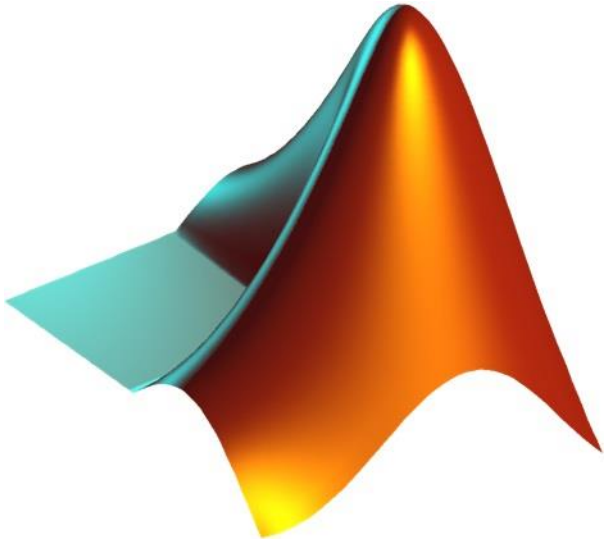
MATLAB / Simulink Lab Course
Control System Toolbox



UPDATED TO
R2019A

Objectives & Preparation “Control System Toolbox"

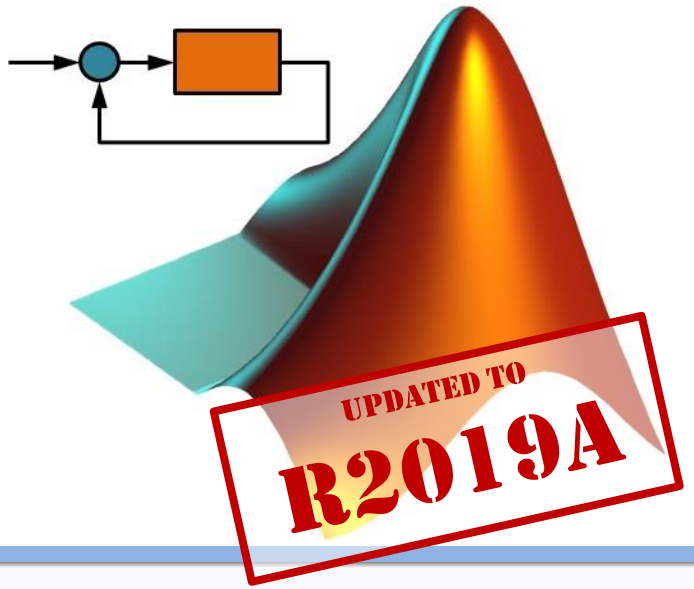
- Which MathWorks products are covered?
  - ⇨ Control System Toolbox

- What skills are learnt?
  - ⇨ Linear system representation
  - ⇨ Model interconnection and transformation
  - ⇨ Stability analysis, time domain analysis and frequency domain analysis
  - ⇨ Control design and controller tuning

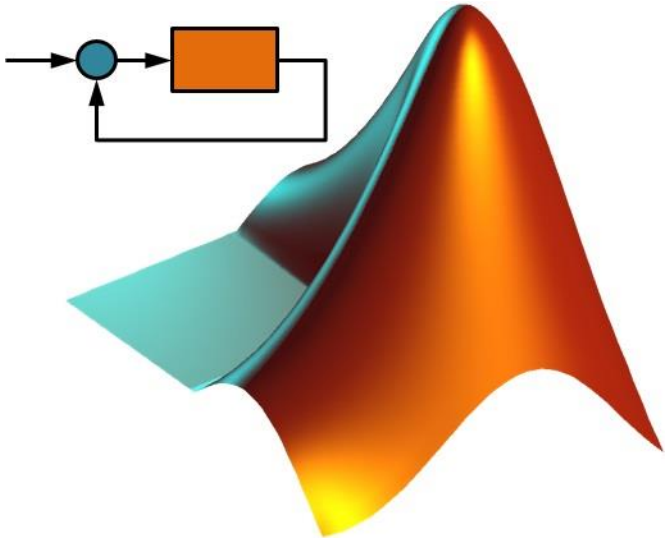
- How to prepare for the session?
  - ⇨ MathWorks Tutorials:
    - https://de.mathworks.com/help/control/getting-started-with-control-system-toolbox.html?s_tid=CRUX_lftnav
    - https://de.mathworks.com/help/control/getstart/linear-lti-models.html
    - https://de.mathworks.com/help/control/ug/control-system-modeling-with-model-objects.html

