inputs.

Optional input `balflg` indicates whether the system's Φ matrix should be balanced (using the MATLAB `balance` command). If `balflg` is nonzero, balancing is performed. Balancing improves the conditioning of the problem, but may cause errors in the frequency response. If `balflg=[ ]` or is omitted, no balancing is performed.

Output `frsp` is the frequency response matrix given in *varying* format. Let $F(\omega)$ denote a matrix-valued function of the independent variable ω. Then the N sampled values $F(\omega_1), \ldots, F(\omega_N)$ are contained in `frsp` as follows:

$$\text{frsp} = \begin{bmatrix} F(\omega_1) & \omega_1 \\ \vdots & \vdots \\ F(\omega_i) & \omega_N \\ \vdots & 0 \\ F(\omega_N) & \vdots \\ & 0 \\ 0 \ldots 0\, N & inf \end{bmatrix}$$

If the dimension of each submatrix $F(\omega_i)$ is n by m, then the dimensions of `frsp` is $n \cdot N + 1$ by $m + 1$.

Optional output `eyefrsp` is in *varying* format and represents $I - F(\omega_i)$ at each frequency. This output can only be specified for square submatrices and may be useful in computing the frequency responses of both the sensitivity and complementary sensitivity functions.

Example

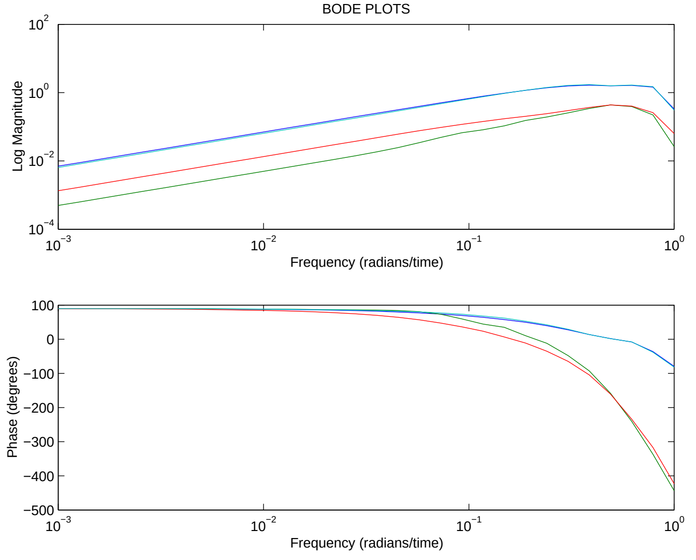
Consider the linear system:

$$\begin{bmatrix} y_1(s) \\ y_2(s) \end{bmatrix} = \begin{bmatrix} \dfrac{12.8e^{-s}}{16.7s+1} & \dfrac{-18.9e^{-3s}}{21.0s+1} \\ \dfrac{6.6e^{-7s}}{10.9s+1} & \dfrac{-19.4e^{-3s}}{14.4s+1} \end{bmatrix} \begin{bmatrix} u_1(s) \\ u_2(s) \end{bmatrix}$$

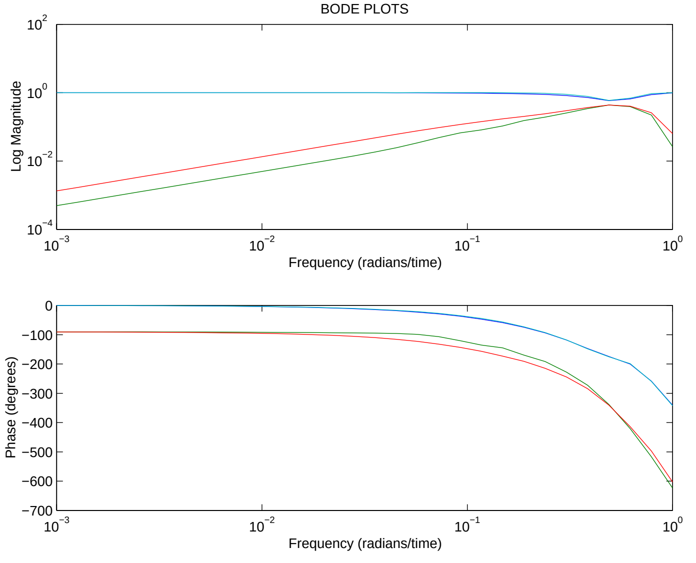
See the `mpccl` example for the commands that build the a closed-loop model for this process using a simple controller. However for this example, `delt=6` and `tfinal=90` are used to reduce the number of step response coefficients.

Now we will calculate and plot the frequency response of the sensitivity and complementary sensitivity functions.

```
freq = [-3,0,30];
in = [1:ny]; % input is r for comp. sensitivity
out = [1:ny]; % output is yp for comp. sensitivity
[frsp,eyefrsp] = mod2frsp(clmod,freq,out,in);
plotfrsp(eyefrsp); % Sensitivity
pause;
```

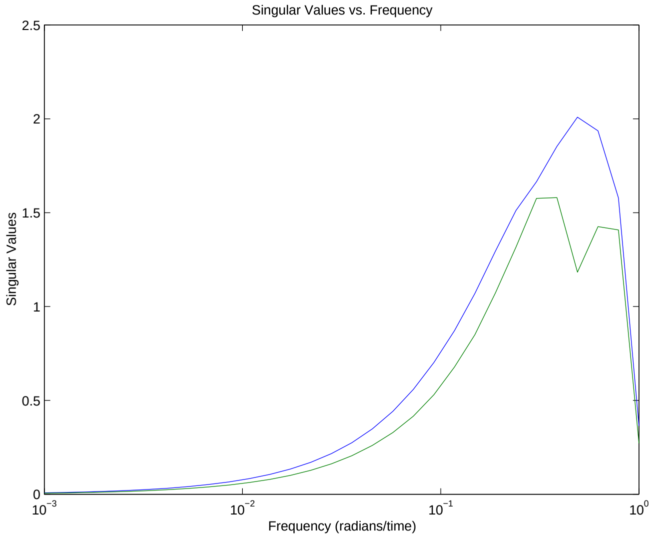



```
plotfrsp(frsp); % Complementary Sensitivity pause;
```


BODE PLOTS
Log Magnitude
10^{2}
10^{0}
10^{-2}
10^{-4}
10^{-3}
10^{-2}
10^{-1}
10^{0}
Frequency (radians/time)
Phase (degrees)
0
−100
−200
−300
−400
−500
−600
−700
10^{-3}
10^{-2}
10^{-1}
10^{0}
Frequency (radians/time)

Calculate and plot the singular values for the sensitivity function response.

```
[sigma,omega] = svdfrsp(eyefrsp);
clg;
semilogx(omega,sigma);
title('Singular Values vs. Frequency');
xlabel('Frequency (radians/time)');
ylabel('Singular Values');
```



Algorithm

The algorithm to calculate the complex frequency response involves a matrix inverse problem which is solved via a Hessenberg matrix.

Reference

A.J. Laub, "Efficient Multivariable Frequency Response Computations," *IEEE Transactions on Automatic Control*, Vol. AC–26, No. 2, pp.407–408, April, 1981.

See Also

`mod`, `mpccl`, `plotfrsp`, `smpccl`, `svdfrsp`

Purpose Changes the sampling period of a model in MPC **mod** format.

Syntax

```
newmod = mod2mod(oldmod,delt2)
```

Description Input `oldmod` is the existing model in MPC **mod** format. Input `delt2` is the new sampling period for the model. `mod2mod` returns `newmod`, which is the system in **mod** format converted to the new sampling time.

See Also `mod`, `ss2mod`

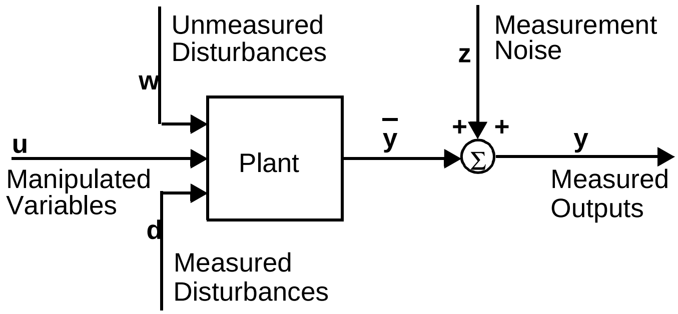
Purpose

Extracts the standard discrete-time state-space matrices and other information from a model stored in the MPC **mod** format.

Syntax

```
[phi,gam,c,d] = mod2ss(mod)
[phi,gam,c,d,minfo] = mod2ss(mod)
```

Description



Consider the process shown in the above block diagram. `mod2ss` assumes that `mod` is a description of the above process in the MPC **mod** format (see `mod` in the online *MATLAB Function Reference* for more details). An equivalent state-space representation is:

$$
\begin{aligned}
x(k+1) &= \Phi x(k) + \Gamma_u u(k) + \Gamma_d d(k) + \Gamma_w w(k) \\
y(k) &= \bar{y}(k) + z(k) \\
&= Cx(k) + D_u u(k) + D_d d(k) + D_w w(k) + z(k)
\end{aligned}
$$

where x is a vector of n state variables, u represents the n_u manipulated variables, d represents n_d measured but freely-varying inputs (i.e., measured disturbances), w represents n_w unmeasured disturbances, y is a vector of n_y plant outputs, z is measurement noise, and Φ, Γ_u, etc., are constant matrices of appropriate size. The variable $\bar{y}$ (k) represents the plant output before the addition of measurement noise. Define:

$$\Gamma = [\Gamma_u \; \Gamma_d \; \Gamma_w]$$

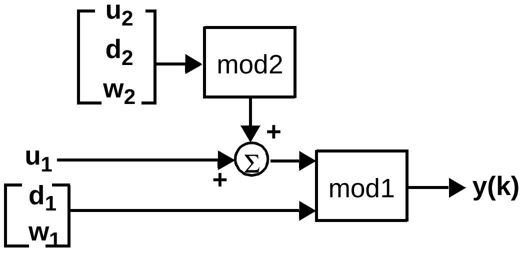
$$D = [D_u \; D_d \; D_w]$$

mod2ss extracts the Φ, Γ, C, and D matrices from the input variable, mod. It also extracts the vector minfo, which contains additional information about the sampling period, number of each type of input and output, etc. see mod in the online *MATLAB Function Reference* for more details on minfo.

Examples

1 See the example in the description of dlqe2.

2 Suppose you have a plant with the structure



where the inputs and outputs are all scalars, and you have constructed mod1 and mod2 using the commands:

```
phi1=diag([-0.7, 0.8]); gam1=[1, -1, 0; 0, 0, 1];
c1=[0.2 -0.4]; d1=zeros(1,3);
minfo1=[1 2 1 1 1 1 0];
mod1=ss2mod(phi1,gam1,c1,d1,minfo1);
phi2=-0.2; gam2=[1, -0.5, 0.2];
c2=3; d2=[0.2, 0, 0];
minfo2=[1 1 1 1 1 1 0];
mod2=ss2mod(phi2,gam2,c2,d2,minfo2);
pmod=addmod(mod1,mod2);
```

Now you want to calculate the response to a step change in d_2, which is the *fourth* input to the composite system, pmod. One way to do it is:

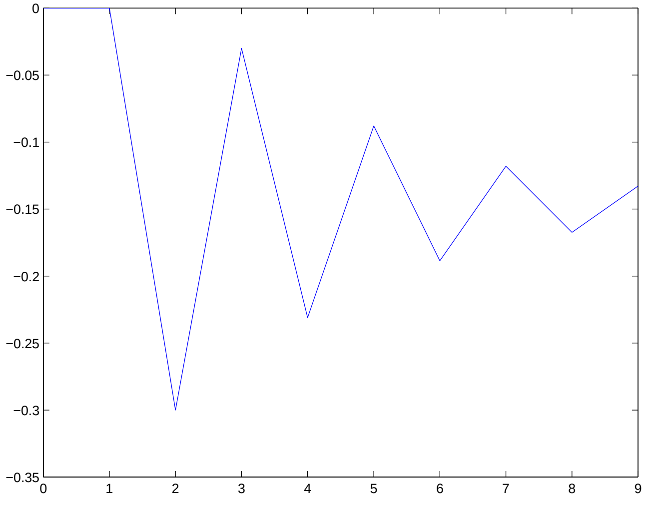
```
[phi,gam,c,d,minfo]=mod2ss(pmod);
nstep=10;
ustep=[zeros(nstep,3) ones(nstep,1) zeros(nstep,2)];
% Define step in d2
y=dlsimm(phi,gam,c,d,ustep);
% simulate response to step input
plot([0:nstep-1],y)
```

The results of the mod2ss call are:

```
phi=
-0.7000          0     3.0000
      0     0.8000          0
      0          0    -2.0000

gam=
1.0000     2.2000    -1.0000        0          0          0
                 0
     0          0          0        0     1.0000          0
     0     1.0000          0   -.5000          0     0.2000
c=
0.2000    -0.4000          0
d=
     0          0          0        0          0          0
minfo=
     1          3          2        2          2          1        0
```

And the step response is:



See Also mod, ss2mod

Purpose

Uses a model in the **mod** format to calculate the step response of a SISO or MIMO system in MPC *step* format.

Syntax

```
plant = mod2step(mod,tfinal)
[plant,dplant] = mod2step(mod,tfinal,delt2,nout)
```

Description

The input variable `mod` is assumed to be a model in the **mod** format (see `mod` in the online *MATLAB Function Reference* for a description). You would normally create it using `ss2mod`, `tfd2mod`, or `th2mod`. The input variable `tfinal` is the time at which you would like to end the step response.

The optional input variable `delt2` is the desired sampling period for the step response. If you use `delt2=[ ]` or omit it, the default is equal to the sampling period of `mod` (contained in the `minfo` vector of `mod`).

The optional input variable `nout` is the output stability indicator. For stable systems, set `nout` equal to the number of outputs, n_y. For systems with one or more integrating outputs, `nout` is a column vector of length n_y with `nout(i)=0` indicating an integrating output and `nout(i)=1` indicating a stable output. If you use `nout=[ ]` or omit it, the default is `nout=`n_y (only stable outputs).

`plant` and `dplant` are matrices in MPC *step* format containing the calculated step responses. `plant` is the response to the manipulated variables, and `dplant` is the response to the disturbances (if any), both measured and unmeasured. The overall dimensions of these matrices are:

plant n-by-n_y + n_y + 2 rows, n_u columns.

`dplant` n-by-n_y + n_y + 2 rows, $n_d + n_w$ columns.

where n = `round (tfinal/delt2)`

It is assumed that stable step responses are nearly constant after n sampling periods, while integrating responses increase with a constant slope after $n - 1$ sampling periods.

Each column gives the step response with respect to the corresponding input variable. Within each column, the first n_y elements are the response for each output at time $t = T$, the next n_y elements give each output at time $t = 2T$, etc.

The last $n_y + 2$ rows contain `nout`, n_y and `delt2`, respectively (all in column 1 — any remaining elements in these rows are set to zero). In other words, for `plant` the arrangement is as follows:

$$
\text{plant} = \begin{bmatrix} & S_1 & & \\ & S_2 & & \\ & \vdots & & \\ & S_n & & \\ \text{nout}(1) & 0 & \dots & 0 \\ \text{nout}(2) & 0 & \dots & 0 \\ \vdots & \vdots & & \vdots \\ \text{nout}(\text{n}_\text{y}) & 0 & \dots & 0 \\ n_y & 0 & \dots & 0 \\ \text{delt2} & 0 & \dots & 0 \end{bmatrix}_{(n \cdot n_y + n_y + 2) \times n_u}
$$

where

$$
S_i = \begin{bmatrix} S_{1,1,i} & S_{1,2,i} & \dots & S_{1,n_u,i} \\ S_{2,1,i} & S_{2,2,i} & \dots & S_{2,n_u,i} \\ \vdots & & & \\ S_{n_y,1,i} & S_{n_y,2,i} & \dots & S_{n_y,n_u,i} \end{bmatrix}
$$

S_{kji} is the i^{th} step response coefficient describing the effect of input j on output k.

The arrangement of `dplant` is similar; the only difference is in the number of columns.

Example

The following process has 3 inputs and 4 outputs:

```
phi=diag([0.3, 0.7, -0.7]);
gam=eye(3);
c=[1 0 0;  0 0 1;  0 1 1;  0 1 0];
d=[1 0 0; zeros(3,3)];
```

We first calculate its step response for 4 samples (including the initial condition) with respect to each of the inputs using the Control Toolbox function, `dstep`:

```
nstep=4; delt=1.5;
yu1=dstep(phi,gam,c,d,1,nstep)
yu2=dstep(phi,gam,c,d,2,nstep)
yu3=dstep(phi,gam,c,d,3,nstep)
```

The results are:

	Response to u_1				Response to u_2				Response to u_3			
Time	y_1	y_2	y_3	y_4	y_1	y_2	y_3	y_4	y_1	y_2	y_3	y_4
0	1	0	0	0	0	0	0	0	0	0	0	0
T	2	0	0	0	0	0	1	1	0	1	1	0
2T	2.3	0	0	0	0	0	1.7	1.7	0	0.3	0.3	0
3T	2.39	0	0	0	0	0	2.19	2.19	0	0.79	0.79	0

We then use `mod2step` to do the same job:

```
plant=mod2step(ss2mod(phi,gam,c,d,delt),(nstep-1)*delt)
```

obtaining the results:

```
plant =
    2.0000         0         0
         0         0    1.0000
         0    1.0000    1.0000
         0    1.0000         0
    2.3000         0         0
         0         0    0.3000
         0    1.7000    0.3000
         0    1.7000         0
    2.3900         0         0
         0         0    0.7900
         0    2.1900    0.7900
         0    2.1900         0
    1.0000         0         0
    1.0000         0         0
    1.0000         0         0
    1.0000         0         0
    4.0000         0         0
    1.5000         0         0
```

See Also `plotstep`, `ss2step`, `tfd2step`

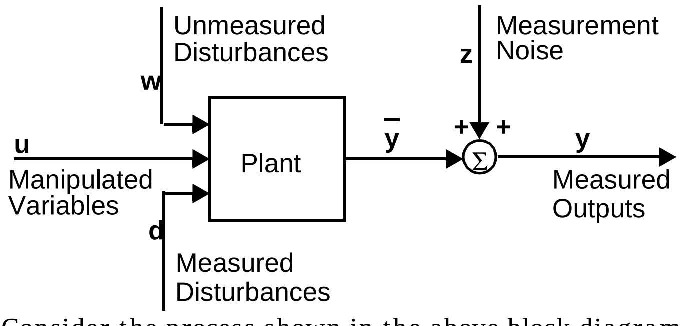
Purpose

Differences the states of a system and augments them with the output variables. Mainly used as a utility function for setting up model predictive controllers.

Syntax

```
[phia,gama,ca] = mpcaugss(phi,gam,c)
[phia,gama,ca,da,na] = mpcaugss(phi,gam,c,d)
```

Description



Consider the process shown in the above block diagram. A state-space representation is:

$$\begin{aligned} z(k+1) &= \Phi x(k) + \Gamma_u u(k) + \Gamma_d d(k) + \Gamma_w w(k) \\ y(k) &= \bar{y}(k) + D_u u(k) + D_d d(k) + D_w w(k) + z(k) \\ \\ &= \bar{y}(k) + z(k) \end{aligned}$$

where x is a vector of n state variables, u is a vector of n_u manipulated variables, d is a vector of n_d measured disturbances, w is a vector of n_w unmeasured disturbances, y is a vector of n_y plant outputs, z is measurement noise, and Φ, Γ_u, Γ_d, Γ_w, etc., are constant matrices of appropriate size. The variable $\tilde{y}(k) = Cx(k)$ represents the plant output before the addition of the direct contribution of the inputs $[D_u u(k) + D_d v(k) + D_w w(k)]$ and the measurement noise [z(k)]. (The variable $\bar{y}$ is the output before addition of the measurement noise). Define:

$$\Delta u(k) = u(k) - u(k-1)$$

$$\Delta x(k) = x(k) - x(k-1)$$

etc. Then equations 4.28 and 4.29 can be converted to the form

$x_a(k+1) = \Phi_a x_a(k) + \Gamma_{ua}\,\Delta u(k) + \Gamma_{da}\,\Delta(k) + \Gamma_{wa}\,\Delta w(k)$

$y(k) = C_a x_a(k) + D_u u(k) + D_d d(k) + D_w w(k) + z(k)$

where, by definition,

$$x_a(k) = \begin{bmatrix} \Delta x(k) \\ \tilde{y}(k) \end{bmatrix}$$

$$\Phi_a = \begin{bmatrix} \Phi & 0 \\ C\Phi & I \end{bmatrix} \quad \Gamma_a = \begin{bmatrix} \Gamma_{ua} & \Gamma_{da} & \Gamma_{wa} \end{bmatrix}$$

$$\Gamma_{ua} = \begin{bmatrix} \Gamma_u \\ C\Gamma_u \end{bmatrix} \quad \Gamma_{da} = \begin{bmatrix} \Gamma_d \\ C\Gamma_d \end{bmatrix} \quad \Gamma_{wa} = \begin{bmatrix} \Gamma_w \\ C\Gamma_w \end{bmatrix}$$

$$C_a = \begin{bmatrix} 0 & I \end{bmatrix} \quad D_a = \begin{bmatrix} D_u & D_d & D_w \end{bmatrix}$$

The `mpcaugss` function takes the matrices Φ, $\Gamma(= [\Gamma_u\ \Gamma_d\ \Gamma_w])$, C as input, and creates the augmented matrices Φ_a, Γ_a, C_a and D_a in the form shown above. The D input matrix is optional. If you include it, `mpcaugss` assumes it has the form $D = [D_u\ D_d\ D_w]$. If you omit it, the default is zero. Note that all MPC design routines require $D_u = D_d = 0$.

The last output variable, *na*, is the order of the augmented system, i.e., $n_a = n + n_y$. It is optional.

Example

The following system has 2 states, 3 inputs, and 2 outputs.

```
phi=diag([0.8, -0.2]);
gam=[1 -1 0;0 2 -0.5];
c=[0.4 0;0 1.5];
```

Here is the augmentation command, followed by the calculated results:

```
[phia,gama,ca,da,na]=mpcaugss(phi,gam,c)

phia =
    0.8000         0         0         0
         0   -0.2000         0         0
    0.3200         0    1.0000         0
         0   -0.3000         0    1.0000
gama =
    1.0000   -1.0000         0
         0    2.0000   -0.5000
    0.4000   -0.4000         0
         0    3.0000   -0.7500
ca =
     0     0     1     0
     0     0     0     1
da =
     0     0     0
     0     0     0
na =
     4
```

mpccl

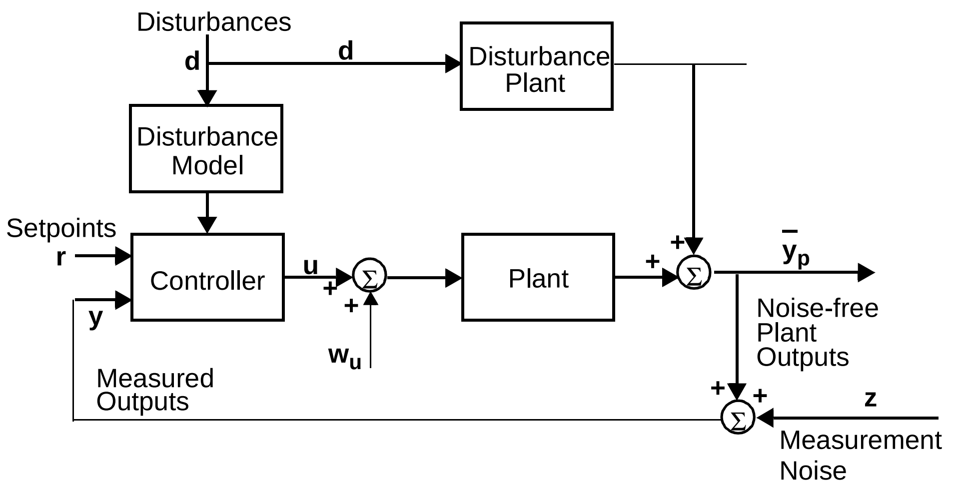
Purpose

Combines a plant model and a controller model in MPC *step* format, yielding a closed-loop system model in the MPC **mod** format. This can be used for stability analysis and linear simulations of closed-loop performance.