Outline

1. Introduction
2. Linear System Representation
   - Basic Models, Tunable Models and PID Models
   - Model Attributes
3. Model Interconnection
4. Model Transformation
   - Type and Continuous-Discrete Conversion
   - Simplification, State-Coordinate Transform and Modal Decomposition
5. Linear Analysis
   - Stability Analysis
   - Time Domain and Frequency Domain Analysis
6. Control Design
   - PID Controller Tuning
   - SISO Feedback Loops
   - Pole Placement, LQR, LQG and Kalman Estimator
7. Matrix Computations
8. List of Commands



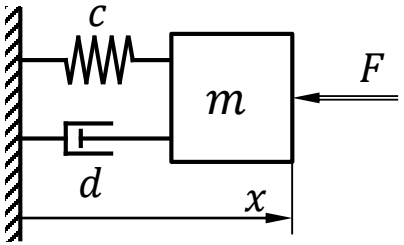
1. Introduction



Introduction

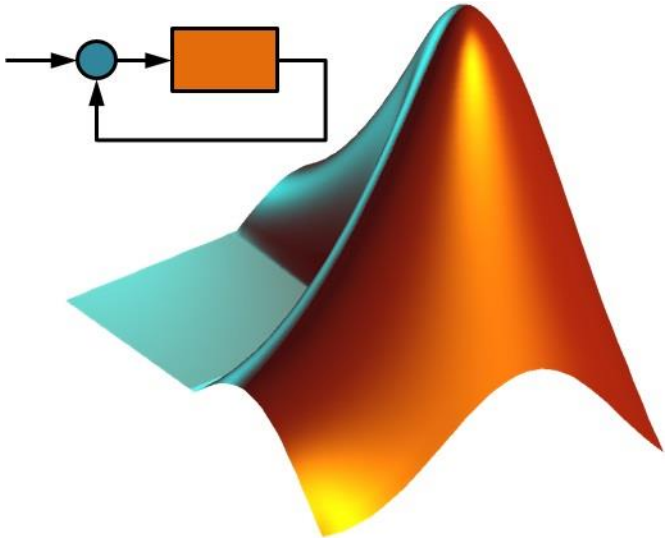
- MATLAB's **Control System Toolbox** is useful for
  - modeling,
  - analysis and
  - design of **linear** dynamic systems

- Dynamic systems change over time, like the mass-spring-damper system below.



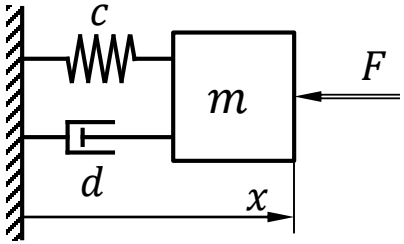
- The toolbox provides key functionalities for the design of **control systems**. Control engineering aims at modifying dynamic behavior using feedbacks of system outputs.

2. Linear System Representation



Linear System Representation

- Consider the following linear spring-mass-damper system:



- Its motion can be described by the following equation:

$$m\ddot{x} + d\dot{x} + cx = F$$

- Using the Laplace transform, the transfer function from input F to output x can be easily obtained:

$$\frac{x}{F} = \frac{1}{m \cdot s^2 + d \cdot s + c}$$

- There are various other possible representations or **models** of this linear system. The following slide shows those models provided by the Control System Toolbox.

Linear System Representation – Basic Models

Model	Example	Characterized by	MATLAB Function
Transfer Function	$\frac{n_2 s^2 + n_1 s + n_0}{d_3 s^3 + d_2 s^2 + d_1 s + d_0}$	Numerator and denominator coefficients n_i and d_i	`tf`
Zero-Pole-Gain Model	$\frac{k \cdot (s - z_1) \cdot (s - z_2)}{(s - p_{12})^2 \cdot (s - p_3)}$	Zeros z_i, poles p_i, gain k	`zpk`
State-Space Model (Explicit)	$\dot{x} = Ax + Bu$ $y = Cx + Du$	System matrices A, B, C, D	`ss`
Descriptor State-Space Model (Implicit)	$E\dot{x} = Ax + Bu$ $y = Cx + Du$	System matrices A, B, C, D, E	`dss`
Discrete Transfer Function (Z-Transform)	$\frac{n_0 + n_1 z^{-1}}{d_0 + d_1 z^{-1} + d_2 z^{-2}}$	Numerator and denominator coefficients n_i and d_i	`filt`
Frequency Response	$f_1 \rightarrow c_1$ $f_2 \rightarrow c_2$ $f_3 \rightarrow c_3$	Gain and phase (complex number c_i) over frequency f_i	`frd`

Linear System Representation – Basic Models

- Assuming the following numeric values for the spring-mass-damper system above

$$m = 50, \qquad d = 35, \qquad c = 10$$

the following models of the system can be defined in MATLAB:

```
%         tf(numerator_coefficients, denominator_coefficients)
sys_tf  = tf(1,                      [50, 35, 10]);

%         zpk(zeros, poles,                          gain)
sys_zpk = zpk([],    [0.35 + 0.278i, 0.35 - 0.278i], 0.1);

%         ss(A,                 B,         C,      D)
sys_ss  = ss([0, 1; -0.2, -0.7], [0; 0.02], [1, 0], 0);
```

$$= \frac{1}{50s^2 + 35s + 10}$$

Note: this gain does not always equal the steady-state gain!

- Note that in this case only the transfer function can be defined straightforward, without further calculations. Should you need a different type of model, you can first define a transfer function and then transform it. (See chapter “Model Transformation”)

Linear System Representation – Basic Models

- A Descriptor State-Space Model can be created much like a State-Space Model

```
sys_dss = ss(A, B, C, D, E);
```

- To create a digital filter $H(z) = \frac{2+z^{-1}}{1+0.4z^{-1}+2z^{-2}}$ use the `filt` command

```
digital_filter = filt([2, 1], [1, 0.4, 2]);
```

- The filt command lets you specify a discrete-time transfer function in what MATLAB calls the DSP format. DSP stands for digital signal processing. In this case, it means that the transfer function is described by coefficients to z^{-n}, instead of z^{n}, with $n \in \mathbb{N}_0$.

Remark on Z-Transform and digital filtering:

$$\frac{y}{u} = H(z)$$

$$(1 + 0.4z^{-1} + 2z^{-2})y = (2 + z^{-1})u$$

$$y[t] + 0.4y[t - T_s] + 2y[t - 2T_s] = 2u[t] + u[t - T_s]$$

Linear System Representation – Discrete-Time Systems

- All models presented above, with the exception of the digital filter, are time-continuous by default. To create discrete-time models, the sample time needs to be provided as an additional argument:

```
discrete_sys_tf  = tf(num, den, Ts);
discrete_sys_zpk = zpk(zeros, poles, gain, Ts);
discrete_sys_ss  = ss(A, B, C, D, Ts);
discrete_sys_dss = ss(A, B, C, D, E, Ts);
discrete_sys_frd = frd(response, frequency, Ts);
```

- To create a discrete-time model without specifying the sample time, set `Ts = -1`
- The `filt` function always returns a discrete-time transfer function. The sample time argument is optional. If it is not provided, the sample time remains unspecified.

```
digital_filter = filt([2, 1], [1, 0.4, 2], 0.3);
```

Note:

`filt([2, 1], [1, 0.4, 2], 0.3)` $\leftrightarrow \frac{2+z^{-1}}{1+0.4z^{-1}+2z^{-2}}$

`tf([2, 1], [1, 0.4, 2], 0.3)` $\leftrightarrow \frac{2z+1}{z^2+0.4z+2}$

Linear System Representation – Systems with Time Delay

- Time delays can be added to all model types by specifying the properties `InputDelay` and/or `OutputDelay`:

```
sys_tf  = tf(num, den, 'InputDelay', delay);
sys_zpk = zpk(zeros, poles, gain, 'OutputDelay', delay);
sys_frd = filt(num, den, Ts, 'InputDelay', delay);
```

Note: in SISO systems, input and output delays are usually equivalent.