Syntax

```
[clmod] = mpccl(plant,model,Kmpc)
[clmod,cmod] = mpccl(plant,model,Kmpc,tfilter,...
  dplant, dmodel)
```

Description



`plant`

Is a model (in *step* format) representing the plant in the above diagram.

`model`

Is a model (in *step* format) that is to be used to design the MPC controller block shown in the diagram. It may be the same as plant (in which case there is no "model error" in the controller design), or it may be different.

`Kmpc`

Is a controller gain matrix, which was calculated by the function `mpccon`.

`tfilter`

Is a (optional) matrix of time constants for the noise filter and the unmeasured disturbances entering at the plant output. If omitted or set to an empty matrix, the default is no noise filtering and steplike unmeasured disturbances. See the documentation for the function `mpcsim` for more details on the design and proper format of `tfilter`.

`dplant`

Is a (optional) model (in *step* format) representing all the disturbances (measured and unmeasured) that affect `plant` in the above diagram. If omitted or set to an empty matrix, the default is that there are no disturbances.

`dmodel`

Is a (optional) model (in *step* format) representing the measured disturbances. If omitted or set to an empty matrix, the default is that there are no measured disturbances. See the documentation for the function `mpcsim` for more details on how disturbances are handled when using step-response models.

`mpccl`

Calculates a model of the closed-loop system, `clmod`. It is in the **mod** format and can be used, for example, with analysis functions such as `smpcgain` and `smpcpole`, and with simulation routines such as `mod2step` and `dlsimm`. `mpccl` also calculates (as an option) a model of the controller element, `cmod`.

The closed-loop model, `clmod`, has the following state-space representation:

$$x_{cl}(k+1) = \Phi_{cl}x_{cl}(k) + \Gamma_{cl}u_{cl}(k)$$

$$y_{cl}(k) = C_{cl}x_{cl}(k) + D_{cl}u_{cl}(k)$$

where x_{cl} is a vector of n state variables, u_{cl} is a vector of input variables, y_{cl} is a vector of outputs, and Φ_{cl}, Γ_{cl}, C_{cl}, and D_{cl} are matrices of appropriate size. The expert user may want to know the significance of the state variables in x_{cl}. They are (in the following order):

- The n_p states of the plant (as specified in `plant`),
- The n_i state estimates (based on the model specified in model),
- n_d integrators that operate on the Δd signal to yield a d signal. If there are no disturbances, these states are omitted.

- n_u integrators that operate on the Δw_u signal to yield a w_u signal.
- n_u integrators that operate on the Δu signal produced by the standard MPC formulation to yield a u signal that can be used as input to the plant and as a closed-loop output.

The closed-loop input and output variables are:

$$u_{cl}(k) = \begin{bmatrix} r(k) \\ z(k) \\ w_u(k) \\ d(k) \end{bmatrix} \quad \text{and} \quad y_{cl}(k) = \begin{bmatrix} \bar{y}_p(k) \\ u(k) \\ \hat{y}(k|k) \end{bmatrix}$$

where $\hat{y}(k|k)$ is the estimate of the noise-free plant output at sampling period k based on information available at period k. This estimate is generated by the controller element.

The state-space form of the controller model, `cmod`, can be written as:

$$x_c(k+1) = \Phi_c x_c(k) + \Gamma_c u_c(k)$$

$$y_c(k) = C_c x_c(k) + D_c u_c(k)$$

where

$$u_c(k) = \begin{bmatrix} r(k) \\ y(k) \\ d(k-1) \end{bmatrix} \quad \text{and} \quad y_c(k) = u(k-1)$$

and the controller states are the same as those of the closed loop system *except* that the n_p plant states are not included.

Example

Consider the linear system:

$$\begin{bmatrix} y_1(s) \\ y_2(s) \end{bmatrix} = \begin{bmatrix} \dfrac{12.8e^{-s}}{16.7s+1} & \dfrac{-18.9e^{-3s}}{21.0s+1} \\ \dfrac{6.6e^{-7s}}{10.9s+1} & \dfrac{-19.4e^{-3s}}{14.4s+1} \end{bmatrix} \begin{bmatrix} u_1(s) \\ u_2(s) \end{bmatrix}$$

We build the step response model using the MPC Toolbox functions `poly2tfd` and `tfd2step`.

```
g11=poly2tfd(12.8,[16.7 1],0,1);
g21=poly2tfd(6.6,[10.9 1],0,7);
g12=poly2tfd(-18.9,[21.0 1],0,3);
g22=poly2tfd(-19.4,[14.4 1],0,3);
delt=3; ny=2; tfinal = 60;
model=tfd2step(tfinal,delt,ny,g11,g21,g12,g22);
plant=model; % No plant/model mismatch
```

Now we design the controller. Since there is delay, we use `M < P`: We specify the defaults for the other tuning parameters, `uwt` and `ywt`, then calculate the controller gain:

```
P=6; % Prediction horizon.
M=2; % Number of moves (input horizon).
ywt=[ ]; % Output weights (default - unity
% on all outputs). uwt=[ ]; % Man. Var weights (default - zero
% on all man. vars).
Kmpc=mpccon(model,ywt,uwt,M,P);
```

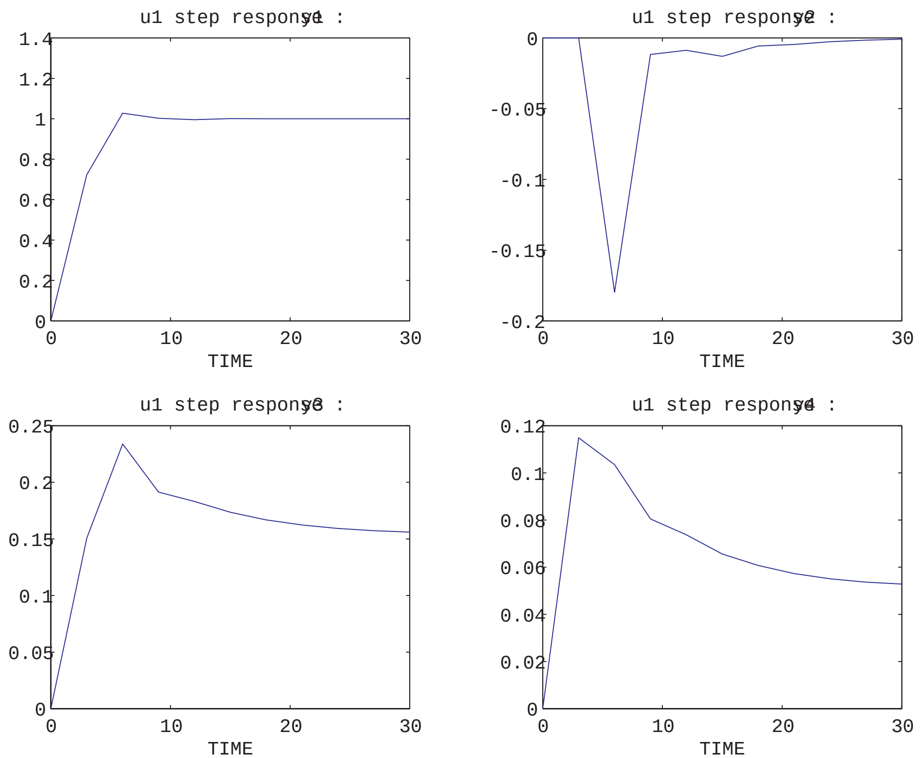
Now we can calculate the model of the closed-loop system:

```
clmod=mpccl(plant,model,Kmpc);
```

You can use the closed-loop model to calculate and plot the step response with respect to all the inputs. The appropriate commands are:

```
tend=30;
clstep=mod2step(clmod, tend);
plotstep(clstep)
```

Since the closed-loop system has $m = 6$ inputs and $p = 6$ outputs, only one of the plots is reproduced here. It shows the response of the first 4 closed-loop outputs to a unit step in the first closed-loop input, which is the setpoint for y_1:



Closed-loop outputs y_1 and y_2 are the true plant outputs (noise-free). Output y_1 goes to the new setpoint quickly with a small overshoot. This causes a small, short-term disturbance in y_2. The plots for y_3 and y_4 show the required variation in the manipulated variables.

Restriction

`model` and `plant` must have been created using the same sampling period.

See Also

`cmpc`, `mod2step`, `step format`, `mpccon`, `mpcsim`, `smpcgain`, `smpcpole`

Purpose

Calculates MPC controller gain using a model in MPC *step* format.

Syntax

```
Kmpc = mpccon(model)
Kmpc = mpccon(model,ywt,uwt,M,P)
```

Description

Combines the following variables (most of which are optional and have default values) to calculate the MPC gain matrix, `Kmpc`.

`model`
is the model of the process to be used in the controller design (in the *step* format).

The following input variables are optional:

`ywt`
Is a matrix of weights that will be applied to the setpoint tracking errors. If you use `ywt=[ ]` or omit it, the default is equal (unity) weighting of all outputs over the entire prediction horizon. If you use `ywt`¦`[ ]`, it must have n_y columns, where n_y is the number of outputs. All weights must be ≥ 0.

You may vary the weights at each step in the prediction horizon by including up to `P` rows in `ywt`. Then the first row of n_y values applies to the tracking errors in the first step in the prediction horizon, the next row applies to the next step, etc.

If you supply only *nrow* rows, where 1 ≤ *nrow* < `P`, `mpccon` will use the last row to fill in any remaining steps. Thus if you wish the weighting to be the same for all `P` steps, you need only specify a single row.

`uwt`
Same format as `ywt`, except that `uwt` applies to the changes in the manipulated variables. If you use `uwt = [ ]` or omit it, the default is zero weighting. If uwt ¦ [ ], it must have n_u columns, where n_u is the number of manipulated variables.

`M`
There are two ways to specify this variable:

- If it is a *scalar*, `mpccon` interprets it as the input horizon (number of moves) as in DMC.
- If it is a *row vector* containing n_b elements, each element of the vector indicates the number of steps over which $\Delta u = 0$ during the optimization and `cmpc` interprets it as a set of n_b blocking factors. There may be $1 \le n_b \le$ `P` blocking factors, and their sum must be $\le$ `P`.

  If you set `M=[ ]` or omit it and `P ¦ Inf`, the default is `M=P`, which is equivalent to `M=ones(1,P)`. The default value for `M` is `1` if `P=Inf`.

P

The number of sampling periods in the prediction horizon. If you set `P=Inf` or omit it, the default is `P=1`. If `P=inf`, the prediction horizon is infinite.

If you take the default values for all the optional variables, *y*ou get the "perfect controller," i.e., a model-inverse controller. This controller is not applicable when one or more outputs can not respond to the manipulated variables within one sampling period due to time delay. In this case, the plant-inverse controller is unrealizable. For nonminimum phase discrete plants, this controller is unstable. To counteract this you can penalize changes in the manipulated variables (variable `uwt`), use blocking (variable `M`), and/or make `P>>M`. The model-inverse controller is also relatively sensitive to model error and is best used as a point of reference from which you can progress to a more robust design.

Algorithm

The controller gain is a component of the solution to the optimization problem:

$$\text{Minimize} \quad J(k) = \sum_{j=1}^{p} \sum_{i=1}^{n_y} (ywt_i(j)[r_i(k+j) - \hat{y}_i(k+j)])^2 + \sum_{j=1}^{n_b} \sum_{i=1}^{n_u} (uwt_i(j)\Delta\hat{u}_i(j))^2$$

with respect to $\Delta\hat{u}_i(j)$ (a series of current and future moves in the manipulated variables), where $\hat{y}_i(k + j)$ is a prediction of output *i* at a time *j* sampling periods into the future (relative to the current time, *k*), which is a function of

$\Delta \hat{u}_i(j)$, $r_i(k+j)$ is the corresponding future setpoint, and n_b is the number of blocks or moves of the manipulated variables.

Example

Consider the linear system:

$$\begin{bmatrix} y_1(s) \\ y_2(s) \end{bmatrix} = \begin{bmatrix} \dfrac{12.8e^{-s}}{16.7s+1} & \dfrac{-18.9e^{-3s}}{21.0s+1} \\ \dfrac{6.6e^{-7s}}{10.9s+1} & \dfrac{-19.4e^{-3s}}{14.4s+1} \end{bmatrix} \begin{bmatrix} u_1(s) \\ u_2(s) \end{bmatrix}$$

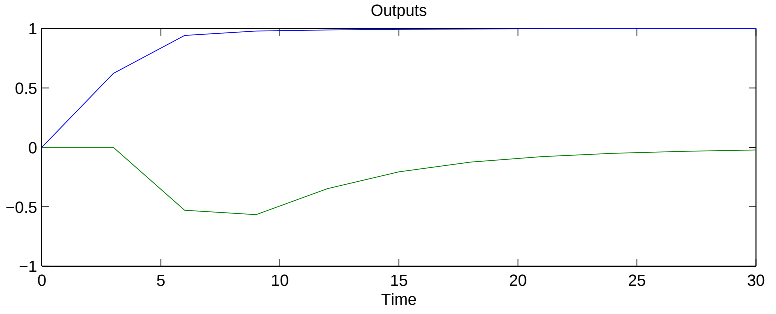
See the `mpccl` example for the commands that build the model and a simple controller for this process.

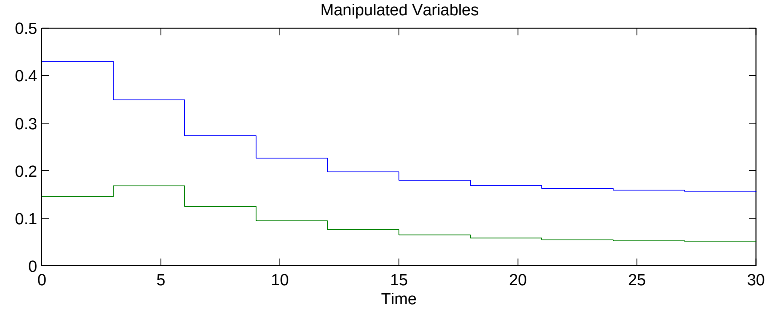
Here is a slightly more complex design with blocking and time-varying weights on the manipulated and output variables:

```
P=6; M=[2 4];
uwt=[1 0; 0 1];
ywt=[1 0.1; 0.8 0.1; 0.1 0.1];
Kmpc=mpccon(model,ywt,uwt,M,P);
tend=30; r=[1 0];
[y,u]=mpcsim(plant,model,Kmpc,tend,r);
```

There is no particular rationale for using time varying weights in this case it is only for illustration. The manipulated variables will make 2 moves during the prediction horizon (see value of `M`, above). The `uwt` selection gives u_1 a unity weight and u_2 a zero weight for the first move, then switches the weights for the second move. If there had been any additional moves they would have had the same weighting as the second move.

The ywt value assigns a constant weight of 0.1 to y_2, and a weight that decreases over the first 3 periods to y_1. The weights for periods 4 to 6 are the same as for period 3. The resulting closed-loop (servo) response is:





See Also

`cmpc`, `mpccl`, `mpcsim`

Purpose Determines the type of a matrix and returns information about the matrix.

Syntax:

```
mpcinfo(mat)
```

Description

`mpcinfo` returns information about the type and size of the matrix, `mat`. The information is determined from the matrix structure. The matrix types include MPC *step* format, MPC **mod** format, *varying* format and constant. `mpcinfo` returns text output to the screen.

If the matrix is in MPC *step* format, the output includes the sampling time used to create the model, number of inputs, number of outputs and number of step response coefficients; it also indicates which outputs are stable and which outputs are integrating.

If the matrix is in MPC **mod** format, the output includes the sampling time used to create the model, number of states, number of manipulated variable inputs, number of measured disturbances, number of unmeasured disturbances, number of measured outputs and number of unmeasured outputs.

For a matrix in *varying* format, as formed in `mod2frsp`, the number of independent variable values, and the number of rows and number of columns of each submatrix are output.

For a constant matrix, the text output consists of the number of rows and number of columns.

Examples

1 MPC *step* format: After running the `mod2step` example `mpcinfo(plant)` returns:

```
This is a matrix in MPC Step format.
sampling time = 1.5
number of inputs = 3
number of outputs = 4
number of step response coefficients = 3
All outputs are stable.
```

2 MPC **mod** format: After running the `ss2mod` example `mpcinfo(pmod)` returns:

```
This is a matrix in MPC Mod format.
    minfo = [2 3 1 1 1 1 0 ]
sampling time = 2
number of states = 3
number of manipulated variable inputs = 1
number of measured disturbances = 1
number of unmeasured disturbances = 1
number of measured outputs = 1
number of unmeasured outputs = 0
```

3 *varying* format: After running the `mod2frsp` example `mpcinfo(eyefrsp)` returns:

```
varying: 30 pts 2 rows 2 cols
```

See Also

`mod`, `step`, `mod2frsp`, `varying format`

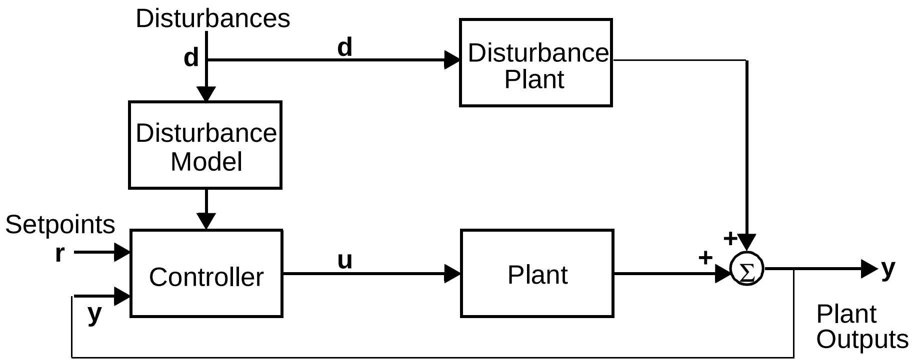
Purpose

Simulates closed-loop systems with saturation *constraints* on the manipulated variables using models in the MPC *step* format. Can also be used for open-loop simulations.

Syntax

```
yp = mpcsim(plant,model,Kmpc,tend,r)
[yp,u,ym] = mpcsim(plant,model,Kmpc,tend,r,usat,...
      tfilter,dplant,dmodel,dstep)
```

Description



`mpcsim` provides a convenient way to simulate the performance of the type of system shown in the above diagram. Measurement noise can be simulated by treating it as an unmeasured disturbance. The required input variables are as follows:

`plant`
Is a model in the MPC *step* format that is to represent the plant.

`model`
Is a model in the MPC *step* format that is to be used for state estimation in the controller. In general, it can be different from `plant` if you want to simulate the effect of plant/controller model mismatch. Note, however, that `model` should be the same as that used to calculate `Kmpc`.

`Kmpc`
Is the MPC controller gain matrix, usually calculated using the function `mpccon`.

If you set `Kmpc` to an empty matrix, `mpcsim` will do an open-loop simulation. Then the inputs to the plant will be r (which must be set to the vector of manipulated variables in this case) and `dstep`.

`tend`

Is the desired duration of the simulation (in time units).

`r`

Is normally a setpoint matrix consisting of N rows and n_y columns, where n_y is the number of output variables, *y:*

$$\mathrm{r} = \begin{bmatrix} r_1(1) & r_2(1) & \dots & r_{n_y}(1) \\ r_1(2) & r_2(2) & \dots & r_{n_y}(2) \\ \vdots & \vdots & \dots & \vdots \\ r_1(N) & r_2(N) & \dots & r_{n_y}(N) \end{bmatrix}$$

where $r_i(k)$ is the setpoint for output j at time $t = kT$, and T is the sampling period (as specified in the *step* format of `plant` and `model`). If `tend` > NT, the setpoints vary for the first N periods in the simulation, as specified by `r`, and are then held constant at the values given in the last row of `r` for the remainder of the simulation.

In many simulations one wants the setpoints to be constant for the entire time, in which case `r` need only contain a single row of n_y values.

If you set `r=[ ]`, the default is a row of n_y zeros.

For open-loop simulations, r specifies the manipulated variables and must contain n_u columns.

The following input variables are optional. In general, setting one of them equal to an empty matrix causes `mpcsim` to use the default value, which is given in the description.

usat

Is a matrix giving the limits on the manipulated variables. Its format is as follows:

$$\text{usat} = \left[\begin{bmatrix} u_{min,1}(1) & \dots & u_{min,n_u}(1) \\ u_{min,1}(2) & \dots & u_{min,n_u}(2) \\ \vdots & \dots & \vdots \\ u_{min,1}(N) & \dots & u_{min,n_u}(N) \end{bmatrix} \begin{bmatrix} u_{max,1}(1) & \dots & u_{max,n_u}(1) \\ u_{max,1}(2) & \dots & u_{max,n_u}(2) \\ \vdots & \dots & \vdots \\ u_{max,1}(N) & \dots & u_{max,n_u}(N) \end{bmatrix} \begin{bmatrix} \Delta u_{max,1}(1) & \dots & \Delta u_{max,n_u}(1) \\ \Delta u_{max,1}(2) & \dots & \Delta u_{max,n_u}(2) \\ \vdots & \dots & \vdots \\ \Delta u_{max,1}(N) & \dots & \Delta u_{max,n_u}(N) \end{bmatrix}\right]$$

Note that it contains three matrices of N rows. N may be different than that for the setpoint matrix, `r`, but the idea is the same: the saturation limits will vary for the first N sampling periods of the simulation, then be held constant at the values given in the last row of `usat` for the remaining periods (if any).

The first matrix specifies the *lower bounds* on the n_u manipulated variables. For example, $u_{min,j}(k)$ is the lower bound for manipulated variable j at time $t = kT$ in the simulation. If $u_{min,j}(k) = -inf$, manipulated variable j will have no lower bound at $t = kT$.

The second matrix gives the *upper bounds* on the manipulated variables. If $u_{max,j}(k) = inf$, manipulated variable j will have no upper bound at $t = kT$.

The lower and upper bounds may be either positive or negative (or zero) as long as $u_{min,j}(k) \le u_{max,j}(k)$.

The third matrix gives the limits on the rate of change of the manipulated variables. In other words, `mpcsim` will force $|u_j(k) - u_j(k-1)| \leq \Delta u_{max,j}(k)$. The limits on the rate of change must be nonnegative.

The default is no saturation constraints, i.e., all the u_{min} values will be set to $-inf$, and all the u_{max} and Δu_{max} values will be set to inf.

Note: Saturation constraints in `mpcsim` are enforced by simply *clipping* the manipulated variable moves so that they satisfy all constraints. This is a nonoptimal solution that, in general, will differ from the results you would get using the `ulim` variable in `cmpc`.

`tfilter`

Is a matrix of time constants for the noise filter and the unmeasured disturbances entering at the plant output. The first row of n_y elements gives the noise filter time constants and the second row of n_y elements gives the time constants of the lags through which the unmeasured disturbance steps pass. If `tfilter` only contains one row, the unmeasured disturbances are assumed to be steps. If you set `tfilter=[ ]` or omit it, the default is no noise filtering and steplike unmeasured disturbances.

`dplant`

Is a model in MPC *step* format representing all the disturbances (measured and unmeasured) that affect `plant` in the above diagram. If `dplant` is provided, then input `dstep` is also required. For output step disturbances, set `dplant=[ ]`. The default is no disturbances.

`dmodel`

Is a model in MPC *step* format representing the measured disturbances. If `dmodel` is provided, then input `dstep` is also required. If there are no measured disturbances, set `dmodel=[ ]`. For output step disturbances, set `dmodel=[ ]`. If there are both measured and un- measured disturbances, set the columns of `dmodel` corresponding to the unmeasured disturbances to zero. The default is no measured disturbances.

dstep

Is a matrix of disturbances to the plant. For output step disturbances (`dplant=[ ]` and `dmodel=[ ]`), the format is the same as for `r`. For disturbances through step-response models (`dplant` only or both `dplant` and `dmodel` nonempty), the format is the same as for `r`, except that the number of columns is n_d rather than n_y. The default is a row of zeros.