- Input and output delays need to be distinguished in the case of State-Space Models, especially if the states have physical meaning!

```
sys_ss  = ss(A, B, C, D, 'InputDelay', delay_in, 'OutputDelay', delay_out);
```

- Moreover, delays can be specified for each channel of a MIMO system separately.

```
sys_ss  = ss(-2, 3, [1; -1], 0, 'InputDelay', 1.5, 'OutputDelay', [0.7; 0]);
```

- In discrete-time models, time delay is specified in numbers of sampling periods. In the following case, a 2.5 second time delay is introduced:

```
sys_discrete = tf(2, [1, -0.95], 0.1, 'InputDelay', 25);
```

Linear System Representation – Time Delay Approximation

- In some cases, actual time delays may need to be approximated by linear functions. MATLAB can generate the common Padé approximation, here of second order for $\tau = 0.3s$

```
[num,den] = pade(0.3, 2)
```

```
num =
     1.0000  -20.0000  133.3333
den =
     1.0000   20.0000  133.3333
```

Reminder – Padé approximation:

$$e^{-\tau s} \approx \frac{1 - k_1 s + k_2 s^2 + \cdots \pm k_n s^n}{1 + k_1 s + k_2 s^2 + \cdots + k_n s^n}$$

- To replace all time delays in an existing system `sys` by a 5th order Padé approximation:

```
sys_pade = pade(sys, 5)
```

- In the case of discrete-time models, use `absorbDelay` to replace a delay of k sampling periods by k poles at `z=0`:

```
sys_absorbed = absorbDelay(sys_discrete)
```

Linear System Representation – Tunable Models

- Tunable models can be useful to represent systems that have both fixed and tunable (or parametric) coefficients. If, for instance, your control system comprises a tunable low-pass filter, you can use MATLAB tunable models to conduct parameter studies.
- To create a tunable low-pass filter, a tunable real parameter has to be created first.

```
a = realp('a',10); % 10 is the initial value
F = tf(a,[1, a]);
```

- F will not be a TF object, but a GENSS object. To convert existing models to GENSS:

```
sys_gen = genss(sys);
```

- Such generalized models have various properties, one of which allows you to access all tunable parameters:

```
F.Blocks
```

```
F_blocks =
     a: [1x1 realp]
```

Linear System Representation – Tunable Models

- Similarly, a generalized FRD model can be created.

```
frd_gen = genfrd(sys, frequencies, frequency_units);
```

- All tunable parameters can be changed using the following function:

```
F_new = replaceBlock(F,'a',5);
```

- It is even possible to hand over multiple parameter values. The result is an array of state-space models:

```
F_new = replaceBlock(F,'a',[2,4,7,12]);
```

- To access the model where a=7

```
F_new(:,:,1,3)
```

  or where a=12

```
F_new(:,:,1,4)
```

Linear System Representation – PID Controllers

- PID controllers are standard in linear control. Therefore, the Control System Toolbox provides dedicated models.

```
C = pid(Kp);              % creates a proportional controller
C = pid(Kp, Ki);          % creates a PI controller
C = pid(Kp, Ki, Kd, Tf); % creates a PID controller
```

Reminder – PID controller:

$$C(s) = K_p + \frac{K_i}{s} + \frac{K_d s}{T_f s + 1}$$

PID controller in standard form:

$$C(s) = K_p \left(1 + \frac{1}{T_i s} + \frac{T_d s^2}{\frac{T_d}{N} s + 1} \right)$$

- To create a PID controller in standard form:

```
C = pidstd(Kp, Ti, Td, N);
```

- PID controllers can also be discrete-time:

```
C = pid(Kp, 0, 0, 0, Ts);      % creates a discrete-time proportional controller
C = pidstd(Kp, Ti, Td, N, Ts); % creates a discrete-time PID controller (standard form)
```

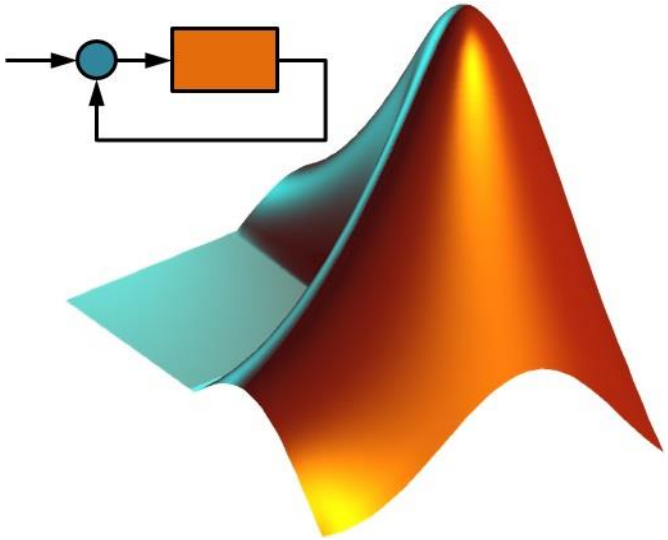
Linear System Representation – Model Attributes

- The following functions determine various model attributes:

```
isct(sys)     % determine if the model is in continuous time
isdt(sys)     % determine if the model is in discrete time
isstable(sys) % determine whether the model is stable
isproper(sys) % determine whether the model is proper
issiso(sys)   % determine if the model is single-input/single-output (SISO)
hasdelay(sys) % determine whether the model contains any delays
order(sys)    % query model order
```

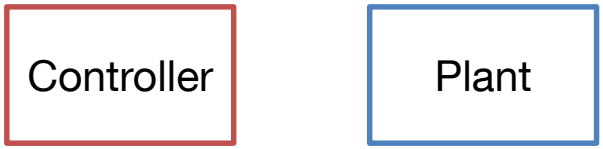
- All these functions return Boolean values, with the exception of `order`.

3. Model Interconnection

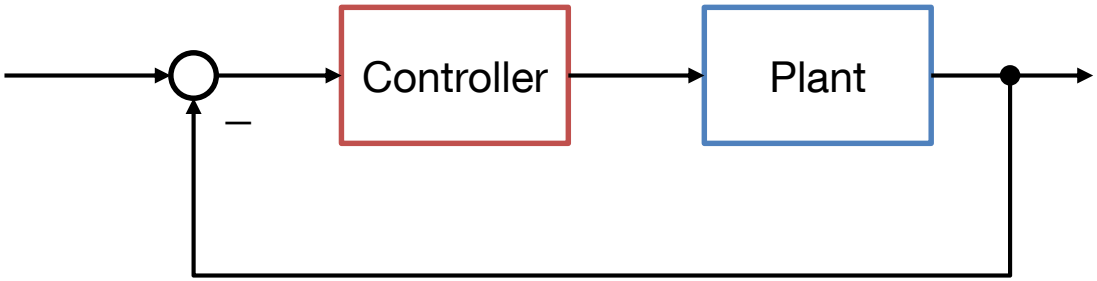


Model Interconnection

- Suppose you have created the model of a plant and the model of a controller.



- Now you want to combine those two models to form a closed control loop.



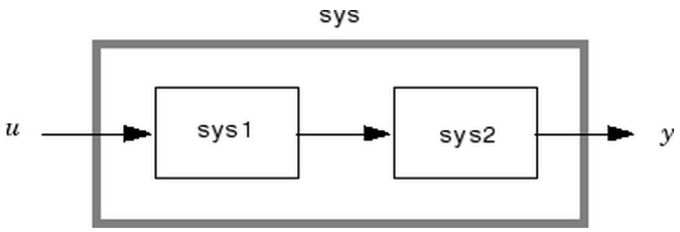
- This is where you need the tools for model interconnection.