Note: You may use a different number of rows in the matrices r, `usat` and `dstep`, should that be appropriate for your simulation.

The calculated outputs are as follows (all but `yp` are optional):

yp

Is a matrix containing M rows and n_y columns, where M = `round(tend/delt2) + 1` and `delt2` is the sampling time. The first row will contain the initial condition, and row $k - 1$ will give the values of the plant outputs, y (see above diagram), at time $t = kT$.

u

Is a matrix containing the same number of rows as `yp` and n_u columns. The time corresponding to each row is the same as for `yp`. The elements in each row are the values of the manipulated variables, u (see above diagram).

ym

Is a matrix of the same structure as `yp`, containing the values of the predicted outputs from the state estimator in the controller. `ym` will, in general, differ from `yp` if `model¦plant` and/or there are unmeasured disturbances. The prediction includes the effect of the most recent measurement, i.e.,$\hat{y}(k|k)$.

Examples

Consider the linear system:

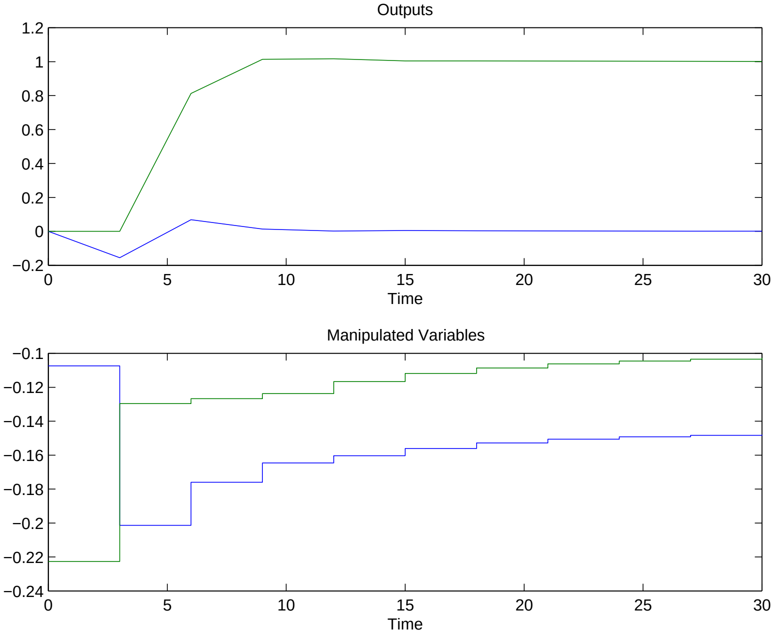
$$\begin{bmatrix} y_1(s) \\ y_2(s) \end{bmatrix} = \begin{bmatrix} \dfrac{12.8e^{-s}}{16.7s+1} & \dfrac{-18.9e^{-3s}}{21.0s+1} \\ \dfrac{6.6e^{-7s}}{10.9s+1} & \dfrac{-19.4e^{-3s}}{14.4s+1} \end{bmatrix} \begin{bmatrix} u_1(s) \\ u_2(s) \end{bmatrix}$$

The following statements build the model and calculate the MPC controller gain:

```
g11=poly2tfd(12.8,[16.7 1],0,1);
g21=poly2tfd(6.6,[10.9 1],0,7);
g12=poly2tfd(-18.9,[21.0 1],0,3);
g22=poly2tfd(-19.4,[14.4 1],0,3);
delt=3; ny=2; tfinal=90;
model=tfd2step(tfinal,delt,ny,g11,g21,g12,g22);
plant=model;
P=6; M=2;
ywt=[ ]; uwt=[1 1];
Kmpc=mpccon(imod,ywt,uwt,M,P);
```

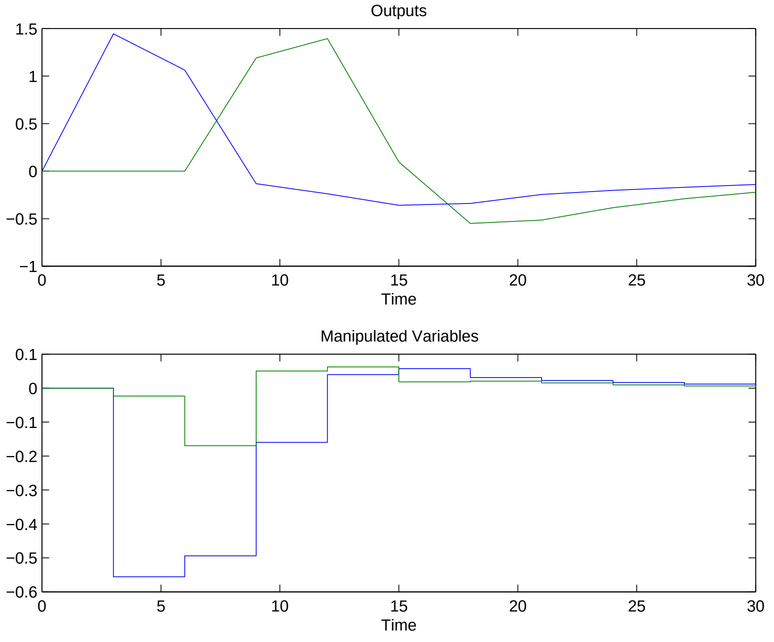
Simulate and plot the closed-loop performance for a unit step in the setpoint for y_2, occurring at $t = 0$.

```
tend=30; r=[0 1];
[y,u]=mpcsim(plant,model,Kmpc,tend,r);
plotall(y,u,delt),pause
```



Try a pulse change in the disturbance that adds to u_1:

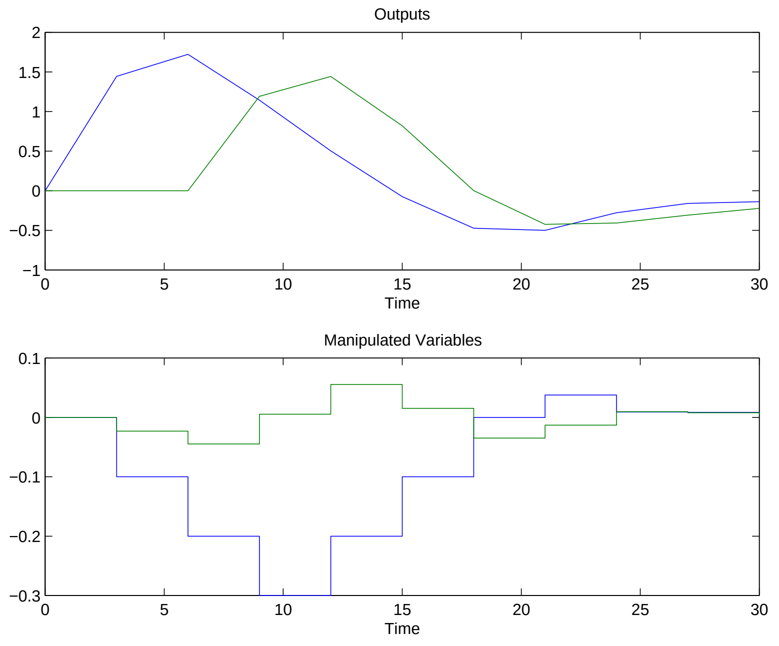
```
r=[ ]; usat=[ ]; tfilter=[ ]; dmodel=[ ];
dplant=plant;
dstep=[ 1 0; 0 0];
[y,u]=mpcsim(plant,model,Kmpc,tend,r,usat,tfilter,...
    dplant,dmodel,dstep);
plotall(y,u,delt),pause
```



For the same disturbance as in the previous case, limit the rates of change of both manipulated variables.

```
usat=[-inf -inf inf inf 0.1 0.05];
[y,u]=mpcsim(plant,model,Kmpc,tend,r,usat,tfilter,...
```

```
   dplant,dmodel,dstep);
plotall(y,u,delt),pause
```



Restriction

Initial conditions of zero are used for all the variables. This simulates the condition where all variables represent a deviation from a steady-state initial condition.

See Also

`plotall`, `ploteach`, `cmpc`, `mpccl`, `mpccon`

Purpose

Model predictive controller for simulating closed-loop systems with hard bounds on manipulated variables and/or controlled variables using linear models in the MPC *step* format for *nonlinear* plants represented as Simulink S-functions.

Description

`nlcmpc` is a Simulink S-function block and can be invoked by typing `nlmpclib` at the MATLAB prompt. Its usage is identical to other Simulink blocks. The input to `nlcmpc` includes both the variables controlled by `nlcmpc` and measured disturbances. The first n_y elements of the input are treated as the controlled variables while the rest is taken as the measured disturbances. The output from `nlcmpc` are the values of the manipulated variables. Initial conditions for the manipulated variables and the measured disturbances must be specified. The controlled variables sent to `nlcmpc` and the manipulated variables returned by `nlcmpc` are actual variables; they are not deviation variables.

Because of the limit on the number of masked variables that can be specified for a Simulink block, `model` and `dmodel` are put together as “one” variable, `r`, `ywt`, and `uwt` as “one” variable, and `ylim` and `ulim` as “one” variable. `m` and `p` should be entered as one *row vector*. `u0` and `d0` should also be entered as one *row vector*. The required input variables are as follows:

modelpd

Equals [`model dmodel`]. `model` is a *linear* model in the MPC *step* format that is to be used for state estimation in the controller. In general, it is a linear approximation of the nonlinear plant. `dmodel` is a model in MPC *step* format representing the effect of the measured disturbances. The default is no measured disturbances. Note that the truncation time for `model` and `dmodel` must be the same and the number of outputs for `model` and `dmodel` must be the same.

ryuwt
Equals `[r ywt uwt]`. `r` is a setpoint matrix consisting of N rows and n_y columns, where n_y is the number of output variables, y:

$$\mathrm{r} = \begin{bmatrix} r_1(1) & r_2(1) & \ldots & r_{n_y}(1) \\ r_1(2) & r_2(2) & \ldots & r_{n_y}(2) \\ \vdots & \vdots & \ldots & \vdots \\ r_1(N) & r_2(N) & \ldots & r_{n_y}(N) \end{bmatrix}$$

Where $r_i(k)$ is the setpoint for output i at time $t = kT$, and T is the sampling period (as specified in the *step* format of model). If the simulation time is larger than NT, the setpoints vary for the first N periods in the simulation, as specified by `r`, and are then held constant at the values given in the last row of `r` for the remainder of the simulation. In many simulations one wants the setpoints to be constant for the entire time, in which case `r` need only contain a single row of n_y values.

ywt
Must have n_y columns, where n_y is the number of outputs. All weights must be ≥ 0.

You may vary the weights at each step in the prediction horizon by including up to `P` rows in `ywt`. Then the first row of n_y values applies to the tracking errors in the first step in the prediction horizon, the next row applies to the next step, etc. See `mpccon` for details on the form of the optimization objective function.

If you supply only *nrow* rows, where $1 \leq nrow <$ `P`, `nlcmpc` will use the last row to fill in any remaining steps. Thus if you wish the weighting to be the same for all `P` steps, you need only specify a single row.

uwt
Has the same format as `ywt`, except that `uwt` applies to the changes in the manipulated variables. If `uwt¦[ ]`, it must have n_u columns, where n_u is the number of manipulated variables.

Notice that the number of rows for `r`, `ywt`, and `uwt` should be the same. If not, one can enter the variable as `parpart(r, ywt, uwt)`. The function `parpart` appends extra rows to `r`, `ywt`, and/or `uwt` so that they have the same number of rows. The default is `r=y0`, where y0 is the initial condition for the output, equal (unity) weighting of all outputs over the entire prediction horizon and zero weighting of all input.

mp

Equals `[M P]`. `P` equals the last element of `MP`. There are two ways to specify `M`: If it is a *scalar*, `nlcmpc` interprets it as the input horizon (number of moves) as in DMC; if it is a *row vector* containing n_b elements, each element of the vector indicates number of the steps over which $\Delta u(k) = 0$ during the optimization and `nlcmpc` interprets it as a set of n_b blocking factors. There may be $1 \le n_b \le$ `P` blocking factors, and their sum must be ≤ `P`. If you set `M=[ ]`, the default is `M = P`, which is equivalent to `M=ones(1,P)`. `P` is the number of sampling periods in the prediction horizon.

yulim

Equals [`ylim ulim`]. `ulim` is a matrix giving the limits on the manipulated variables. Its format is as follows:

$$\text{ulim} = \left[\begin{bmatrix} u_{min,1}(1) & \cdots & u_{min,n_u}(1) \\ u_{min,1}(2) & \cdots & u_{min,n_u}(2) \\ \vdots & \cdots & \vdots \\ u_{min,1}(N) & \cdots & u_{min,n_u}(N) \end{bmatrix} \begin{bmatrix} u_{max,1}(1) & \cdots & u_{max,n_u}(1) \\ u_{max,1}(2) & \cdots & u_{max,n_u}(2) \\ \vdots & \cdots & \vdots \\ u_{max,1}(N) & \cdots & u_{max,n_u}(N) \end{bmatrix} \begin{bmatrix} \Delta u_{max,1}(1) & \cdots & \Delta u_{max,n_u}(1) \\ \Delta u_{max,1}(2) & \cdots & \Delta u_{max,n_u}(2) \\ \vdots & \cdots & \vdots \\ \Delta u_{max,1}(N) & \cdots & \Delta u_{max,n_u}(N) \end{bmatrix}\right]$$

Note that it contains three matrices of N rows. In this case, the limits on N are $1 \le N \le n_b$, where n_b is the number of times the manipulated variables are to change over the input horizon. If you supply fewer than n_b rows, the last row is repeated automatically.

The first matrix specifies the *lower bounds* on the n_u manipulated variables. For example, $u_{min},_j(2)$ is the lower bound for manipulated variable j for the second move of the manipulated variables (where the first move is at the start of the prediction horizon). If $u_{min},_j(k) = -inf$, manipulated variable j will have no lower bound for that move.

The second matrix gives the *upper bounds* on the manipulated variables. If $u_{max,j}(k) = inf$, manipulated variable j will have no upper bound for that move.

The lower and upper bounds may be either positive or negative (or zero) as long as $u_{min},_j(k) \le u_{max,j}(k)$.

The third matrix gives the limits on the rate of change of the manipulated variables. In other words, `cmpc` will force $|u_j(k) - u_j(k-1)| \le \Delta u_{max,j}(k)$. The limits on the rate of change must be nonnegative and *finite*. If you want it to be unbounded, set the bound to a large number (but not too large — a value of 10^6 should work well in most cases).

`ylim` has the same format as `ulim`, but for the lower and upper bounds of the outputs. The first row applies to the first point in the prediction horizon.

Note that the number of rows for `ylim` and `ulim` should be the same. If the number of rows for `ylim` and `ulim` differs, one can use `parpart(ylim, ulim)`. The function `parpart` appends extra rows to `ylim` or `ulim` so that they have the same number of rows. If you set `yulim = [ ]`, then $u_{min} = -inf$, $u_{max} = inf$, $\Delta u_{max} = 10^6$, $y_{min} = -inf$ and $y_{max} = inf$.

`tfilter`

Is a matrix of time constants for the noise filter and the unmeasured disturbances entering at the plant output. The first row of n_y elements gives the noise filter time constants and the second row of n_y elements gives the time constants of the lags through which the unmeasured disturbance steps pass. If `tfilter` only contains one row, the unmeasured disturbances are assumed to be steps. If you set `tfilter= [ ]`, no noise filtering and steplike unmeasured disturbances are assumed.

ud0

Equals [u0 d0]. u0 are initial values of the manipulated variables arranged in a *row vector* having n_u elements; n_u is the number of the manipulated variables computed by nlcmpc. d0 are initial values of the measured disturbances arranged in a *row vector* having n_d elements; n_d is the number of the measured disturbances. The default is u0 = 0 and d0 = 0.

Notes

- Initial conditions for the manipulated variables that are calculated by nlcmpc are specified through nlcmpc while initial conditions for the controlled variables are specified through the S-function for the nonlinear plant.
- You may use a different number of rows in the matrices r, ulim and ylim, should that be appropriate for your simulation.
- The ulim constraints used here are fundamentally different from the usat constraints used in the nlmpcsim block. The ulim constraints are defined relative to the beginning of the prediction horizon, which moves as the simulation progresses. Thus at each sampling period, k, the ulim constraints apply to a block of calculated moves that begin at sampling period k and extend for the duration of the input horizon. The usat constraints, on the other hand, are relative to the fixed point $t = 0$, the start of the simulation.
- For unconstrained problems, nlcmpc and nlmpcsim should give the same results. The latter will be faster because it uses an analytical solution of the QP problem, whereas nlcmpc solves it by iteration.

Example

See the examples for nlmpcsim with one modification: replace the block nlmpcsim with nlcmpc. Clearly, additional variables should be defined appropriately.

See Also

cmpc, nlmpcsim

Purpose

Model predictive controller for simulating closed-loop systems with *saturation constraints* on the manipulated variables using linear models in the MPC *step* format for nonlinear plants represented as Simulink S-functions.

Description

`nlmpcsim` is a Simulink S-function block and can be invoked by typing `nlmpclib` at the MATLAB prompt. Its usage is identical to other Simulink blocks. The input to `nlmpcsim` includes both the variables controlled by `nlmpcsim` and measured disturbances. The first n_y elements of the input are treated as the controlled variables while the rest is taken as the measured disturbances. The output from `nlmpcsim` are the values of the manipulated variables. Initial conditions for the manipulated variables and the measured disturbances must be specified. Both the controlled variables sent to `nlmpcsim` and the manipulated variables returned by `nlmpcsim` are the actual variables; they are not deviation variables.

Because of the limit on the number of masked variables that can be specified for a Simulink block, `model` and `dmodel` are put together as one variable. `u0` and `d0` should be entered as one *row vector*. The required input variables are as follows:

`modelpd`

Equals [`model` `dmodel`]. model is a linear model in the MPC *step* format that is to be used for state estimation in the controller. In general, it is a linear approximation for the nonlinear plant. Note, however, that `model` should be the same as that used to calculate `Kmpc`. `dmodel` is a model in MPC *step* format representing the measured disturbances. If `dmodel = [ ]`, the default is no measured disturbances. Note that the truncation time for `model` and `dmodel` should be the same and the number of outputs for `model` and `dmodel` should be the same.

`Kmpc`

Is the MPC controller gain matrix, usually calculated using the function `mpccon`.

r

Is a setpoint matrix consisting of N rows and n_y columns, where n_y is the number of controlled variables, y:

$$\mathrm{r} = \begin{bmatrix} r_1(1) & r_2(1) & \ldots & r_{n_y}(1) \\ r_1(2) & r_2(2) & \ldots & r_{n_y}(2) \\ \vdots & \vdots & \ldots & \vdots \\ r_1(N) & r_2(N) & \ldots & r_{n_y}(N) \end{bmatrix}$$

Where $r_i(k)$ is the setpoint for output i at time $t = kT$, and T is the sampling period (as specified in the *step* format of model). If the simulation time is larger than NT, the setpoints vary for the first N periods in the simulation, as specified by `r`, and are then held constant at the values given in the last row of `r` for the remainder of the simulation.

In many simulations one wants the setpoints to be constant for the entire time, in which case `r` need only contain a single row of n_y values.

Note that `r` is the actual setpoint. If you set `r=[ ]`, the default is `y0`.

usat

Is a matrix giving the saturation limits on the manipulated variables. Its format is as follows:

$$\text{usat} = \left[\begin{bmatrix} u_{min,1}(1) & \cdots & u_{min,n_u}(1) \\ u_{min,1}(2) & \cdots & u_{min,n_u}(2) \\ \vdots & \cdots & \vdots \\ u_{min,1}(N) & \cdots & u_{min,n_u}(N) \end{bmatrix} \begin{bmatrix} u_{max,1}(1) & \cdots & u_{max,n_u}(1) \\ u_{max,1}(2) & \cdots & u_{max,n_u}(2) \\ \vdots & \cdots & \vdots \\ u_{max,1}(N) & \cdots & u_{max,n_u}(N) \end{bmatrix} \begin{bmatrix} \Delta u_{max,1}(1) & \cdots & \Delta u_{max,n_u}(1) \\ \Delta u_{max,1}(2) & \cdots & \Delta u_{max,n_u}(2) \\ \vdots & \cdots & \vdots \\ \Delta u_{max,1}(N) & \cdots & \Delta u_{max,n_u}(N) \end{bmatrix} \right]$$

Note that it contains three matrices of N rows. N may be different from that for the setpoint matrix, `r`, but the idea is the same: the saturation limits will vary for the first N sampling periods of the simulation, then be held constant at the values given in the last row of `usat` for the remaining periods (if any).

The first matrix specifies the *lower bounds* on the n_u manipulated variables. For example, $u_{min,j}(k)$ is the lower bound for manipulated variable j at time $t = kT$ in the simulation. If $u_{min,j}(k) = -inf$, manipulated variable j will have no lower bound at $t = kT$.

The second matrix gives the *upper bounds* on the manipulated variables. If $u_{max,j}(k) = inf$, manipulated variable j will have no upper bound at $t = kT$.

The lower and upper bounds may be either positive or negative (or zero) as long as $u_{min,j}(k) \le u_{max,j}(k)$.

The third matrix gives the limits on the rate of change of the manipulated variables. In other words, mpcsim will force $|u_j(k) - u_j(k-1)| \le \Delta u_{max,j}(k)$. The limits on the rate of change must be nonnegative.

If usat = [ ], then all the u_{min} values will be set to $-inf$, and all the u_{max} and u_{max} values will be set to inf.

Note: Saturation constraints are enforced by simply *clipping* the manipulated variable moves so that they satisfy all constraints. This is a nonoptimal solution that, in general, will differ from the results you would get using the ulim variable in cmpc or nlcmpc.

---

tfilter

Is a matrix of time constants for the noise filter and the unmeasured disturbances entering at the plant output. The first row of n_y elements gives the noise filter time constants and the second row of n_y elements gives the time constants of the lags through which the unmeasured disturbance steps pass. If tfilter only contains one row, the unmeasured disturbances are assumed to be steps. If you set tfilter= [ ], no noise filtering and steplike unmeasured disturbances are assumed.

ud0

Equals [u0 d0]. u0 are initial values of the manipulated variables arranged in a *row vector* having n_u elements; n_u is the number of the manipulated variables computed by nlmpcsim. d0 are initial values of the measured disturbances arranged in a *row vector* having n_d elements; n_d is the number of the measured disturbances. The default is u0 = 0 and d0 = 0.

Note: You may use a different number of rows in the matrices r and usat, should that be appropriate for your simulation.

Examples

Let us now demonstrate the use of the controller nlmpcsim. Since the plant used in Example 1 is linear, using mpcsim would be much faster. The point, however, is to show how masked variables are specified for nlmpcsim.

1 The plant is linear with two inputs and two outputs. It is represented by

$$\frac{dx}{dt} = \begin{bmatrix} -1.2 & 0 \\ 0 & -\frac{1}{1.5} \end{bmatrix} x + \begin{bmatrix} 0.2 \\ 1 \end{bmatrix} u + \begin{bmatrix} 50 \\ 0 \end{bmatrix}$$

$$y = x$$

The Simulink S-function for this plant is in `mpcplant.m`. The nominal steady-state operating condition is `y0 = [58.3 1.5]` and `u0 = [100 1]`. The Simulink block to simulate this plant using `nlmpcsim` is in `nlmpcdm1.m` and shown in Figure 1-1.

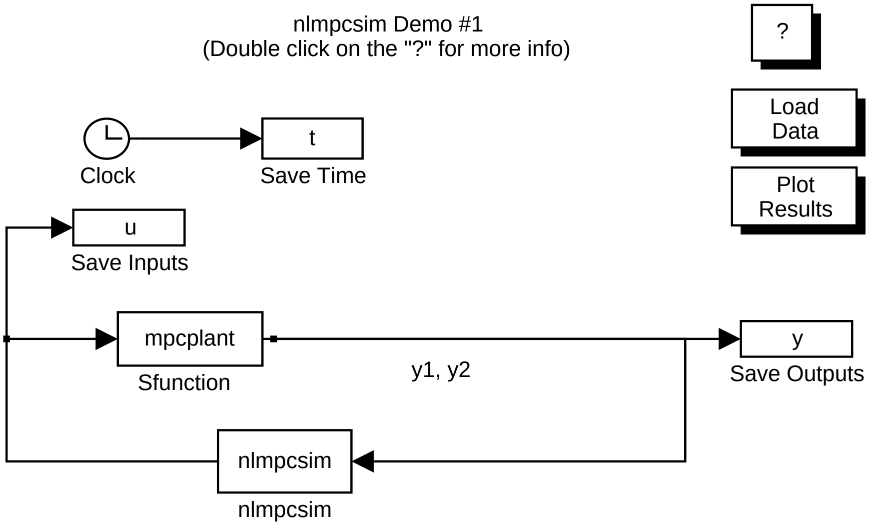


Figure 1-1 Simulink Block for Example 1

The following statements build the step response model and specify the parameter values. Note that `model` does not equal the plant model stored in mpcplant.m. The important thing to notice is that both r and usat are actual variables. They are not deviation variables.

```
g11=poly(0.4,[1 2]);
g21=poly2tfd(0,1);
g12=poly2tfd(0,1);
g22=poly2tfd(1,[1 1]);
tfinal=8;
delt=0.2;
nout=2;
model=tfd2step(tfinal,delt,nout,g11,g21,g12,g22);
ywt=[1 1];
uwt=[0 0];
M=4;
P=10;
r=[68.3 2];
usat=[100 1 200 3 200 200];
tfilter=[ ];
Kmpc = mpccon(model,ywt,uwt,M,P);
dmodel = [ ];
```

There are two ways to simulate the closed loop system. We can set the simulation parameters and click on **Start** under **Simulation** or via the following statements.

```
plant='nlmpcdm1'; y0=[58.3 1.5];
u0=[100 1];
tfsim = 2;
tol=[1e-3];
minstep=[ ];
maxstep=[ ];
[t,yu]=gear(plant,tfsim,[y0 u0],[tol,minstep,maxstep]);
```

Figure 1-2 shows the response for the setpoint change.

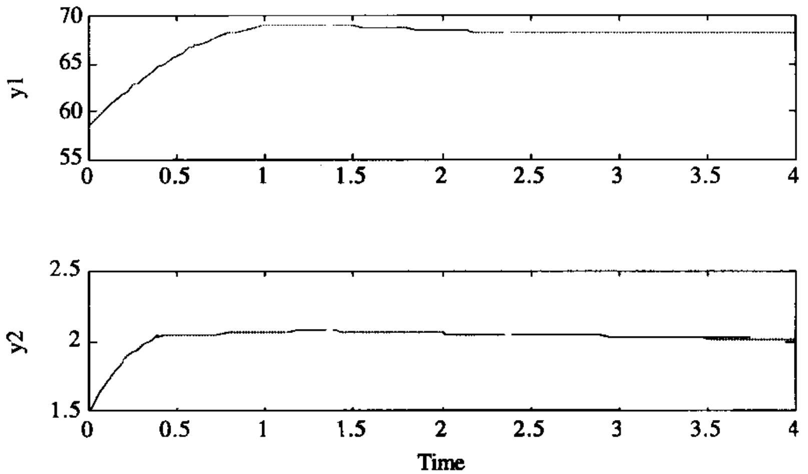


Figure 1-2 Output responses for a setpoint change for Example 1

2 The plant is the paper machine headbox discussed in the section, "Application: Paper Machine Headbox Control" in Chapter 3. The nonlinear plant model is represented as a Simulink S-function and is in `pap_mach.m`. The plant has two inputs, three outputs, four states, one measured disturbance, and one unmeasured disturbance. All these variables are zero at the nominal steady-state. Since the model for `nlmpcsim` must be linear, we linearize the nonlinear plant at the nominal steady-state to obtain a linear model. Since the model is simple, we can linearize it analytically to obtain A, B, C, and D.

The Simulink block to simulate this nonlinear plant using `nlmpcsim` is in `nlmpcdm2.m` and shown in Figure 1-3.

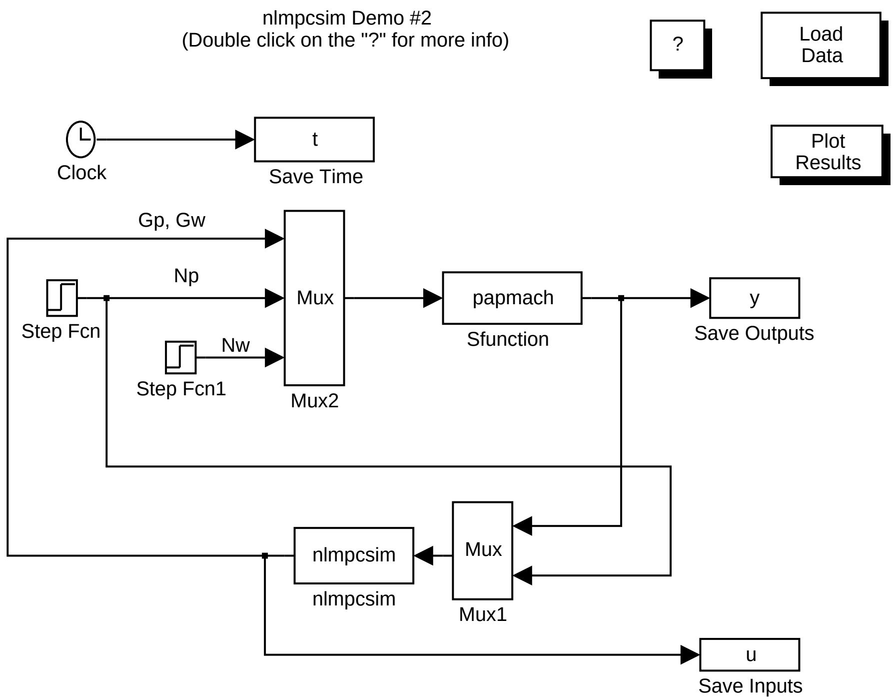


Figure 1-3 Simulink Block for Example 2

The following statements build the step response model and specify the parameter values.

```
A=[-1.93 0 0 0; .394 -.426 0 0; 0 0 -.63 0; .82 -.784
   .413 -.426];
B=[1.274 1.274 0; 0 0 0; 1.34 -.65 .203; 0 0 0];
C=[0 1 0 0; 0 0 1 0; 0 0 0 1];
D=zeros(3,3);
% Discretize the linear model and save in MOD form.
dt=2;
[PHI,GAM]=c2dmp(A,B,dt);
minfo=[dt,4,2,1,0,3,0];
imod=ss2mod(PHI,GAM,C,D,minfo);
% Store plant model and measured disturbance model in MPC
% step format
```

```
[model,dmodel]=mod2step(imod,20);
m=5;
p=20;
ywt=[1 0 5]; % unequal weighting of y1 and y3, no control
% of y2
uwt=[1 1]; % Equal weighting of u1 and u2
ulim=[-10 -10 10 10 2 2]; % Constraints on u
ylim=[ ]; % No constraints on y
usat=ulim;
tfilter=[ ];
y0=[0 0 0];
u0=[0 0];
r=[0 0 0];
Kmpc=mpccon(model,ywt,uwt,M,P);
```

Figure 1-4 shows the output responses for a unit-step measured disturbance $Np = 1$ and a step unmeasured disturbance with $Nw = 5$.

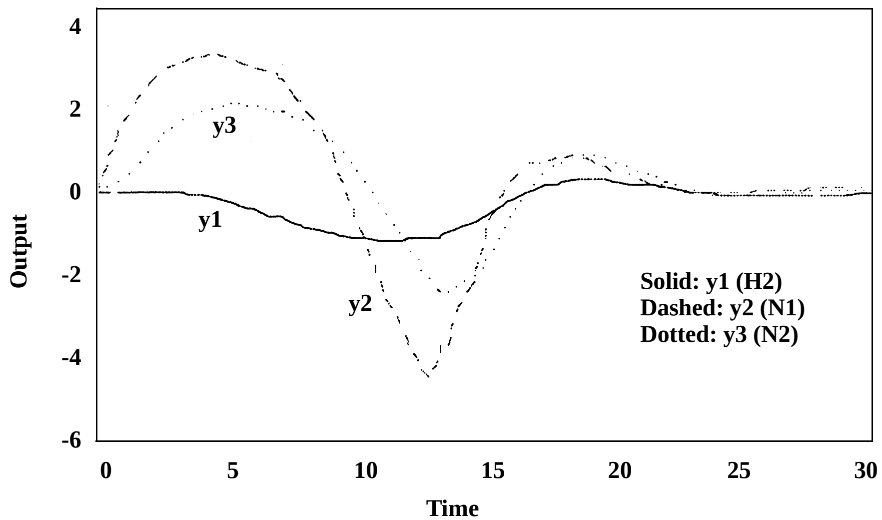


Figure 1-4 Output responses for a unit-step measured disturbance $Np = 1$ **and a step unmeasured disturbance** $Nw = 5$

See Also

`mpcsim`, `nlcmpc`

paramod

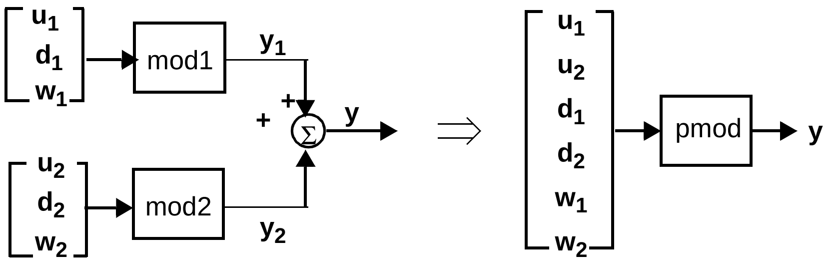
Purpose

Puts two models in parallel by connecting their outputs. Mimics the utility function mpcparal, except that paramod works on models in the MPC **mod** format.

Syntax

```
pmod = paramod(mod1,mod2)
```

Description



mod1 and mod2 are models in the MPC **mod** format (see mod in the online *MATLAB Function Reference* format section for a detailed description). You would normally create them using either the tfd2mod, ss2mod or th2mod functions.

paramod combines them to form a composite system, pmod, as shown in the above diagram. It is also in the **mod** format. Note how the inputs to mod1 and mod2 are ordered in pmod.

Restriction

mod1 and mod2 must have been created with equal sampling periods and they must have the same number of measured and unmeasured outputs.

See Also

addmd, addmod, addumd, appmod, sermod

plotall

Purpose Plots outputs and manipulated variables from a simulation, all on one "page."

Syntax

```
plotall(y,u) plotall(y,u,t)
```

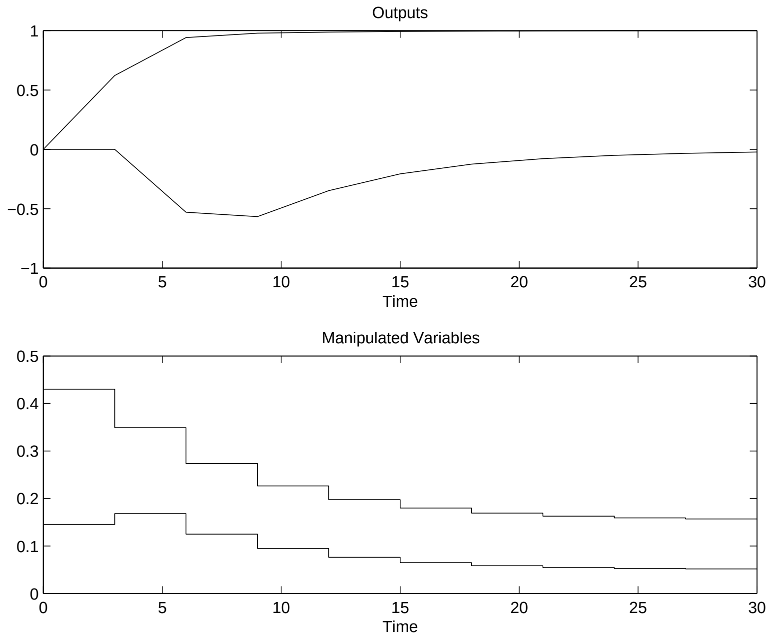
Description Input variables `y` and `u` are matrices of outputs and manipulated variables, respectively. Each row represents a sample at a particular time. Each column shows how a particular output (or manipulated) variable changes with time.

Input variable `t` is optional. If you supply it as a scalar, `plotall` interprets is as the sampling period, and calculates the time axis for the plots accordingly. It can also be a column vector, in which case it must have the same number of rows as `y` and `u` and is interpreted as the times at which the samples of `y` and `u` were taken. If you do not supply `t`, `plotall` uses a sampling period of 1 by default.

`plotall` plots all the outputs on a single graph. If there are multiple outputs that have very different numerical scales, this may be unsatisfactory. In that case, use `ploteach`.

`plotall` plots all the manipulated variables in "stairstep" form (i.e., assuming a zero-order hold) on a single graph. Again, `ploteach` may be the better choice if scales are very different.

Example output: (`mpccon` example)



See Also

`ploteach`, `plotstep`, `plotfrsp`

Purpose

Plots outputs and manipulated variables from a simulation on separate graphs, up to four per page.

Syntax

```
ploteach(y)
ploteach(y,u)
ploteach([ ],u)
ploteach(y,[ ],t)
ploteach([ ],u,t)
ploteach(y,u,t)
```

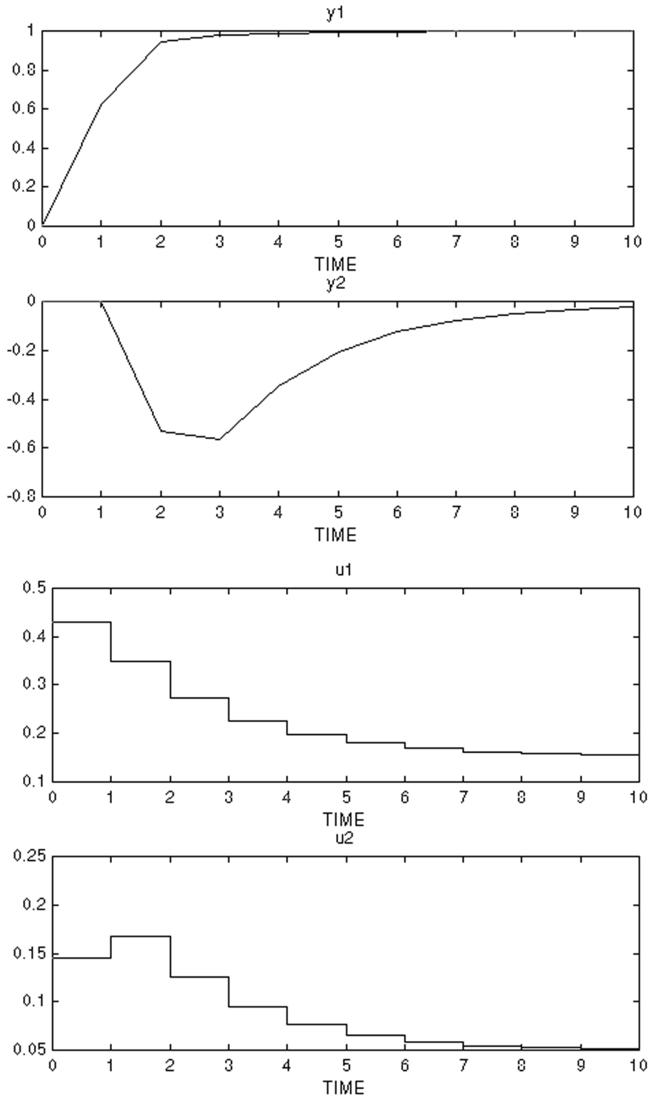
Description

Input variables `y` and `u` are matrices of outputs and manipulated variables, respectively. Each row represents a sample at a particular time. Each column shows how a particular output (or manipulated) variable changes with time. As shown above, you may supply both `y` and `u`, or omit either one of them.

Input variable `t` is optional. If you supply it as a scalar, `ploteach` interprets is as the *sampling period*, and calculates the time axis for the plots accordingly. It can also be a column vector, in which case it must have the same number of rows as `y` and `u` and is interpreted as the times at which the samples of `y` and `u` were taken. If you do not supply `t`, `ploteach` uses a sampling period of 1 by default.

`ploteach` plots the manipulated variables in “stairstep” form (i.e., assuming a zero-order hold).

Example output: (mpccon example)



See Also `plotall`, `plotfrsp`, `plotstep`

Purpose

Plots the frequency response generated by `mod2frsp` as a Bode plot.

Syntax

```
plotfrsp(vmat)
plotfrsp(vmat,out,in)
```

Description

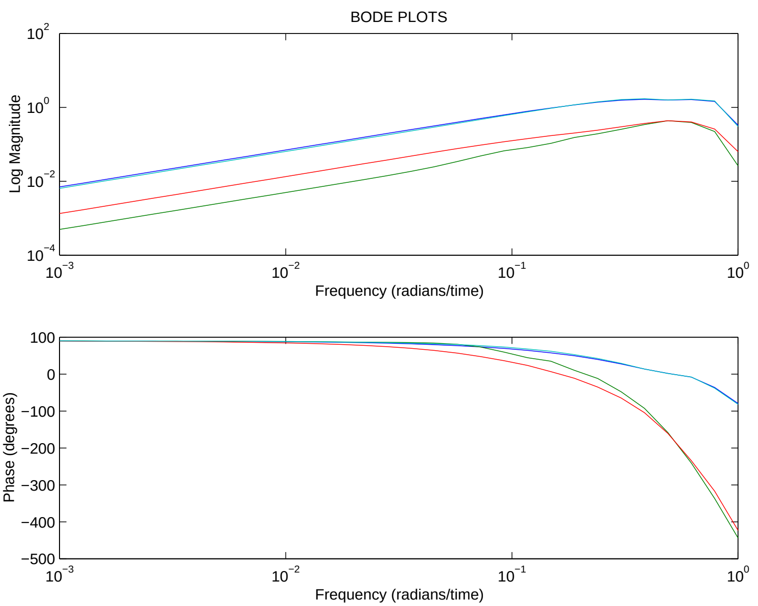
`vmat` is a *varying* matrix which contains the data to be plotted.

Let $F(\omega)$ denote the matrix (whose entries are functions of the independent variable ω) whose sampled values $F(\omega_1)$, ... , $F(\omega_N)$ are contained in `vmat`.

`plotfrsp(vmat)` will generate Bode plots of all elements of $F(\omega)$.

Optional inputs out and in are *row vectors* which specify the row and column indices respectively of a submatrix of $F(\omega)$. `plotfrsp` will then generate Bode plots of the elements of the specified submatrix of $F(\omega)$.

Example Output: (`mod2frsp` example)



See Also

`mod2frsp`, `varying format`

plotstep

Purpose

Plots multiple step responses as calculated by `mod2step`, `ss2step` or `tfd2step`.

Syntax

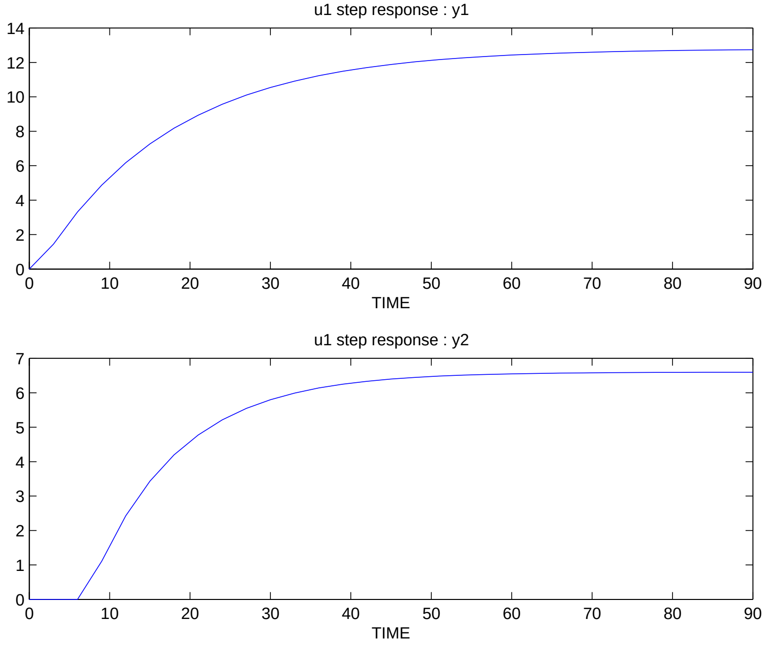
```
plotstep(plant)
plotstep(plant,opt)
```

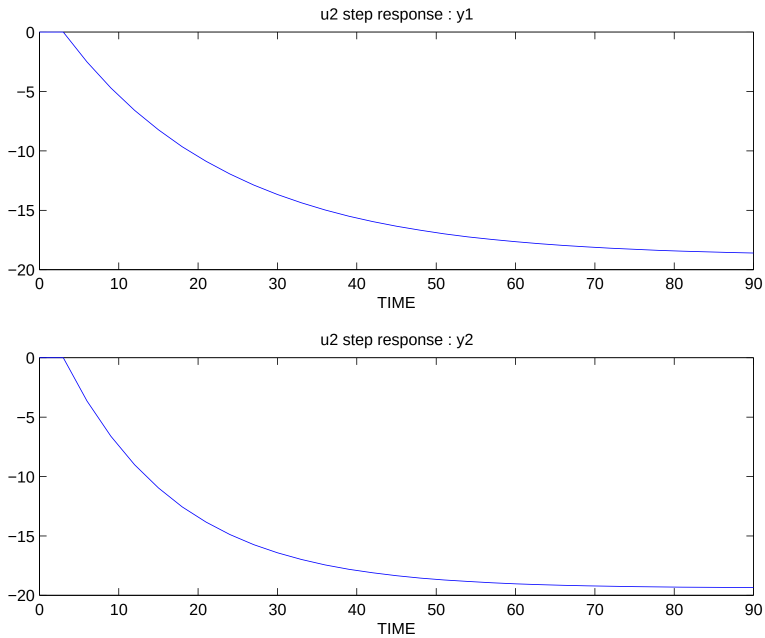
Description

`plant` is a step-response matrix in the MPC *step* format created by `mod2step`, `ss2step` or `tfd2step`.

`opt` is an optional scalar or *row vector* that allows you to select the outputs to be plotted. If you omit `opt`, `plotstep` plots every output. If, for example, `plant` contains results for 6 outputs, setting `opt=[1,4,5]` would cause only y_1, y_4 and y_5 to be plotted.

Example output: (`tfd2step` example)





See Also

`imp2step`, `mod2step`, `step format`, `plotall`, `ploteach`, `plotfrsp`, `ss2step`, `tfd2step`

Purpose

Determine the impulse response coefficients for a multi-input single-output system via Partial Least Squares (PLS).

Syntax

```
[theta,w,cw,ssqdif,yres] = plsr(xreg,yreg,ninput,lv)
[theta,w,cw,ssqdif,yres] = plsr(xreg,yreg,ninput,lv, plotopt)
```

Description

Given a set of regression data, `xreg` and `yreg`, the impulse response coefficient matrix, `theta`, is determined via PLS. Column *i* of `theta` corresponds to the impulse response coefficients for input *i*. Only a single output is allowed. The number of inputs, `ninput`, and the number of latent variables, `lv`, must be specified.

Optional output `w` is a matrix of dimension *n* (number of impulse response coefficients) by `lv` consisting of orthogonal column vectors maximizing the cross variance between input and output. Column vector `cw` (optional) contains the coefficients associated with each orthogonal vector for calculating `theta` (`theta=w*cw`).

Optional output `ssqdif` is an `lv`-by-2 matrix containing the percent variances captured by PLS. The first column contains information for the input; the second column for the output. Row *i* of `ssqdif` gives a measure of the variance captured by using the first *i* latent variables.