Model Interconnection

- Series connection of two models:

```
sys = series(sys1, sys2);
```

- Both systems must be either both continuous or both discrete with identical sample time.

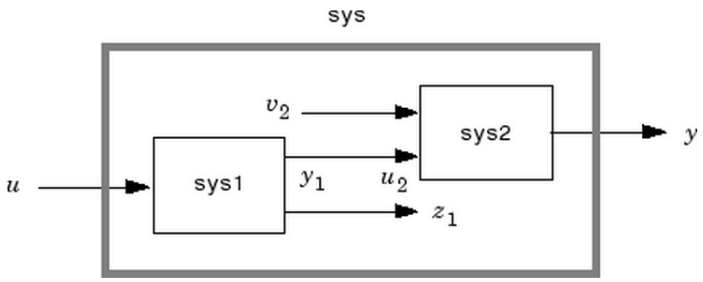


- This command is equivalent to:

```
sys = sys2 * sys1;
```

- In the case of MIMO systems, you can specify which inputs and outputs to connect.

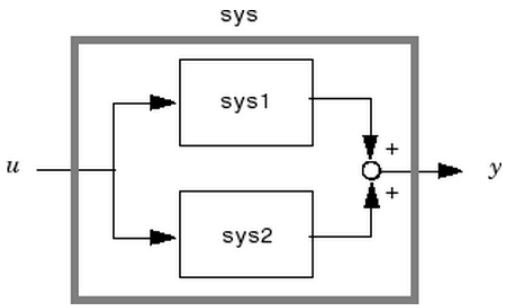
```
sys = series(sys1, sys2, outputs1, inputs2);
```



Model Interconnection

- Parallel connection of two models:

```
sys = parallel(sys1, sys2);
```

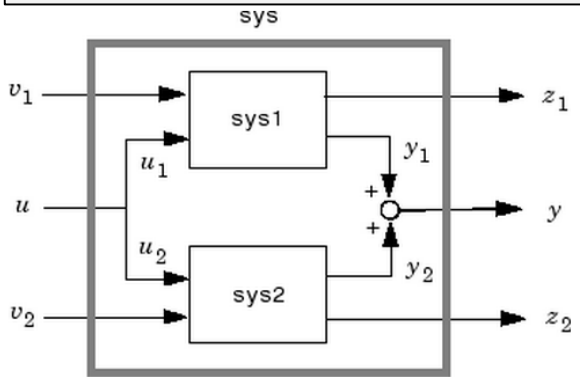


- Both systems must be either both continuous or both discrete with identical sample time.
- This command is equivalent to:

```
sys = sys1 + sys2;
```

- In the case of MIMO systems, you can specify which inputs and outputs to connect.

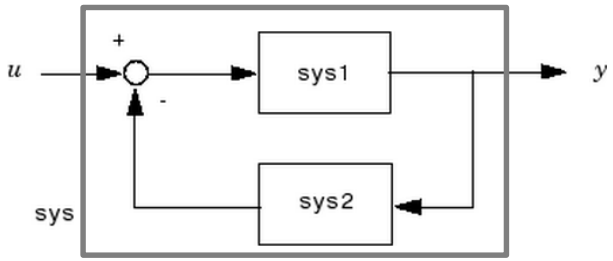
```
sys = parallel(sys1, sys2, input1, input2, output1, output2);
```



Model Interconnection

- Feedback interconnection of two models:

```
sys = feedback(sys1, sys2);
```

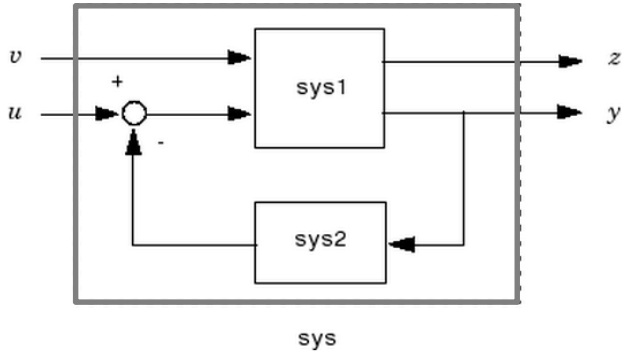


- Both systems must be either both continuous or both discrete with identical sample time.
- Negative feedback is default. To apply positive feedback:

```
sys = feedback(sys1, sys2, +1);
```