The output residual or prediction error (`yres`) is also returned (optional).

No plot is produced if `plotopt` is equal to 0, which is the default; a plot of the actual output and the predicted output is produced if `plotopt=1`; two plots — plot of actual and predicted output, and plot of output residual — are produced for `plotopt=2`.

Example

Consider the following two-input single-output system:

$$y(s) = \begin{bmatrix} \dfrac{5.72e^{-14s}}{60s+1} & \dfrac{1.52e^{-15s}}{25s+1} \end{bmatrix} \begin{bmatrix} u_1(s) \\ u_2(s) \end{bmatrix}$$

Load the input and output data. The input and output data were generated from the above transfer function and random zero-mean noise was added to the output. Sampling time of 7 minutes was used.

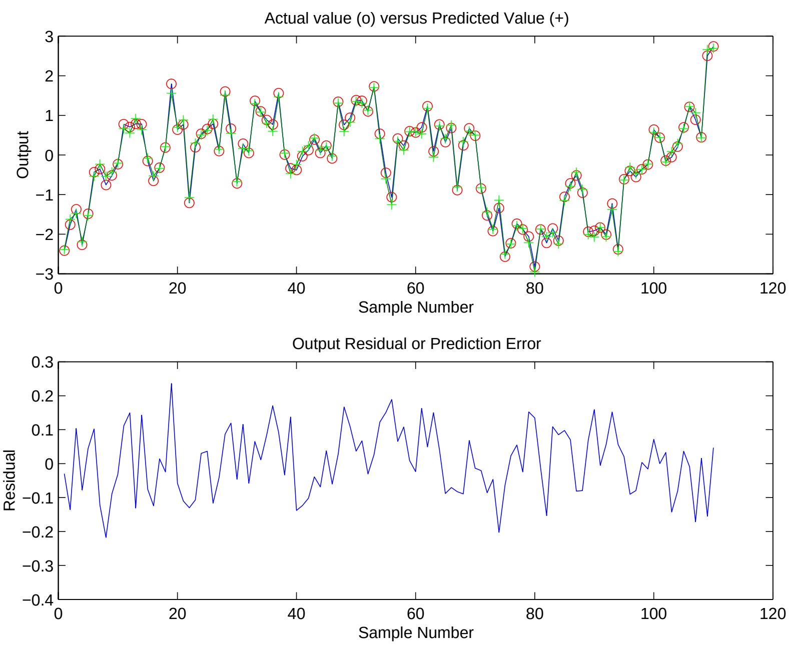
```
load plsrdat;
```

Put the input and output data in a form such that they can be used to determine the impulse response coefficients. 30 impulse response coefficients (n) are used.

```
n = 30;
[xreg,yreg] = wrtreg(x,y,n);
```

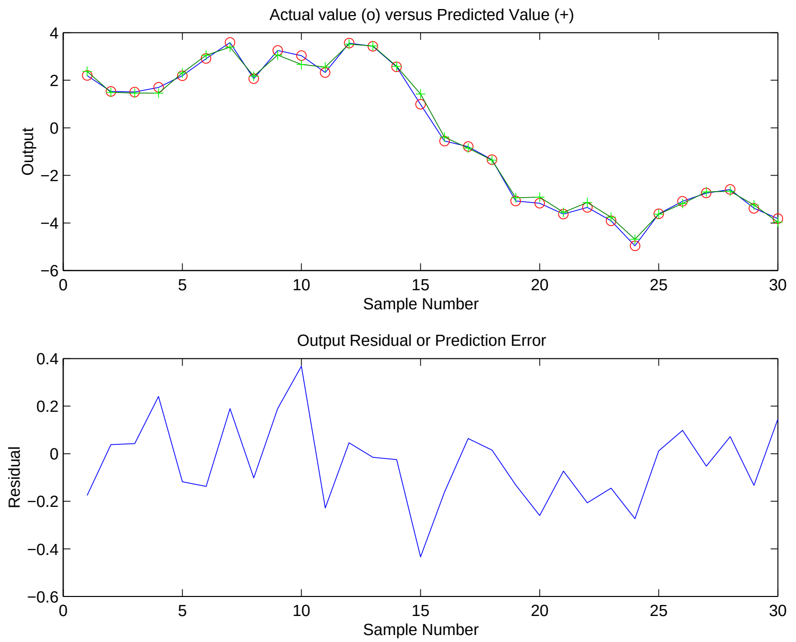
Determine the impulse response coefficients via `plsr` using 10 latent variables. By specifying `plotopt=2`, two plots — plot of predicted output and actual output, and plot of the output residual (or predicted error) — are produced.

```
ninput = 2;
lv = 10;
plotopt = 2;
theta = plsr(xreg,yreg,ninput,lv,plotopt);
```



Use a new set of data to validate the impulse model.

```
[newxreg,newyreg] = wrtreg(newx,newy,n);
yres = validmod(newxreg,newyreg,theta,plotopt);
```

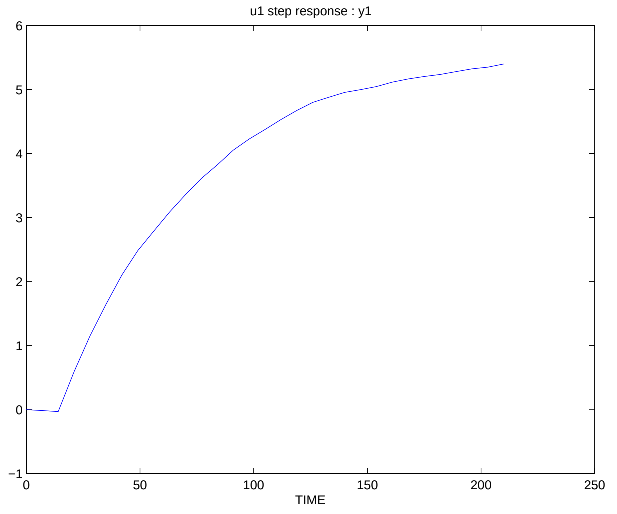


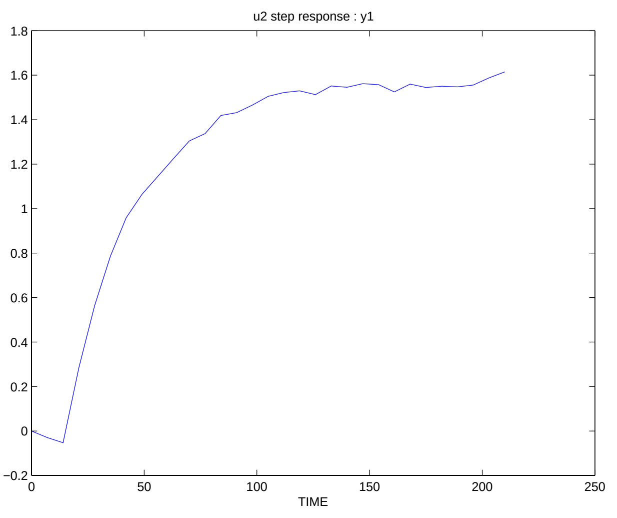
Convert the impulse model to a step model to be used in MPC design. Sampling time of 7 minutes was used in determining the impulse model. Number of outputs (1 in this case) must be specified.

```
nout = 1;
delt = 7;
model = imp2step(delt,nout,theta);
```

Plot the step response coefficients.

```
plotstep(model)
```





See Also

`mlr`, `validmod`, `wrtreg`

Purpose

`poly2tfd` converts a transfer function (continuous or discrete) from the standard MATLAB poly format into the MPC tf format.

Syntax

```
g = poly2tfd(num,den)
g = poly2tfd(num,den,delt,delay)
```

Description

Consider a continuous-time (Laplace domain) transfer function such as

$$G(s) = \frac{b_0 s^n + b_1 s^{n-1} + \ldots + b_n}{a_0 s^n + a_1 s^{n-1} + \ldots + a_n}$$

or a discrete-time transfer function such as

$$G(z) = \frac{b_0 + b_1 z^{-1} + \ldots + b_n z^{-n}}{a_0 + a_1 z^{-1} + \ldots + a_n z^{-n}}$$

where z is the forward-shift operator. Using the MATLAB *poly* format, you would represent either of these as a numerator polynomial and a denominator polynomial, giving the coefficients of the highest-order terms first:

num = [b_0 b_1 ... b_n];
den = [a_0 a_1 ... a_n];

If the numerator contains leading zeros, they may be omitted, i.e., the number of elements in `num` can be ≤ the number of elements in `den`.

`poly2tfd` uses `num` and `den` as input to build a transfer function, g, in the MPC *tf* format (see `tf` section for details). Optional variables you can include are:

`delt`

The sampling period. If this is zero or you omit it, `poly2tfd` assumes that you are supplying a continuous-time transfer function. If you are supplying a discrete-time transfer function you must specify `delt`. Otherwise g will be misinterpreted when you use it later in the MPC Toolbox functions.

`delay`

The time delay. For a continuous-time transfer function, `delay` should be in time units. For a discrete-time transfer function, `delay` should be specified as the integer number of sampling periods of time delay. If you omit it, `poly2tfd` assumes a delay of zero.

Examples

Consider the continuous-time transfer function:

$$G(s) = 0.5\frac{3s-1}{5s^2+2s+1}.$$

It has no delay. The following command creates the MPC *tf* format:

```
g=poly2tfd(0.5*[3 -1],[5 2 1]);
```

Now suppose there were a delay of 2.5 time units:

$$G(s) = 0.5\frac{3s-1}{5s^2+2s+1}e^{-2.5s}$$. You could use:

```
g=poly2tfd(0.5*[3 -1],[5 2 1],0,2.5);
```

Next let's get the equivalent transfer function in discrete form. An easy way is to get the correct poly form using `cp2dp`, then use `poly2tfd` to get it in the *tf* form. Here are the commands to do it using a sampling period of 0.75 time units:

```
delt=0.75;
[numd,dend]=cp2dp(0.5*[3 -1],[5 2 1],delt,rem(2.5,delt));
g=poly2tfd(numd,dend,delt,fix(2.5/delt));
```

Note that `cp2dp` is used to handle the fractional time delay and the integer number of sampling periods of time delay is an input to `poly2tfd`. The results are:

```
numd =
         0    0.1232   -0.1106   -0.0607

dend =
    1.0000   -1.6445    0.7408         0

g =
         0    0.1232   -0.1106   -0.0607
    1.0000   -1.6445    0.7408         0
    0.7500    3.0000         0         0
```

See Also

`cp2dp`, `tf`, `th2mod`, `tfd2step`

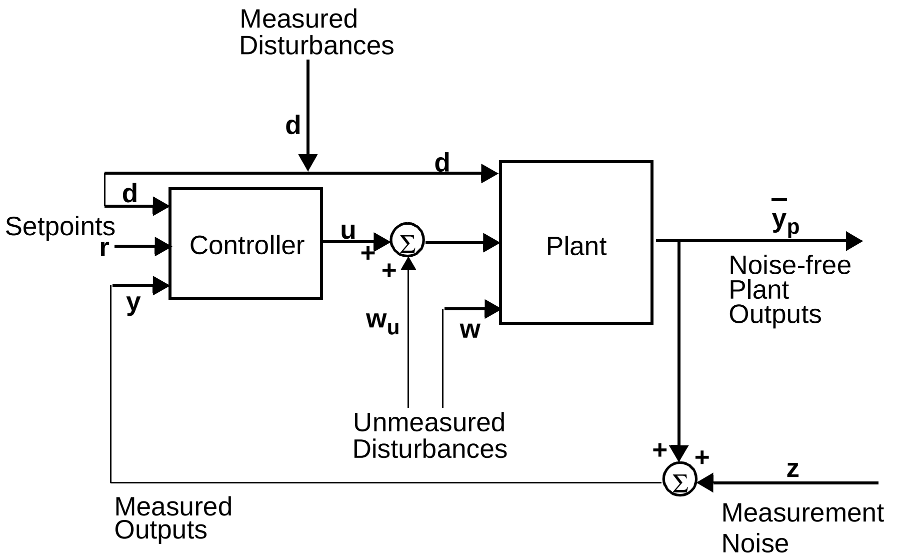
Purpose

Simulates closed-loop systems with hard bounds on manipulated variables and/or outputs using models in the MPC **mod** format. Solves the MPC optimization problem by quadratic programming.

Syntax

```
yp = scmpc(pmod,imod,ywt,uwt,M,P,tend,r)
[yp,u,ym] = scmpc(pmod,imod,ywt,uwt,M,P,tend, ...
      r,ulim,ylim,Kest,z,d,w,wu)
```

Description



scmpc simulates the performance of the type of system shown in the above diagram when there are bounds on the manipulated variables and/or outputs.

The required input variables are as follows:

pmod

Is a model in the MPC **mod** format that is to represent the plant.

imod

Is a model in the MPC **mod** format that is to be used for state estimation in the controller. In general, it can be different from pmod if you want to simulate the effect of plant/controller model mismatch.

ywt

Is a matrix of weights that will be applied to the setpoint tracking errors (optional). If `ywt=[ ]`, the default is equal (unity) weighting of all outputs over the entire prediction horizon. If `ywt¦[ ]`, it must have n_y columns, where n_y is the number of outputs. All weights must be ≥ 0.

You may vary the weights at each step in the prediction horizon by including up to `P` rows in `ywt`. Then the first row of n_y values applies to the tracking errors in the first step in the prediction horizon, the next row applies to the next step, etc. See `smpccon` for details on the form of the optimization objective function.

If you supply only *nrow* rows, where 1 ≤ *nrow* < `P`, `scmpc` will use the last row to fill in any remaining steps. Thus if you wish the weighting to be the same for all `P` steps, you need only specify a single row.

uwt

Is as for `ywt`, except that `uwt` applies to the *changes* in the manipulated variables. If you use `uwt=[ ]`, the default is zero weighting. If `uwt¦[ ]`, it must have n_u columns, where n_u is the number of manipulated variables.

M

There are two ways to specify this variable:

If it is a *scalar*, `scmpc` interprets it as the input horizon (number of moves) as in DMC.

If it is a *row vector* containing n_b elements, each element of the vector indicates the number of steps over which $\Delta u = 0$ during the optimization and `scmpc` interprets it as a set of n_b blocking factors. There may be $1 \le n_b \le$ `P` blocking factors, and their sum must be ≤ `P`.

If you set `M=[ ]`, the default is `M=P`, which is equivalent to `M=ones(1,P)`.

P

The number of sampling periods in the prediction horizon.

tend

Is the desired duration of the simulation (in time units).

r

Is a setpoint matrix consisting of N rows and n_y columns, where n_y is the number of output variables, y:

$$\mathrm{r} = \begin{bmatrix} r_1(1) & r_2(1) & \dots & r_{n_y}(1) \\ r_1(2) & r_2(2) & \dots & r_{n_y}(2) \\ \vdots & \vdots & \dots & \vdots \\ r_1(N) & r_2(N) & \dots & r_{n_y}(N) \end{bmatrix}$$

where $r_i(k)$ is the setpoint for output j at time $t = kT$, and T is the sampling period (as specified by the `minfo` vector in the **mod** format of `pmod` and `imod`). If `tend` > NT, the setpoints vary for the first N periods in the simulation, as specified by `r`, and are then held constant at the values given in the last row of `r` for the remainder of the simulation.

In many simulations one wants the setpoints to be constant for the entire time, in which case `r` need only contain a single row of n_y values.

If you set `r=[ ]`, the default is a row of n_y zeros.

The following input variables are optional. In general, setting one of them equal to an empty matrix causes `scmpc` to use the default value, which is given in the description.

ulim

Is a matrix giving the limits on the manipulated variables. Its format is as follows:

$$\text{ulim} = \left[\begin{bmatrix} u_{min,1}(1) & \dots & u_{min,n_u}(1) \\ u_{min,1}(2) & \dots & u_{min,n_u}(2) \\ \vdots & \dots & \vdots \\ u_{min,1}(N) & \dots & u_{min,n_u}(N) \end{bmatrix} \begin{bmatrix} u_{max,1}(1) & \dots & u_{max,n_u}(1) \\ u_{max,1}(2) & \dots & u_{max,n_u}(2) \\ \vdots & \dots & \vdots \\ u_{max,1}(N) & \dots & u_{max,n_u}(N) \end{bmatrix} \begin{bmatrix} \Delta u_{max,1}(1) & \dots & \Delta u_{max,n_u}(1) \\ \Delta u_{max,1}(2) & \dots & \Delta u_{max,n_u}(2) \\ \vdots & \dots & \vdots \\ \Delta u_{max,1}(N) & \dots & \Delta u_{max,n_u}(N) \end{bmatrix} \right]$$

Note that it contains three matrices of N rows. In this case, the limits on N are $1 \le N \le n_b$, where n_b is the number of times the manipulated variables are to change over the input horizon. If you supply fewer than n_b rows, the last row is repeated automatically.

The first matrix specifies the *lower bounds* on the n_u manipulated variables. For example, $u_{min,j}(2)$ is the lower bound for manipulated variable j for the second move of the manipulated variables (where the first move is at the start of the prediction horizon). If $u_{min,j}(k) = -inf$, manipulated variable j will have no lower bound for that move.

The second matrix gives the *upper bounds* on the manipulated variables. If $u_{max,j}(k) = inf$, manipulated variable j will have no upper bound for that move.

The lower and upper bounds may be either positive or negative (or zero) as long as $u_{min,j}(k) \le u_{max,j}(k)$.

The third matrix gives the limits on the rate of change of the manipulated variables. In other words, `cmpc` will force $|u_j(k) - u_j(k-1)| \le \Delta u_{max,j}(k)$. The limits on the rate of change must be nonnegative and *finite*. If you want it to be unbounded, set the bound to a large number (but not too large — a value of 10^6 should work well in most cases).

The default is $u_{min} = -inf$, $u_{max} = inf$ and $\Delta u_{max} = 10^6$

ylim

Same format as for `ulim`, but for the lower and upper bounds of the outputs. The first row applies to the first point in the prediction horizon. The default is $y_{min} = -inf$, and $y_{max} = inf$.

Kest

Is the estimator gain matrix. The default is the DMC estimator. See `smpcest` for more details.

z

Is measurement noise that will be added to the outputs (see above diagram). The format is the same as for `r`. The default is a row of n_y zeros.

d

Is a matrix of measured disturbances (see above diagram). The format is the same as for `r`, except that the number of columns is n_d rather than n_y. The default is a row of n_d zeros.s

w

Is a matrix of unmeasured disturbances (see above diagram). The format is the same as for `r`, except that the number of columns is n_w rather than n_y.The default is a row of n_w zeros.

wu

Is a matrix of unmeasured disturbances that are added to the manipulated variables (see above diagram). The format is the same as for `r`, except that the number of columns is n_u rather than n_y. The default is a row of n_u zeros.

Notes

- You may use a different number of rows in the matrices `r`, `z`, `d`, `w` and `wu`, should that be appropriate for your simulation.

- The `ulim` constraints used here are fundamentally different from the `usat` constraints used in the `smpcsim` function. The `ulim` constraints are defined relative to the beginning of the prediction horizon, which moves as the simulation progresses. Thus at each sampling period, k, the `ulim` constraints apply to a block of calculated moves that begin at sampling period k and extend for the duration of the input horizon. The `usat` constraints, on the other hand, are relative to the fixed point $t = 0$, the start of the simulation.

The calculated outputs are as follows (all but `yp` are optional):

yp

Is a matrix containing M rows and n_y columns, where $M = max(fix(tend/T) + 1, 2)$. The first row will contain the initial condition, and row $k - 1$ will give the values of the noise-free plant outputs, $\bar{y}_p$ (see above diagram), at time $t = kT$.

u

Is a matrix containing the same number of rows as `yp` and n_u columns. The time corresponding to each row is the same as for `yp`. The elements in each row are the values of the manipulated variables, u (see above diagram).

Note The u values are those coming from the controller before the addition of the unmeasured disturbance, w_u.

---

ym

Is a matrix of the same structure as `yp`, containing the values of the predicted output from the state estimator in the controller. These will, in general, differ from those in `yp` if `imod`¦`pmod` and/or there are unmeasured disturbances. *The prediction includes the effect of the most recent measurement, i.e.*, it is $\hat{y}(k|k)$.

For unconstrained problems, `scmpc` and `smpcsim` should give the same results. The latter will be faster because it uses an analytical solution of the QP problem, whereas `scmpc` solves it by iteration.

Examples

Consider the linear system:

$$\begin{bmatrix} y_1(s) \\ y_2(s) \end{bmatrix} = \begin{bmatrix} \dfrac{12.8e^{-s}}{16.7s+1} & \dfrac{-18.9e^{-3s}}{21.0s+1} \\ \dfrac{6.6e^{-7s}}{10.9s+1} & \dfrac{-19.4e^{-3s}}{14.4s+1} \end{bmatrix} \begin{bmatrix} u_1(s) \\ u_2(s) \end{bmatrix}$$

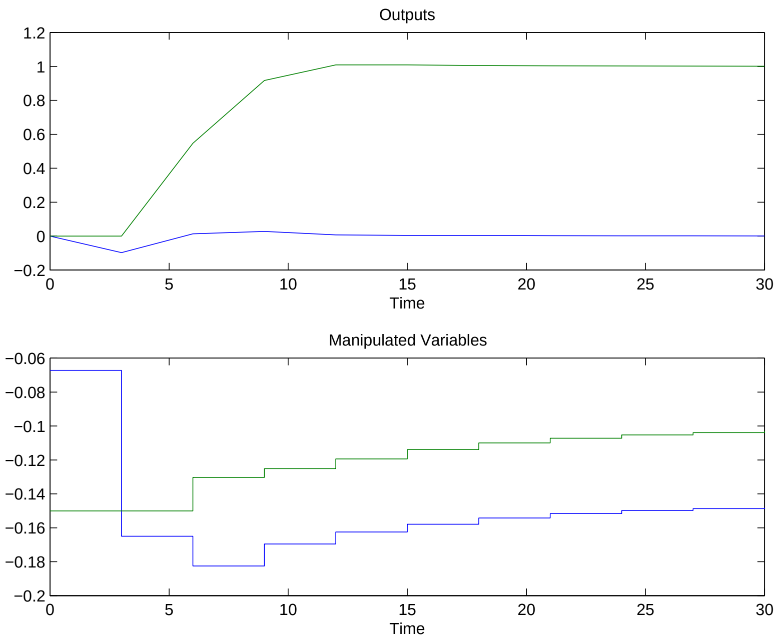
The following statements build the model and set up the controller in the same way as in the smpcsim example.

```
g11=poly2tfd(12.8,[16.7 1],0,1);
g21=poly2tfd(6.6,[10.9 1],0,7);
g12=poly2tfd(-18.9,[21.0 1],0,3);
g22=poly2tfd(-19.4,[14.4 1],0,3);
delt=3; ny=2;
imod=tfd2mod(delt,ny,g11,g21,g12,g22);
pmod=imod; P=6; M=2; ywt=[ ]; uwt=[1 1];
tend=30; r=[0 1];
```

Here, however, we will demonstrate the effect of constraints. First we set a limit of 0.1 on the rate of change of u_1 and a minimum of –0.15 for u_2.

```
ulim=[-inf -0.15 inf inf 0.1 100];
ylim=[ ];
[y,u]=scmpc(pmod,imod,ywt,uwt,M,P,tend,r,ulim,ylim);
plotall(y,u,delt), pause
```

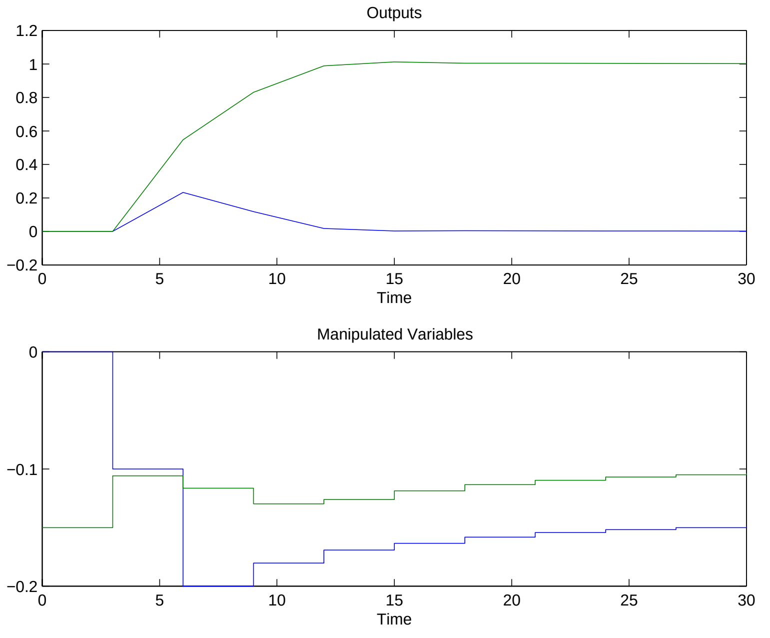
Note that Δu_2 has a large (but finite) limit. It never comes into play.



We next apply a lower bound of zero to both outputs:

```
ulim=[-inf -0.15 inf inf 0.1 100];
ylim=[0 0 inf inf];
[y,u]=scmpc(pmod,imod,ywt,uwt,M,P,tend,r,ulim,ylim);
plotall(y,u,delt),
pause
```

The following results show that no constraints are violated.



Restrictions

- Initial conditions of zero are used for all states in `imod` and `pmod`. This simulates the condition where all variables represent a deviation from a steady-state initial condition.
- The first $n_u + n_d$ columns of the D matrices in `imod` and `pmod` must be zero. In other words, neither u nor d may have an immediate effect on the outputs.

Suggestions

Problems with many inequality constraints can be very time consuming. You can minimize the number of constraints by:

- Using small values for `P` and/or `M`.
- Leaving variables unconstrained (limits at ±*inf*) intermittently unless you think the constraint is important.

See Also

`plotall`, `ploteach`, `smpccl`, `smpccon`, `smpcest`, `smpcsim`

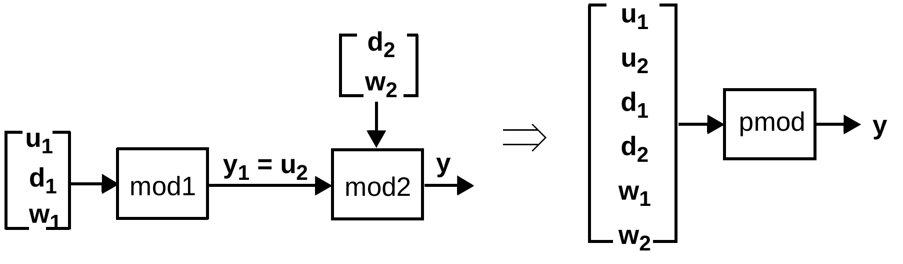
Purpose

Puts two models in series by connecting the output of one to the input of the other. Mimics the `series` function of the Control System Toolbox, except that `sermod` works on models in the MPC **mod** format.

Syntax

```
pmod = sermod(mod1,mod2)
```

Description



`mod1` and `mod2` are models in the MPC **mod** format (see `mod` in the online *MATLAB Function Reference* for a detailed description). You would normally create them using either the `tfd2mod`, `ss2mod` or `th2mod` functions.

`sermod` combines them to form a composite system, `pmod`, as shown in the above diagram. It is also in the **mod** format. Note how the inputs to `mod1` and `mod2` are ordered in `pmod`.

Restrictions

- `mod1` and `mod2` must have been created with equal sampling periods.
- The number of *measured* outputs in `mod1` must equal the number of manipulated variables in `mod2`.

See Also

`addmd`, `addmod`, `addumd`, `appmod`, `paramod`

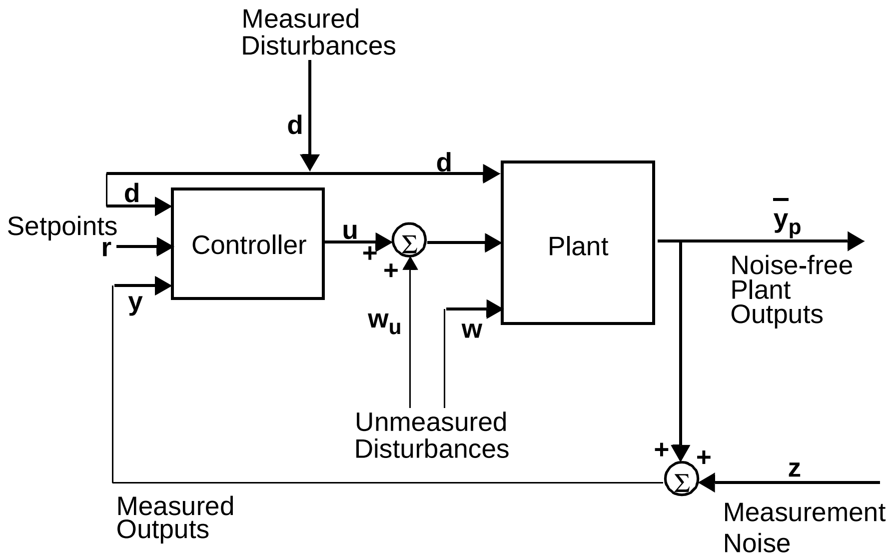
Purpose

Combines a plant model and a controller model in the MPC **mod** format, yielding a closed-loop system model in the MPC format. This can be used for stability analysis and linear simulations of closed-loop performance.

Syntax

```
[clmod,cmod] = smpccl(pmod,imod,Ks)
  [clmod,cmod] = smpccl(pmod,imod,Ks,Kest)
```

Description



pmod
Is a model (in the **mod** format) representing the `plant` in the above diagram.

imod
Is a model (in the same format) that is to be used to design the MPC controller block shown in the diagram. It may be the same as `pmod` (in which case there is no *model error* in the controller design), or it may be different.

Ks
Is a controller gain matrix, which must have been calculated by the function `smpccon`.

`Kest`

Is an (optional) estimator gain matrix. If omitted or set to an empty matrix, the default is to use the *DMC estimator* index DMC estimator. See the documentation for the function `smpcest` for more details on the design and proper format of `Kest`.

`smpccl`

Calculates a model of the closed-loop system, `clmod`. It is in the **mod** format and can be used, for example, with analysis functions such as `smpcgain` and `smpcpole`, and with simulation routines such as `mod2step` and `dlsimm`. `smpccl` also calculates a model of the controller element, `cmod`.

The closed-loop model, `clmod`, has the following state-space representation:

$$x_{cl}(k+1) = \Phi_{cl} x_{cl}(k) + \Gamma_{cl} u_{cl}(k)$$

$$y_{cl}(k) = C_{cl} x_{cl}(k) + D_{cl} u_{cl}(k)$$

where x_{cl} is a vector of n state variables, u_{cl} is a vector of input variables, y_{cl} is a vector of outputs, and Φ_{cl}, Γ_{cl}, C_{cl}, and D_{cl} are matrices of appropriate size. The expert user may want to know the significance of the state variables in x_{cl}. They are (in the following order):

- The n_p states of the plant (as specified in `pmod`),
- The n_i changes in the state estimates (based on the model specified in `imod` and the estimator gain, `Kest`),
- The n_y estimates of the *noise-free* plant output $\hat{y} = (k|k-1)$ (from the state estimator),
- n_u integrators that operate on the Δu signal produced by the standard MPC formulation to yield a u signal that can be used as input to the plant and as a closed-loop output, and
- n_d differencing elements that operate on the d signal to produce the Δd signal required in the standard MPC formulation. If there are no measured disturbances, these states are omitted.

The closed-loop input and output variables are:

$$u_{cl}(k) = \begin{bmatrix} r(k) \\ z(k) \\ w_u(k) \\ d(k) \\ w(k) \end{bmatrix} \quad \text{and} \quad y_{cl}(k) = \begin{bmatrix} \bar{y}_p(k) \\ u(k) \\ \hat{y}(k|k) \end{bmatrix}$$

where $\hat{y} = (k|k)$ is the estimate of the noise-free plant output at sampling period k based on information available at period k. This estimate is generated by the controller element.

Note that u_{cl} will include d and/or w automatically whenever `pmod` includes measured disturbances and/or unmeasured disturbances. Thus the length of the u_{cl} vector will depend on the inputs you have defined in `pmod` and `imod`. Similarly, `ycl` depends on the number of outputs and manipulated variables. Let m and p be the lengths of u_{cl} and y_{cl}, respectively. Then

$$m = 2n_y + n_u + n_d + n_w$$
$$p = 2n_y + n_u$$

The state-space form of the controller model, `cmod`, can be written as:

$$x_c(k+1) = \Phi_c x_c(k) + \Gamma_{cl} u_c(k)$$

$$y_c(k) = C_c x_c(k) + D_c u_c(k)$$

where

$$u_c(k) = \begin{bmatrix} r(k) \\ y(k) \\ d(k) \end{bmatrix} \quad \text{and} \quad y_c(k) = u(k)$$

and the controller states are the same as those of the closed loop system except that the n_p plant states are not included.

Examples

Consider the linear system:

$$\begin{bmatrix} y_1(s) \\ y_2(s) \end{bmatrix} = \begin{bmatrix} \dfrac{12.8e^{-s}}{16.7s+1} & \dfrac{-18.9e^{-3s}}{21.0s+1} \\ \dfrac{6.6e^{-7s}}{10.9s+1} & \dfrac{-19.4e^{-3s}}{14.4s+1} \end{bmatrix} \begin{bmatrix} u_1(s) \\ u_2(s) \end{bmatrix}$$

We build this model using the MPC Toolbox functions `poly2tfd` and `tfd2mod`.

```
g11=poly2tfd(12.8,[16.7 1],0,1);
g21=poly2tfd(6.6,[10.9 1],0,7);
g12=poly2tfd(-18.9,[21.0 1],0,3);
g22=poly2tfd(-19.4,[14.4 1],0,3);
delt=3; ny=2;
imod=tfd2mod(delt,ny,g11,g21,g12,g22);
pmod=imod; % No plant/model mismatch
```

Now we design the controller. Since there is delay, we use `M < P`: We specify the defaults for the other tuning parameters, `uwt` and `ywt`, then calculate the controller gain:

```
P=6; % Prediction horizon.
M=2; % Number of moves (input horizon).
ywt=[ ]; % Output weights (default - unity on
% all outputs).
uwt=[ ]; % Man. Var weights (default - zero on
% all man. vars).
Ks=smpccon(imod,ywt,uwt,M,P);
```

Now we can calculate the model of the closed-loop system and check its poles for stability:

```
clmod=smpccl(pmod,imod,Ks);
maxpole=max(abs(smpcpole(clmod)))
```

The result is:

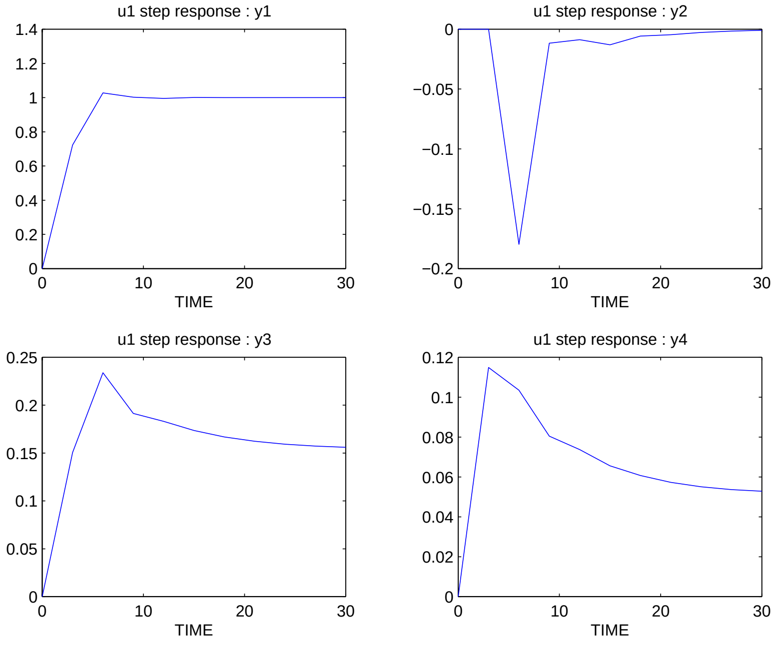
```
maxpole = 0.8869
```

Since this is less than 1, the plant and controller combination will be closed-loop stable. (The closed-loop system has 20 states in this example).

You can also use the closed-loop model to calculate and plot the step response with respect to *all* the inputs. The appropriate commands are:

```
tend=30;
clstep=mod2step(clmod,tend);
plotstep(clstep)
```

Since the closed-loop system has $m = 6$ inputs and $p = 6$ outputs, only one of the plots is reproduced here. It shows the response of the first 4 closed-loop outputs to a step in the first closed-loop input, which is the setpoint for y_1:



Closed-loop outputs y_1 and y_2 are the true plant outputs (noise-free). Output y_1 goes to the new setpoint quickly with a small overshoot. This causes a small, short-term disturbance in y_2. The plots for y_3 and y_4 show the required variation in the manipulated variables.

The following commands show how you could use `dlsimm` to calculate the response of the closed-loop system to a step in the setpoint for y_1, with added random measurement noise.

```
r=[ones(11,1) zeros(11,1)];
z=0.1*rand(11,2);
wu=zeros(11,2);
d=[ ];
w=[ ];
ucl=[r z wu d w];
[phicl,gamcl,ccl,dcl]=mod2ss(clmods);
ycl=dlsimm(phicl,gamcl,ccl,dcl,ucl);
y=ycl(:,1:2); u=ycl(:,3:4); ym=ycl(:,5:6);
```

Restrictions

- `imod` and `pmod` must have been created using the same sampling period, and an equal number of outputs, measured disturbances, and manipulated variables.
- Both `imod` and `pmod` must be strictly proper, i.e., the D matrices in their state-space descriptions must be zero. *Exception:* the last n_w columns of the D matrices may be nonzero, i.e., the unmeasured disturbance may have an immediate effect on the outputs.

See Also

`mod2step`, `scmpc`, `smpccon`, `smpcest`, `smpcgain`, `smpcpole`, `smpcsim`

smpccon

Purpose

Calculates MPC controller gain using a model in MPC **mod** format.

Syntax

```
Ks = smpccon(imod)
Ks = smpccon(imod,ywt,uwt,M,P)
```

Description

Combines the following variables (most of which are optional and have default values) to calculate the state-space MPC gain matrix, `Ks`.

`imod` is the model of the process to be used in the controller design (in the **mod** format).

The following input variables are optional:

`ywt`

Is a matrix of weights that will be applied to the setpoint tracking errors. If you use `ywt=[ ]` or omit it, the default is equal (unity) weighting of all outputs over the entire prediction horizon. If `ywt¦[ ]`, it must have n_y columns, where n_y is the number of outputs. All weights must be ≥ 0.

You may vary the weights at each step in the prediction horizon by including up to `P` rows in `ywt`. Then the first row of n_y values applies to the tracking errors in the first step in the prediction horizon, the next row applies to the next step, etc.

If you supply only *nrow* rows, where $1 \leq nrow <$ `P`, `smpccon` will use the last row to fill in any remaining steps. Thus if you wish the weighting to be the same for all `P` steps, you need only specify a single row.

`uwt`

Same format as `ywt`, except that `uwt` applies to the *changes* in the manipulated variables. If you use `uwt=[ ]` or omit it, the default is zero weighting. If `uwt¦[ ]`, it must have n_u columns, where n_u is the number of manipulated variables.

`M`

There are two ways to specify this variable:

If it is a *scalar*, `smpccon` interprets it as the input horizon (number of moves) as in DMC.

If it is a *row vector* containing n_b elements, each element of the vector indicates the number of steps over which $\Delta u = 0$ during the optimization and `smpccon` interprets it as a set of n_b blocking factors. There may be $1 \le n_b \le$ `P` blocking factors, and their sum must be $\le$ `P`.

If you set `M=[ ]` or omit it, the default is `M=P`, which is equivalent to `M=ones(1,P)`.

P

The number of sampling periods in the prediction horizon. If you set `P=[ ]` or omit it, the default is `P=1`.

If you take the default values for all the optional variables, you get the "perfect controller," i.e., a model-inverse controller. This controller is not applicable in the following situations:

- When one or more outputs cannot respond to the manipulated variables within 1 sampling period due to time delay, the plant-inverse controller is unrealizable. To counteract this you can penalize changes in the manipulated variables (variable `uwt`), use blocking (variable `M`), and/or make `P>>M`.
- When `imod` contains transmission zeros outside the unit circle the plant-inverse controller will be unstable. To counteract this, you can use blocking (variable `M`), restrict the input horizon (variable `M`), and/or penalize changes in the manipulated variables (variable `uwt`).

The model-inverse controller is also relatively sensitive to model error and is best used as a point of reference from which you can progress to a more robust design.

Algorithm

The controller gain is a component of the solution to the optimization problem:

Minimize
$$J(k) = \sum_{j=1}^{p} \sum_{i=1}^{n_y} (ywt_i(j)[r_i(k+j) - \hat{y}_i(k+j)])^2 + \sum_{j=1}^{n_b} \sum_{i=1}^{n_u} (uwt_i(j)\Delta\hat{u}_i(j))^2$$

with respect to $\Delta\hat{u}_i(j)$ (a series of current and future moves in the manipulated variables), where $\hat{y}_i$ (k + j) is a prediction of output i at a time j sampling periods into the future (relative to the current time, k), which is a function of

$\Delta\hat{u}_i(j)$, $r_i(k+j)$ is the corresponding future setpoint, and n_b is the number of blocks or moves of the manipulated variables.

References

Ricker, N. L. "Use of Quadratic Programming for Constrained Internal Model Control," *Ind. Eng. Chem. Process Des. Dev.*, 1985, 24, 925–936.

Ricker, N. L. "Model-predictive control with state estimation," *I & EC Res.*, 1990, 29, 374.

Example

Consider the linear system:

$$\begin{bmatrix} y_1(s) \\ y_2(s) \end{bmatrix} = \begin{bmatrix} \dfrac{12.8e^{-s}}{16.7s+1} & \dfrac{-18.9e^{-3s}}{21.0s+1} \\ \dfrac{6.6e^{-7s}}{10.9s+1} & \dfrac{-19.4e^{-3s}}{14.4s+1} \end{bmatrix} \begin{bmatrix} u_1(s) \\ u_2(s) \end{bmatrix}$$

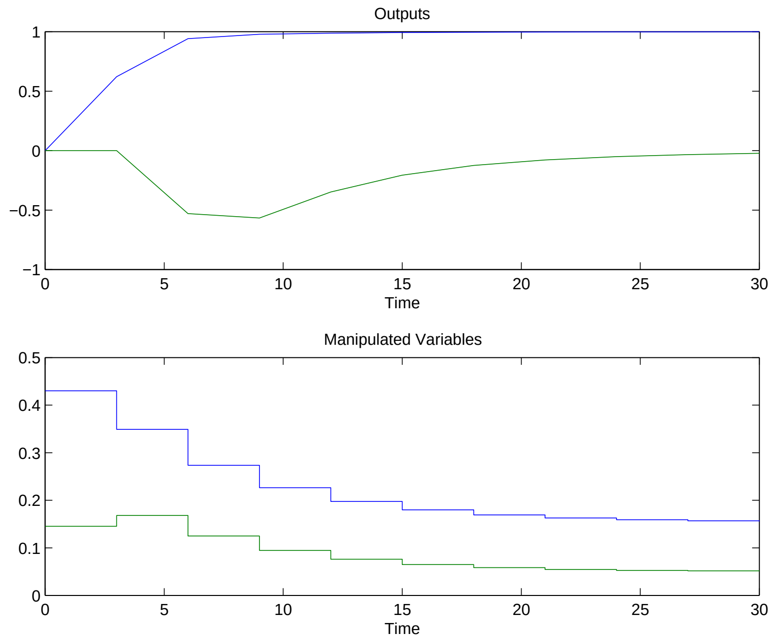
See the `smpccl` example for the commands that build the model and a simple controller for this process.

Here is a slightly more complex design with blocking and time-varying weights on the manipulated and output variables:

```
P=6; M=[2 4];
 uwt=[1 0; 0 1];
 ywt=[1 0.1; 0.8 0.1; 0.1 0.1];
 Ks=smpccon(imod,ywt,uwt,M,P);
 tend=30; r=[1 0];
 [y,u]=smpcsim(pmod,imod,Ks,tend,r);
```

There is no particular rationale for using time varying weights in this case — it is only for illustration. The manipulated variables will make 2 moves during the prediction horizon (see value of `M`, above). The `uwt` selection gives u_1 a unity weight and u_2 a zero weight for the first move, then switches the weights for the second move. If there had been any additional moves they would have had the same weighting as the second move.

The `ywt` value assigns a constant weight of 0.1 to y_2, and a weight that decreases over the first 3 periods to y_1. The weights for periods 4 to 6 are the same as for period 3. The resulting closed-loop (servo) response is:



See Also

`scmpc`, `smpccl`, `smpcsim`

Purpose

Sets up a state-estimator gain matrix for use with MPC controller design and simulation routines using models in MPC **mod** format. Can use either a disturbance/noise model that you specify, or a simplified form in which each output is affected by an independent disturbance (plus measurement noise).

Syntax

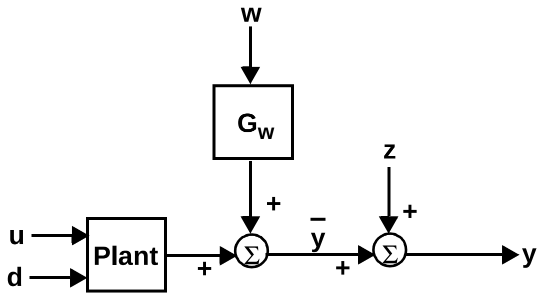
For the general case:

```
[Kest] = smpcest(imod,Q,R)
```

For simplified disturbance modeling:

```
[Kest,newmod] = smpcest(imod)
[Kest,newmod] = smpcest(imod,tau,signoise)
```

Description



In the above block diagram, u is a vector of n_u manipulated variables ($n_u \geq 1$), d is a vector of n_d measured disturbances ($n_d \geq 0$), w is a vector of unmeasured disturbances, z is measurement noise, y is a vector of outputs, and $\bar{y}$ represents these outputs before the addition of measurement noise. The objective of the state estimator in MPC is to estimate the present and future values of $\bar{y}$, rejecting as much of the measurement noise as possible. The inputs u and d are assumed perfectly measurable, whereas w and z are unknown and must be inferred from the measurements. G_w is a transfer function matrix representing the effect of each element of w on each output in y.

General Case

`imod`

Is the model (in **mod** format) to be used as the basis for the state estimator. It should be the same as that used to calculate the controller gain (see `smpccon`). *It must include a model of the disturbances*, i.e., the G_w element in the above diagram. You could, for example, use `addumd` to combine a plant and disturbance model, yielding a composite model in the proper form.

Q

Is a symmetric, positive semi-definite matrix giving the covariances of the disturbances in w. It must be n_w by n_w, where n_w (≥ 1) is the number of unmeasured disturbances in `imod` (i.e., the length of w).

R

Is a symmetric, positive-definite matrix giving the covariances of the measurement noise, z. It must be n_{ym} by n_{ym}, where n_{ym} (≥ 1) is the number of measured outputs in `imod`.

The calculated output variable is:

Kest

The estimator gain matrix. It will contain $n + n_y$ rows and n_{ym} columns, where n is the number of states in `imod`, and n_y is the total number of outputs (measured plus unmeasured).

Simplified disturbance modeling

For the *simplified disturbance/noise model* we make the following assumptions:

- The vectors w, z, y and $\bar{y}$ are all length n_y.
- G_w is diagonal. Thus each element of w affects one (and only one) element of y. Diagonal element G_{wi} has the discrete (sampled-data) form:

$$G_{wi}(q) = \frac{1}{q - a_i}$$

  where $a_i = e^{-T/\tau^i}$, $0 \leq \tau_i \leq \infty$, and T is the sampling period.

  As $\tau_i \rightarrow 0$, $Gwi(q)$ approaches a unity gain, while as $\tau_i \rightarrow \infty$, G_{wi} becomes an integrator.
- Element i of Δw is a stationary white-noise signal with zero mean and standard deviation σ_{wi} (where $w_i(k) = w_i(k) - w_i(k-1)$).
- Element i of z is a stationary white-noise signal with zero mean and standard deviation σ_{zi}.

The input variables are then as follows:

imod

Is the model (in **mod** format) to be used as the basis for the state estimator. It should be the same as that used to calculate the controller gain (see `smpccon`).

tau

Is a *row vector*, length n_y, giving the values of τ_i to be used in eq. 1. Each element must satisfy: $0 \le \tau_i \le \infty$. If you use `tau=[ ]`, `smpcest` uses the default, which is n_y zeros.

signoise

Is a *row vector*, length n_y, giving the signal-to-noise ratio for the each disturbance, defined as $\gamma_i = \sigma_{wi} = \sigma_{zi}$. Each element must be nonnegative. If omitted, `smpcsim` uses an infinite signal-to-noise ratio for each output.

The calculated output variables are:

Kest

The estimator gain matrix.

newmod

The modified version of `imod`, which must be used in place of `imod` in any simulation/analysis functions that require `Kest` (e.g., `smpccl`, `smpcsim`, `scmpc`).

If `imod` contains n states, and there are n_1 outputs for which $\tau_i > 0$, then newmod will have $n + n_1$ states. The optimal gain matrix, `Kest`, will have $n + n_1 + n_y$ rows and n_{ym} columns. The first n rows will be zero, the next n_1 rows will have the gains for the estimates of the n_1 added states (if any), and the last n_y rows will have the gains for estimating the noise-free outputs, $\bar{y}$.

Examples

Consider the linear system:

$$\begin{bmatrix} y_1(s) \\ y_2(s) \end{bmatrix} = \begin{bmatrix} \dfrac{12.8e^{-s}}{16.7s+1} & \dfrac{-18.9e^{-3s}}{21.0s+1} \\ \dfrac{6.6e^{-7s}}{10.9s+1} & \dfrac{-19.4e^{-3s}}{14.4s+1} \end{bmatrix} \begin{bmatrix} u_1(s) \\ u_2(s) \end{bmatrix} + \begin{bmatrix} \dfrac{3.8e^{-8s}}{14.9s+1} \\ \dfrac{4.9e^{-3s}}{13.2s+1} \end{bmatrix} w(s)$$

The following statements build two models: pmod, which contains the model of the disturbance, *w*, and imod, which does not.

```
g11=poly2tfd(12.8,[16.7 1],0,1);
g21=poly2tfd(6.6,[10.9 1],0,7);
g12=poly2tfd(-18.9,[21.0 1],0,3);
g22=poly2tfd(-19.4,[14.4 1],0,3);
delt=1; ny=2;
imod=tfd2mod(delt,ny,g11,g21,g12,g22);
gw1=poly2tfd(3.8,[14.9 1],0,8);
gw2=poly2tfd(4.9,[13.2 1],0,3);
pmod=addumd(imod,tfd2mod(delt,ny,gw1,gw2));
```

Calculate the gain for a typical MPC (unconstrained) controller

```
P=6; M=2;
ywt=[ ]; uwt=[1 1];
Ks=smpccon(imod,ywt,uwt,M,P);
```

Next design an estimator using the G_w model in pmod. The choices of Q and R are arbitrary. R was made relatively small (since measurement noise will be negligible in the simulations).

```
Kest1=smpcest(pmod,1,0.001*eye(ny));
Ks1=smpccon(pmod,ywt,uwt,M,P);
```

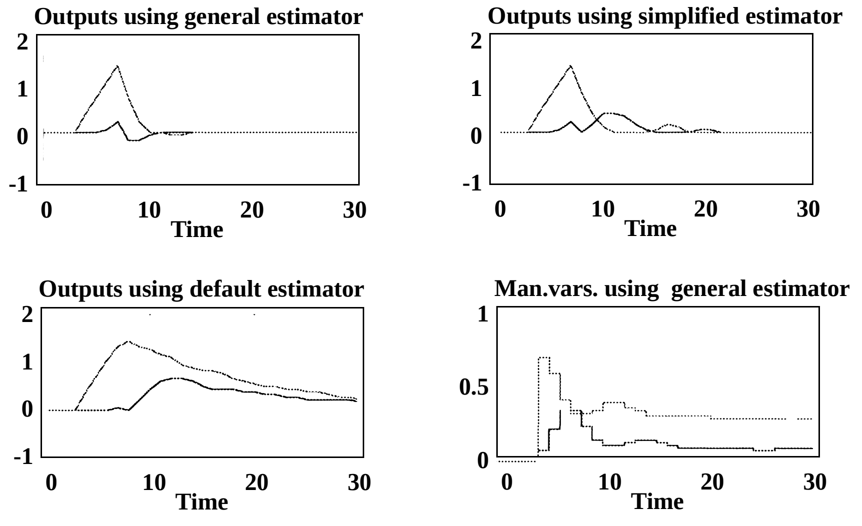
Now design another estimator using a simplified disturbance model in which each output is affected by a disturbance with a first-order time constant of 10 and a signal-to-noise ratio of 3.

```
tau=[10 10]; signoise=[3 3];
[Kest2,newmod]=smpcest(imod,tau,signoise);
Ks2=smpccon(newmod,ywt,uwt,M,P);
```

Compare the performance of these two estimators to the default (DMC) estimator when there is a unit step in *w*:

```
r=[ ]; ulim=[ ]; z=[ ]; d=[ ]; w=[1]; wu=[ ]; tend=30;
[y1,u1]=smpcsim(pmod,pmod,Ks1,tend,r,ulim,Kest1, z,d,w,wu);
[y2,u2]=smpcsim(pmod,newmod,Ks2,tend,r,ulim,Kest2, z,d,w,wu);
[y3,u3]=smpcsim(pmod,imod,Ks,tend,r,ulim,[ ],z,d,w,wu);
```

The solid lines in the following plots are for y_1 (or u_1) and the dashed lines are for y_2 (or u_2). Both outputs have setpoints at zero. You can see that the default estimator is much more sluggish than the others in counteracting this type of disturbance. The simplified disturbance design does nearly as well as that using the exact model of the disturbances. The main difference is that it allows more error in y_1 following the disturbance in y_2.



The first 14 states in both `imod` and `pmod` are for the response of the outputs to u. Since the unmeasured disturbance has no effect on them, their gains are zero. pmod contains 10 additional disturbance states and there are 2 outputs, so the last 12 rows of Kest1 are nonzero:

```
Kest1(15:26,:)=
   -0.0556    8.8659
   -0.0594    7.1499
   -0.0635    5.1314
   -0.0679    2.7748
   -0.0725    0.0411
   -0.0781   -0.0182
   -0.0915   -0.0008
   -0.0520    0.0001
    1.2663    0.0000
    0.0281   -0.0000
    0.3137    0.0000
    0.0000    0.9925
```

and the last 4 rows of Kest2 are nonzero:

```
Kest2(15:18,:) =
    0.7274         0
         0    0.7274
    0.9261         0
         0    0.9261
```

Algorithm

In the general case, smpcest uses dlqe2 to calculate the optimal estimator gain, Kest. In the simplified case, it uses an analytical solution of the discrete Riccati equation (which is possible to obtain in this case because the disturbances are independent with low-order dynamics).

The number of rows in Kest is larger than that in newmod because the MPC analysis and simulation functions augment the model states with the outputs (see mpcaugss), and Kest must be set up to account for this.

If all $\tau_i = 0$ and all $\gamma_i = \infty$, we get the DMC *estimator*, which has n rows of zeros followed by an identity matrix of dimension n_y. This is the default for all of the MPC analysis and simulation routines that require an estimator gain as input.

Important note: `smpcest` decides whether you are using the general case or the simplified approach by checking the number of output arguments you have supplied. If there is only one, it assumes you want the general case. Otherwise, it proceeds as for the simplified case. It checks the dimensions of your input arguments to make sure they are consistent with this decision.

If you get unexpected results or an error message, make sure you have specified the correct number of output arguments.

See Also

`scmpc`, `smpccl`, `smpccon`, `smpcsim`

Purpose

Calculates steady-state gain matrix or poles for a system in the MPC **mod** format.

Syntax

```
g = smpcgain(mod)
poles = smpcpole(mod)
```

Description

`mod` is a dynamic model in the MPC **mod** format. `smpcgain` and `smpcpole` convert it to its equivalent state-space form:

$$x(k+1) = \Phi x(k) + \Gamma v\ (k)$$

$$y(k) = Cx(k) + Dv(k)$$

where v includes all of the inputs in `mod`. `smpcgain` then calculates the gain matrix:

$$G = C(I - \Phi)^{1}\Gamma + D$$

which contains n_y rows, corresponding to each of the outputs in `mod`, and n_u + n_d + n_w columns, corresponding to each of the inputs.

`smpcpole` calculates the poles, i.e., the eigenvalues of the Φ matrix.

Example

See `smpccl` for an example of the use of `smpcpole`.

Restriction

If `mod` is not asymptotically stable, `smpcgain` terminates with an error message.

See Also

`mod`

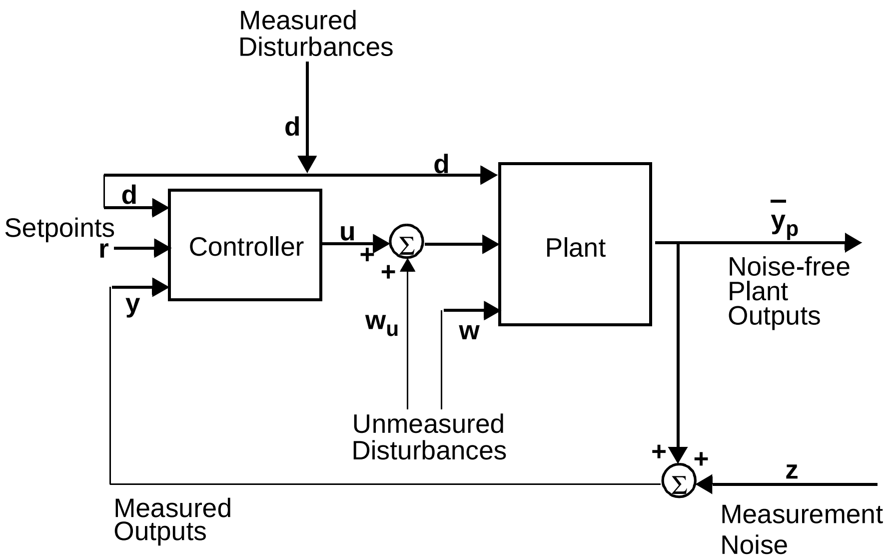
Purpose

Simulates closed-loop systems with *saturation constraints* on the manipulated variables using models in the MPC **mod** format. Can also be used for open-loop simulations.

Syntax

```
yp = smpcsim(pmod,imod,Ks,tend,r)
[yp,u,ym] = smpcsim(pmod,imod,Ks,tend,r,usat,...
    Kest,z,d,w,wu)
```

Description



`smpcsim` provides a convenient way to simulate the performance of the type of system shown in the above diagram. The required input variables are as follows:

`pmod`

Is a model in the MPC **mod** format that is to represent the plant.

`imod`

Is a model in the MPC **mod** format that is to be used for state estimation in the controller. In general, it can be different from `pmod` if you wish to simulate the effect of plant/controller model mismatch. Note, however, that `imod` should be the same as that used to calculate `Ks`.

Ks

Is the MPC controller gain matrix, usually calculated using the function `smpccon`.

If you set `Ks` to an empty matrix, `smpcsim` will do an open-loop simulation. Then the inputs to the plant will be `r` (which must be set to the vector of manipulated variables in this case), `d`, `w`, and `wu`. The measurement noise input, `z`, will be ignored.

tend

Is the desired duration of the simulation (in time units).

r

Is *normally* a setpoint matrix consisting of N rows and n_y columns, where n_y is the number of output variables, y:

$$\mathrm{r} = \begin{bmatrix} r_1(1) & r_2(1) & \dots & r_{n_y}(1) \\ r_1(2) & r_2(2) & \dots & r_{n_y}(2) \\ \vdots & \vdots & \dots & \vdots \\ r_1(N) & r_2(N) & \dots & r_{n_y}(N) \end{bmatrix}$$

where $r_i(k)$ is the setpoint for output j at time $t = kT$, and T is the sampling period (as specified by the `minfo` vector in the **mod** format of `pmod` and `imod`). If `tend` > NT, the setpoints vary for the first N periods in the simulation, as specified by `r`, and are then held constant at the values given in the last row of `r` for the remainder of the simulation.

In many simulations one wants the setpoints to be constant for the entire time, in which case `r` need only contain a single row of n_y values.

If you set `r=[ ]`, the default is a row of n_y zeros.

For *open-loop* simulations, `r` specifies the *manipulated variables* and must contain n_u columns.

The following input variables are optional. In general, setting one of them equal to an empty matrix causes `smpcsim` to use the default value, which is given in the description.

usat

Is a matrix giving the saturation limits on the manipulated variables. Its format is as follows:

$$\text{usat} = \begin{bmatrix} \begin{bmatrix} u_{min,1}(1) & \cdots & u_{min,n_u}(1) \\ u_{min,1}(2) & \cdots & u_{min,n_u}(2) \\ \vdots & \cdots & \vdots \\ u_{min,1}(N) & \cdots & u_{min,n_u}(N) \end{bmatrix} \\ \begin{bmatrix} u_{max,1}(1) & \cdots & u_{max,n_u}(1) \\ u_{max,1}(2) & \cdots & u_{max,n_u}(2) \\ \vdots & \cdots & \vdots \\ u_{max,1}(N) & \cdots & u_{max,n_u}(N) \end{bmatrix} \\ \begin{bmatrix} \Delta u_{max,1}(1) & \cdots & \Delta u_{max,n_u}(1) \\ \Delta u_{max,1}(2) & \cdots & \Delta u_{max,n_u}(2) \\ \vdots & \cdots & \vdots \\ \Delta u_{max,1}(N) & \cdots & \Delta u_{max,n_u}(N) \end{bmatrix} \end{bmatrix}$$

Note that it contains three matrices of N rows. N may be different than that for the setpoint matrix, `r`, but the idea is the same: the saturation limits will vary for the first N sampling periods of the simulation, then be held constant at the values given in the last row of `usat` for the remaining periods (if any).

The first matrix specifies the *lower bounds* on the n_u manipulated variables. For example, $u_{min,j}(k)$ is the lower bound for manipulated variable j at time $t = kT$ in the simulation. If $u_{min,j}(k) = -inf$, manipulated variable j will have no lower bound at $t = kT$.

The second matrix gives the *upper bounds* on the manipulated variables. If $u_{max,j}(k) = inf$, manipulated variable j will have no upper bound at $t = kT$.

The lower and upper bounds may be either positive or negative (or zero) as long as $u_{min,j}(k) \leq u_{max,j}(k)$.

The third matrix gives the limits on the rate of change of the manipulated variables. In other words, `smpcsim` will force $|u_j(k) - u_j(k-1)| \le \Delta u_{max,j}(k)$. The limits on the rate of change must be nonnegative.

The default is no saturation constraints, i.e., all the umin values will be set to $-inf$, and all the u_{max} and Δu_{max} values will be set to inf.

Note: Saturation constraints are enforced by simply "clipping" the manipulated variable moves so that they satisfy all constraints. This is a nonoptimal solution that, in general, will differ from the results you would get using the `ulim` variable in `scmpc`.

Kest

Is the estimator gain matrix. The default is the DMC estimator. See `smpcest` for more details.

z

Is measurement noise that will be added to the outputs (see above diagram). The format is the same as for `r`. The default is a row of n_y zeros.

d

Is a matrix of measured disturbances (see above diagram). The format is the same as for `r`, except that the number of columns is n_d rather than n_y. The default is a row of n_d zeros.

w

Is a matrix of unmeasured disturbances (see above diagram). The format is the same as for `r`, except that the number of columns is n_w rather than n_y. The default is a row of n_w zeros.I

wu

Is a matrix of unmeasured disturbances that are added to the manipulated variables (see above diagram). The format is the same as for `r`, except that the number of columns is n_u rather than n_y. The default is a row of n_u zeros.

Note: You may use a different number of rows in the matrices r, usat, z, d, w and wu, should that be appropriate for your simulation.

The calculated outputs are as follows (all but yp are optional):

yp
Is a matrix containing M rows and n_y columns, where $M = max(fix(tend=T) + 1, 2)$. The first row will contain the initial condition, and row $k - 1$ will give the values of the plant outputs, y (see above diagram), at time $t = kT$.

u
Is a matrix containing the same number of rows as yp and n_u columns. The time corresponding to each row is the same as for yp. The elements in each row are the values of the manipulated variables, u (see above diagram).

Note: The u values are those coming from the controller *before* the addition of the unmeasured disturbance, w_u.

---

ym
Is a matrix of the same structure as yp, containing the values of the predicted output from the state estimator in the controller. These will, in general, differ from those in yp if imod¦pmod and/or there are unmeasured disturbances. *The prediction includes the effect of the most recent measurement, i.e.*, it is $\hat{y}(k|k)$.

Examples

Consider the linear system:

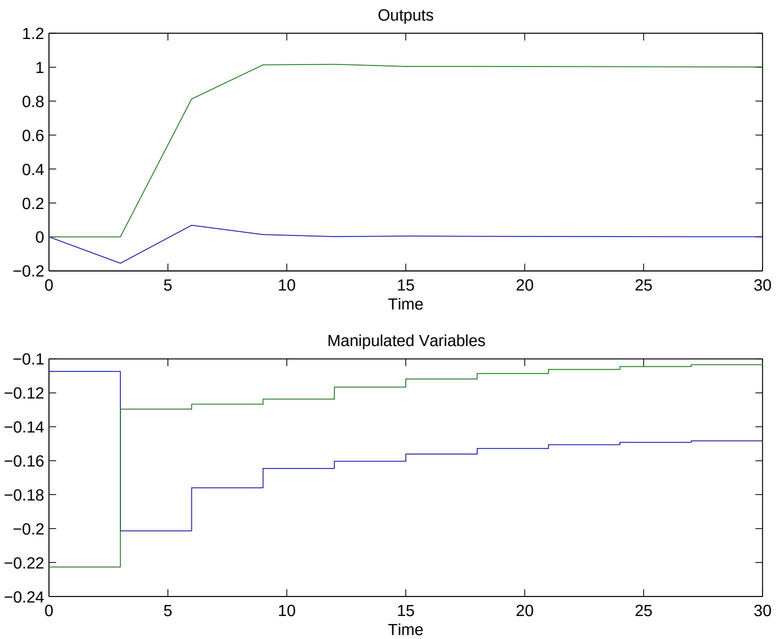
$$\begin{bmatrix} y_1(s) \\ y_2(s) \end{bmatrix} = \begin{bmatrix} \dfrac{12.8e^{-s}}{16.7s + 1} & \dfrac{-18.9e^{-3s}}{21.0s + 1} \\ \dfrac{6.6e^{-7s}}{10.9s + 1} & \dfrac{-19.4e^{-3s}}{14.4s + 1} \end{bmatrix} \begin{bmatrix} u_1(s) \\ u_2(s) \end{bmatrix}$$

The following statements build the model and calculate the MPC controller gain:

```
g11=poly2tfd(12.8,[16.7 1],0,1);
g21=poly2tfd(6.6,[10.9 1],0,7);
g12=poly2tfd(-18.9,[21.0 1],0,3);
g22=poly2tfd(-19.4,[14.4 1],0,3);
delt=3; ny=2;
imod=tfd2mod(delt,ny,g11,g21,g12,g22);
pmod=imod;
P=6; M=2;
ywt=[ ]; uwt=[1 1];
Ks=smpccon(imod,ywt,uwt,M,P);
```

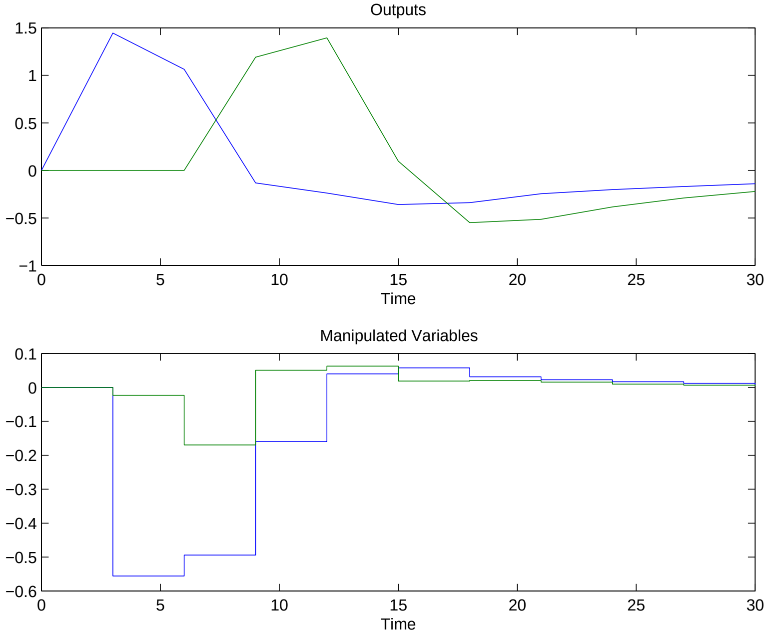
Simulate and plot the closed-loop performance for a unit step in the setpoint for y_2, occurring at $t = 0$.

```
tend=30; r=[0 1];
[y,u]=smpcsim(pmod,imod,Ks,tend,r);
plotall(y,u,delt),pause
```



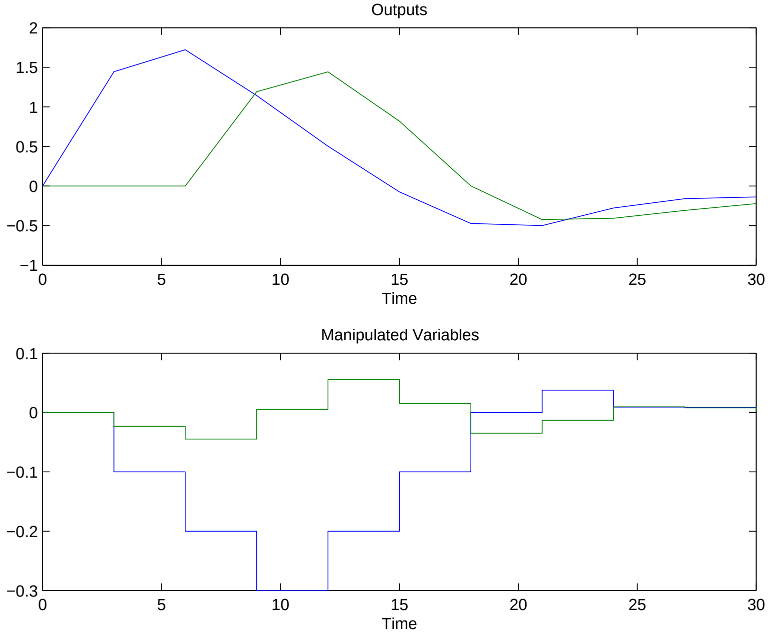
Try a pulse change in the disturbance that adds to u_1:

```
r=[ ]; usat=[ ]; Kest=[ ]; z=[ ]; d=[ ]; w=[ ];
wu=[ 1 0; 0 0];
[y,u]=smpcsim(pmod,imod,Ks,tend,r,usat,Kest,z,d,w,wu);
plotall(y,u,delt),pause
```



For the same disturbance as in the previous case, limit the rates of change of both manipulated variables.

```
usat=[-inf -inf inf inf 0.1 0.05];
[y,u]=smpcsim(pmod,imod,Ks,tend,r,usat,Kest,z,d,w,wu);
plotall(y,u,delt),pause
```



Restrictions

- Initial conditions of zero are used for all states in `imod` and `pmod`. This simulates the condition where all variables represent a deviation from a steady-state initial condition.
- The first $n_u + n_d$ columns of the D matrices in `pmod` and `imod` must be zero. In other words, neither u nor d may have an immediate effect on the outputs.

See Also

`plotall`, `ploteach`, `scmpc`, `smpccl`, `smpccon`, `smpcest`

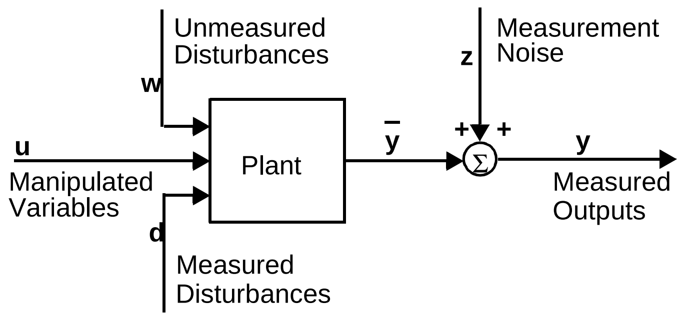
Purpose

Converts a discrete-time state-space system model into the MPC **mod** format.

Syntax

```
pmod = ss2mod(phi,gam,c,d)
pmod = ss2mod(phi,gam,c,d,minfo)
```

Description



Consider the process shown in the above block diagram. `ss2mod` assumes the following state-space representation:

$$
\begin{aligned}
x(k+1) &= \Phi x(k) + \Gamma_u u(k) + \Gamma_d d(k) + \Gamma_w w(k) \\
y(k) &= \bar{y}(k) + z(k) \\
&= Cx(k) + D_u u(k) + D_d d(k) + D_w w(k) + z(k)
\end{aligned}
$$

where x is a vector of n state variables, u represents n_u manipulated variables, d represents n_d measured disturbances, w represents n_w unmeasured disturbances, y is a vector of n_y plant outputs, z is measurement noise, and Φ, Γ_u, etc., are constant matrices of appropriate size. The variable $\bar{y}(k)$ represents the plant output before the addition of measurement noise. We further define:

$$D = [D_u \; D_d \; D_w]$$

$$\Gamma = [\Gamma_u \; \Gamma_d \; \Gamma_w]$$

`ss2mod` uses the Φ, Γ, C, and D matrices you supply to build a model, `pmod`, in the MPC **mod** format. See the `mod` section for more details.

You can also divide the outputs into n_{ym} measured outputs and n_{yu} unmeasured outputs, where $n_{ym} + n_{yu} = n_y$. Then the first n_{ym} elements in y and the first n_{ym} rows in C and D are assumed to be for the measured outputs, and the rest are for the unmeasured outputs.

`minfo` is an optional variable that allows you to specify certain characteristics of the system. The general form is a *row vector* with 7 elements, the interpretation of which is as follows:

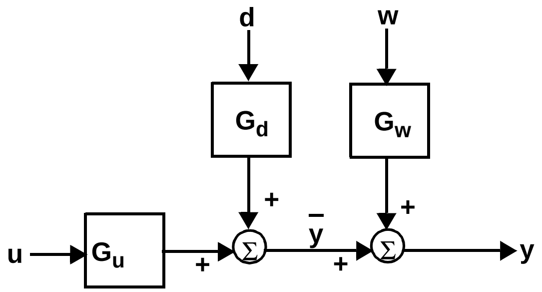
`minfo` (1)	T, the sampling period used to create the model.
(2)	n, the number of states.
(3)	n_u, the number of manipulated variable inputs.
(4)	n_d, the number of measured disturbances.
(5)	n_w, the number of unmeasured disturbances.
(6)	n_{ym}, the number of measured outputs.
(7)	n_{yu}, the number of unmeasured outputs.

If you specify `minfo` as a scalar, `ss2mod` takes it as the sampling period and sets the remaining elements of `minfo` as follows:

```
minfo(2) = # rows in phi, minfo(3) = # columns in gam,
minfo(4) = minfo(5) = 0, minfo(6) = # rows in c, minfo(7) = 0.
```

In other words, the default is to assume that all inputs are manipulated variables and all outputs are measured. If you omit `minfo`, `ss2mod` sets the sampling period to 1 and uses the defaults for the remaining elements.

Example



Suppose you have the situation shown in the above diagram where u, d, w, and y are scalar signals, and the three transfer functions are first-order:

$$G_u(z) = \frac{0.7}{1 - 0.9z^{-1}} \qquad G_d(z) = \frac{-1.5}{1 - 0.85z^{-1}}$$

$$G_w(z) = \frac{1}{1 + 0.6z^{-1}}$$

The sampling period is $T = 2$.

One way to build the model of the complete system is to convert these to state-space form and use ss2mod:

```
[phiu,gamu,cu,du]=tf2ss(0.7,[1 -0.9]);
[phid,gamd,cd,dd]=tf2ss(-1.5,[1 -0.85]);
[phiw,gamw,cw,dw]=tf2ss(1,[1 0.6]);
[phi,gam,c,d]=mpcparal(phiu,gamu,cu,du,phid,gamd,cd,dd);
[phi,gam,c,d]=mpcparal(phi,gam,c,d,phiw,gamw,cw,dw);
delt=2;
minfo=[delt 3 1 1 1 1 0];
pmod=ss2mod(phi,gam,c,d,minfo)
```

You must be careful to build up the parallel structure in the correct order. For example, the columns corresponding to Γ_u must always come first in Γ.

Another, more foolproof way is to use the addmd and addumd functions:

```
ny=1;
gu=poly2tfd(0.7,[1 -0.9],delt);
gd=poly2tfd(-1.5,[1 -0.85],delt);
gw=poly2tfd(1,[1 0.6],delt);
pmod=tfd2mod(delt,ny,gu);
pmod=addmd(pmod,tfd2mod(delt,ny,gd));
pmod=addumd(pmod,tfd2mod(delt,ny,gw))
```

Using either approach, the result is:

```
pmod =
2.0000   3.0000   1.0000   1.0000   1.0000   1.0000     0
NaN      0.9000   0        0        1.0000   0          0
0        0        0.8500   0        0        1.0000     0
0        0        0       -0.6000   0        0          1.0000
0        0.7000  -1.5000   1.0000   0        0          0
```

See Also

mod format, mod2ss

ss2step

Purpose

Uses a model in state-space format to calculate the step response of a SISO or MIMO system, in MPC *step* format.

Syntax

```
plant = ss2step(phi,gam,c,d,tfinal)
plant = ss2step(phi,gam,c,d,tfinal,delt1,delt2,nout)
```

Description

The input variables *phi*, *gam*, *c*, and *d* are assumed to be a state-space model of a process. The model can be either continuous time:

$$\dot{x}(t) = \Phi x(t) + \Gamma u(t)$$

$$y(t) = Cx(t) + Du(t)$$

or discrete time:

$$x(k+1) = \Phi x(k) + \Gamma u(k)$$

$$y(k) = Cx(k) + Du(k)$$

where x is a vector of n state variables, u is a vector of n_u inputs (usually but not necessarily manipulated variables), y is a vector of n_y plant outputs, and Φ, Γ, etc., are constant matrices of appropriate size. The `ss2step` function calculates the step responses of all the outputs of this process *with respect to all the inputs in u*, and puts this information into the variable plant in MPC *step* format. The section for `mod2step` describes the *step* format in detail.

The input variable `tfinal` is the time at which you would like to end the step response calculation, and `delt1` is the sampling period. For continuous systems, use `delt1=0`. If you do not specify `delt1`, the default is `delt1=0`.

The optional input variable `delt2` is the desired sampling period for the step response model. If you use `delt2=[ ]` or omit it, the default is `delt2=delt1` if delt1 is specified and `delt1` *neq* `0`; otherwise, the default is `delt2=1`.

The optional input variable `nout` is the output stability indicator. For stable systems, set `nout` equal to the number of outputs, n_y. For systems with one or more integrating outputs, `nout` is a column vector of length n_y with `nout(i)=0` indicating an integrating output and `nout(i)=1` indicating a stable output. If you use `nout=[ ]` or omit it, the default is `nout=`n_y (only stable outputs).

Example

The following process has 3 inputs and 4 outputs (and is the same one used for the example in the mod2step section):

```
phi=diag([0.3, 0.7, -0.7]);
gam=eye(3);
c=[1 0 0; 0 0 1; 0 1 1; 0 1 0];
d=[1 0 0; zeros(3,3)];
```

The following command duplicates the results obtained with mod2step:

```
delt1=1.5; tfinal=3*1.5;
plant=ss2step(phi,gam,c,d,tfinal,delt1)
```

See Also

plotstep, mod2step, tfd2step

svdfrsp

Purpose Calculates the singular values of a varying matrix, for example, the frequency response generated by `mod2frsp`.

Syntax

```
[sigma,omega] = svdfrsp(vmat)
```

Description `vmat` is a *varying* matrix which contains the sampled values $F(\omega_1)$, ... , $F(\omega_N)$ of a matrix function $F(\omega)$.

If the smaller dimension of $F(\omega_i)$ is m, and if $\sigma_1(\omega_i)$, . . . , $\sigma_m(\omega_i)$ are the singular values of $F(\omega_i)$, in decreasing magnitude, then the output sigma is a matrix of singular values arranged as follows:

$$\text{sigma} = \begin{bmatrix} \sigma_1(\omega_1) & \sigma_2(\omega_1) & \cdots & \sigma_m(\omega_1) \\ \sigma_1(\omega_2) & \sigma_2(\omega_2) & \cdots & \sigma_m(\omega_2) \\ \vdots & \vdots & & \vdots \\ \sigma_1(\omega_N) & \sigma_2(\omega_N) & \cdots & \sigma_m(\omega_N) \end{bmatrix}$$

The output `omega` is a column vector containing the frequencies ω_1 , . . . , ω_N.

Example See `mod2frsp, varying format` for an example of the use of this function.

See Also `mod2frsp`

Purpose

`tfd2mod` converts a transfer function (continuous or discrete) from the MPC *tf* format into the MPC **mod** format, converting to discrete time if necessary.

Syntax

model = tfd2mod(delt2,n_y,g1,g2,g3,...,g25)

Description

Consider a transfer function such as

$$G(s) = \frac{b_0 s^n + b_1 s^{n-1} + \ldots + b_n}{a_0 s^n + a_1 s^{n-1} + \ldots + a_n}$$

or

$$G(z) = \frac{b_0 + b_1 z^{-1} + \ldots + b_n z^{-n}}{a_0 + a_1 z^{-1} + \ldots + a_n z^{-n}}$$

The MPC *tf* format is a matrix consisting of three rows:

row 1	The n coefficients of the numerator polynomial, b_0 to b_n.
row 2	The n coefficients of the denominator polynomial, a_0 to a_n.
row 3	column 1: The sampling period. This must be zero if the coefficients in the above rows are for a continuous system. It must be positive otherwise. column 2: The time delay. For a continuous-time transfer function, it is in time units. For a discrete-time transfer function, it is the integer number of sampling periods of time delay.

The *tf* matrix will always have at least two columns, since that is the minimum width of the third row.

The input arguments for `tfd2mod` are:

delt2

The sampling period for the system. If any of the transfer functions `g1, ...,` `gN` are continuous-time or discrete-time with sampling period not equal to `delt2`, `tfd2mod` will convert them to discrete-time with this sampling period.

ny

The number of output variables in the plant you are modeling.

g1, g2,...gN

A sequence of N transfer functions in the *tf* format described above, where $N \geq 1$. These are assumed to be the individual elements of a transfer-function matrix:

$$\begin{bmatrix} g_{1,1} & g_{1,2} & \cdots & g_{1,n_u} \\ g_{2,1} & g_{2,2} & \cdots & g_{2,n_u} \\ \vdots & \vdots & \ddots & \vdots \\ g_{n_y,1} & g_{n_y,2} & \cdots & g_{n_y,n_u} \end{bmatrix}$$

Thus it should be clear that N must be an integer multiple (n_u) of the number of outputs, n_y.

Also, `tfd2mod` assumes that you are supplying the transfer functions in a *column-wise* order. In other words, you should first give the n_y transfer functions for input 1 ($g_{1,1}$ to g_{ny}, 1), then the n_y transfer functions for input 2 ($g_{1,2}$ to g_{ny}, 2), etc.

`tfd2mod` converts the transfer functions to discrete-time, if necessary, and combines them to form the output variable, model, which is a composite system in the MPC **mod** form.

Example

Consider the linear system:

$$\begin{bmatrix} y_1(s) \\ y_2(s) \end{bmatrix} = \begin{bmatrix} \dfrac{12.8e^{-s}}{16.7s+1} & \dfrac{-18.9e^{-3s}}{21.0s+1} \\ \dfrac{6.6e^{-7s}}{10.9s+1} & \dfrac{-19.4e^{-3s}}{14.4s+1} \end{bmatrix} \begin{bmatrix} u_1(s) \\ u_2(s) \end{bmatrix} + \begin{bmatrix} \dfrac{3.8e^{-8s}}{14.9s+1} \\ \dfrac{4.9e^{-3s}}{13.2s+1} \end{bmatrix} w(s)$$

The following commands build separate models of the response to the manipulated variables, u, and the unmeasured disturbance, w, all for a sampling period $T = 3$ then combines them using addumd to get a model of the entire system (the pmod variable):

```
g11=poly2tfd(12.8,[16.7 1],0,1);
g21=poly2tfd(6.6,[10.9 1],0,7);
g12=poly2tfd(-18.9,[21.0 1],0,3);
g22=poly2tfd(-19.4,[14.4 1],0,3);
delt=3; ny=2;
umod=tfd2mod(delt,ny,g11,g21,g12,g22);
gw1=poly2tfd(3.8,[14.9 1],0,8);
gw2=poly2tfd(4.9,[13.2 1],0,3);
wmod=tfd2mod(delt,ny,gw1,gw2);
pmod=addumd(umod,wmod);
```

Restriction

The current limit on the number of input transfer functions is $N = 25$.

See Also

`mod`, `poly2tfd`, `tfd2step`

Purpose

Calculates the MIMO step response of a model in the MPC *tf* format. The resulting step response is in the MPC *step* format.

Syntax

```
plant = tfd2step(tfinal,delt2,nout,g1)
plant = tfd2step(tfinal,delt2,nout,g1,...,g25)
```

Description

The input variables are as follows:

`tfinal`

Truncation time for step response.

`delt2`

Desired sampling period for step response.

`nout`

Output stability indicator. For stable systems, this argument is set equal to the number of outputs, n_y. For systems with one or more integrating outputs, this argument is a column vector of length n_y with `nout(i)=0` indicating an integrating output and `nout(i)=1` indicating a stable output.

`g1, g2,...gN`

A sequence of N transfer functions in the *tf* format (see `tf` format section), where N ≥ 1. These are assumed to be the individual elements of a transfer-function matrix:

$$\begin{bmatrix} g_{1,1} & g_{1,2} & \cdots & g_{1,n_u} \\ g_{2,1} & g_{2,2} & \cdots & g_{2,n_u} \\ \vdots & \vdots & \ddots & \vdots \\ g_{n_y,1} & g_{n_y,2} & \cdots & g_{n_y,n_u} \end{bmatrix}$$

Thus it should be clear that N must be an integer multiple (n_u) of the number of outputs, n_y.

`tfd2step` assumes that you are supplying the transfer functions in a *column-wise* order. In other words, you should first give the n_y transfer functions for input 1 ($g_{1,1}$ to g_{n_y}, 1), then the n_y transfer functions for input 2 ($g_{1,2}$ to g_{n_y}, 2), etc.

The output variable `plant` is the calculated step response of the n_y outputs with respect to *all* inputs. The format is as described in the *step* section.

Example

Consider the linear system:

$$\begin{bmatrix} y_1(s) \\ y_2(s) \end{bmatrix} = \begin{bmatrix} \dfrac{12.8e^{-s}}{16.7s+1} & \dfrac{-18.9e^{-3s}}{21.0s+1} \\ \dfrac{6.6e^{-7s}}{10.9s+1} & \dfrac{-19.4e^{-3s}}{14.4s+1} \end{bmatrix} \begin{bmatrix} u_1(s) \\ u_2(s) \end{bmatrix}$$

which is the same as that considered in the `mpcsim` example. We build the individual *tf* format models, then calculate and plot the MIMO step response.

```
g11=poly2tfd(12.8,[16.7 1],0,1);
g21=poly2tfd(6.6,[10.9 1],0,7);
g12=poly2tfd(-18.9,[21.0 1],0,3);
g22=poly2tfd(-19.4,[14.4 1],0,3);
delt=3; ny=2; tfinal=90;
plant=tfd2step(tfinal,delt,ny,g11,g21,g12,g22,gw1,gw2);
plotstep(plant)
```

The plots should match the example output in the `plotstep` description.

Restriction

The current limit on the number of input transfer functions is $N = 25$.

See Also

`mod2step`, `plotstep`, `ss2step`

Purpose

Converts a SISO or MISO model from the *theta* format (as used in the System Identification Toolbox) to one in the MPC **mod** format. Can also combine such models to form a MIMO system.

Syntax

```
umod = th2mod(th)
[umod,emod] = th2mod(th1,th2,...,thN)
```

Description

The System Identification Toolbox allows you to identify single-input, single-output (SISO) and multi-input, single-output (MISO) transfer functions from data. The MISO form relating an output, y, to m inputs, u_1 to u_m, and a noise input, e, is:

$$A(z)y(k) = \frac{B_1(z)}{F_1(z)}u_1(k) + \frac{B_2(z)}{F_2(z)}u_2(k) + \ldots + \frac{B_m(z)}{F_m(z)}u_m(k) + \frac{C(z)}{D(z)}e(k)$$

where A, B_i, C, D, and F_i are polynomials in the forward-shift operator, z.

The System Identification Toolbox automatically stores such models in a special format, the *theta* format. See the *System Identification Toolbox User's Guide* for details.

`th2mod` converts one or more MISO *theta* models into the MPC **mod** format, which you can then use with the MPC Toolbox functions. If you supply a single input argument, `th`, and a single output argument, `umod`, then `umod` will model the response of a single output, y, to m inputs, u_1 to u_m, where $m \geq 1$. The value of m depends on the number of inputs included in the input model, `th`. Note that `umod` will reflect the values of the A(z), B(z), and F(z) polynomials in eq. 1.

If you supply a second output argument, `emod`, it will model the response of y to the noise, e, i.e., the A(z), C(z) and D(z) polynomials in eq. 1.

If you supply p input models ($1 \leq p \leq 8$), `tfd2mod` assumes that they define a MIMO system in the following form:

$$\begin{aligned}
A_1(z)y_1(k) &= \frac{B_{11}(z)}{F_{11}(z)}u_1(k) + \ldots + \frac{B_{1m}(z)}{F_{1m}(z)}u_m(k) + \frac{C_1(z)}{D_1(z)}e_1(k) \\
&\vdots \\
A_p(z)y_p(k) &= \frac{B_{p1}(z)}{F_{p1}(z)}u_1(k) + \ldots + \frac{B_{pm}(z)}{F_{pm}(z)}u_m(k) + \frac{C_p(z)}{D_p(z)}e_p(k)
\end{aligned}$$

The p output variables have independent noise inputs. In this case, each of the p input models must include the same number of inputs, m. The p outputs are arranged in parallel in the resulting `umod` output model (and the `emod` model, if used).

If the `th` models are auto-regressive (i.e., $m = 0$), then `umod` will be set to an empty matrix and only `emod` will be nonempty.

Example

The following commands create three SISO *theta* models using the `mktheta` command (System Identification Toolbox), then converts them to the equivalent **mod** form:

```
th1=mktheta([1 0 -.2],[0 0 -1]);
th2=mktheta([1 -.8 .1],[0 -.5 .3],1,1,1);
th3=mktheta([1 -.2],[0 1],[1 2 0],[1 -1.2 .3], 1);
[umod,emod]=th2mod(th1,th2,th3)
```

The results are:

```
umod =
1.0000    5.0000    1.0000         0         0    3.0000         0
   NaN         0    0.2000         0         0         0    1.0000
     0    1.0000         0         0         0         0         0
     0         0         0    0.8000   -0.1000         0    1.0000
     0         0         0    1.0000         0         0         0
     0         0         0         0         0    2.0000    1.0000
     0         0   -1.0000         0         0         0         0
     0         0         0   -0.5000    0.3000         0         0
     0         0         0         0         0    1.0000

emod =
 Columns 1 through 7
1.0000    7.0000    3.0000         0         0    3.0000         0
   NaN         0    0.2000         0         0         0         0
     0    1.0000         0         0         0         0         0
     0         0         0    0.8000   -0.1000         0         0
     0         0         0    1.0000         0         0         0
     0         0         0         0         0    1.4000   -0.5400
     0         0         0         0         0    1.0000         0
     0         0         0         0         0         0    1.0000
     0         0    0.2000         0         0         0         0
     0         0         0    0.8000   -0.1000         0         0
     0         0         0         0         0    3.4000   -0.5400

  Columns 8 through 11

     0         0         0         0
     0    1.0000         0         0
     0         0         0         0
     0         0    1.0000         0
     0         0         0         0
0.0600         0         0    1.0000
     0         0         0         0
     0         0         0         0
     0    1.0000         0         0
     0         0    1.0000         0
    0.0600         0         0    1.0000
```

Restriction The System Identification Toolbox must be installed to use this function.

See Also `mod`

validmod

Purpose

Validates an impulse response model for a new set of data.

Syntax

```
yres = validmod(xreg,yreg,theta)
yres = validmod(xreg,yreg,theta,plotopt)
```

Description

Model validation is a very important part of building a model. For a new set of data, `xreg` and `yreg`, the impulse response model is tested by calculating the output residual, `yres`. `theta` consists of impulse response coefficients as determined by routines such as `plsr` or `mlr`.

Optional input, `plotopt`, can be supplied to produce various plots. No plot is produced if `plotopt` is equal to 0 which is the default; a plot of the actual output and the predicted output is produced if `plotopt=1`; two plots — plot of actual and predicted output, and plot of output residual — are produced for `plotopt=2`.

Example

See `plsr` for an example of the use of this function.

See Also

`mlr`, `plsr`

Purpose

Writes input and output data matrices for a multi-input single-output system such that they can be used in regression routines `mlr` and `pls` for determining impulse response coefficients.

Syntax

```
[xreg,yreg] = wrtreg(x,y,n)
```

Description

`x` is the input data of dimension N by n_u where N is number of data points and n_u is number of inputs. `y` is the output of dimension N by 1. `n` is number of impulse response coefficients for all inputs. `x` is rearranged to produce `xreg` of dimension $(N - n - 1)$ by $n * n_u$ while `yreg` is produced by deleting the first n rows of y. This operation is illustrated as follows:

$$\mathrm{x} = \begin{bmatrix} x_1(1) & \ldots & x_{n_u}(1) \\ x_1(2) & \ldots & x_{n_u}(2) \\ \vdots & & \vdots \\ x_1(N) & \ldots & x_{n_u}(N) \end{bmatrix}$$

$$\mathrm{y} = \begin{bmatrix} y(1) \\ \vdots \\ y(N) \end{bmatrix}$$

then

$$\mathrm{xreg} = \begin{bmatrix} x_1(n) & \ldots & x_1(1) & \ldots & x_{n_u}(n) & \ldots & x_{n_u}(1) \\ x_1(n+1) & \ldots & x_1(2) & \ldots & x_{n_u}(n+1) & \ldots & x_{n_u}(2) \\ \vdots & & \vdots & & \vdots & & \vdots \\ x_1(N-1) & \ldots & x_1(N-n) & \ldots & x_{n_u}N-(1) & \ldots & x_{n_u}(N-n) \end{bmatrix}$$

$$\mathrm{yreg} = \begin{bmatrix} y(n+1) \\ \vdots \\ y(N) \end{bmatrix}$$

A single sampling delay is assumed for all inputs. `y` must be a column vector, i.e., only one output can be specified.

Example See `mlr` and `plsr` for examples of the use of this function.

See Also `mlr`, `plsr`

Index