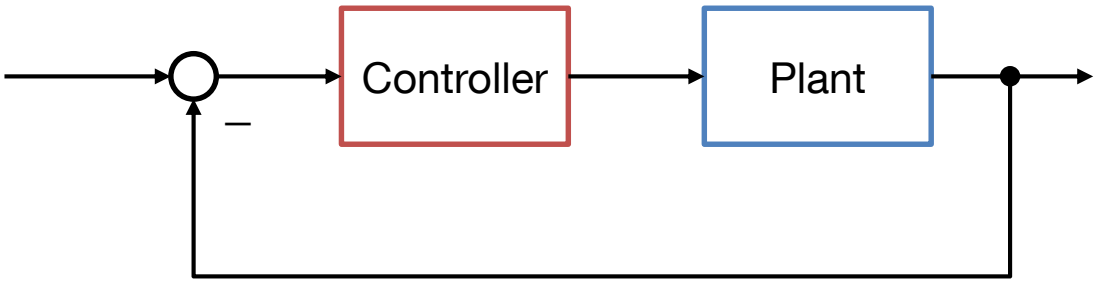
- In the case of MIMO systems, you can specify which inputs and outputs to connect.

```
sys = feedback(sys1, sys2, feedin, feedout);
```



Model Interconnection

- Returning to the initial example…

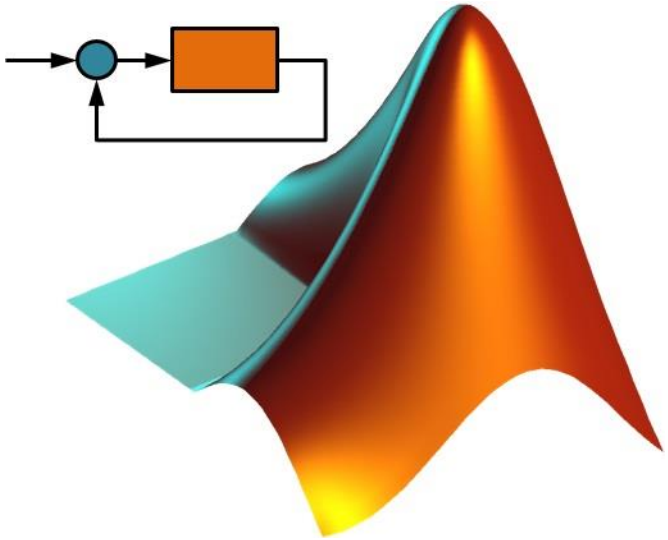


Other functions for model interconnection:
`connect`
`sumblk`
`append`
`blkdiag`
`imp2exp`
`inv`
`lft`

```
open_loop   = series(controller, plant);
closed_loop = feedback(open_loop, 1);
```

Note: the feedback system is a simple unitary gain.

4. Model Transformation



Model Transformation – Type Conversion

- To convert the model type, simply apply the desired function of the new model type with the existing model as argument:

```
sys_tf   = tf(1, [1 1]);   % original model
sys_zpk  = zpk(sys_tf);    % conversion to Zero-Pole-Gain Model
sys_ss   = ss(sys_tf);     % conversion to State-Space Model
sys_pid  = pid(sys_tf);    % conversion to PID Model
sys_pids = pidstd(sys_tf); % conversion to PID (standard form) Model
sys_tf2  = tf(sys_ss);     % conversion to transfer function
```

- Conversion to FRD model requires the desired frequencies as second argument:

```
sys_frd = frd(sys_tf, logspace(-1, 3));
```

- MIMO models can be assembled from SISO models using matrix notation.
  You can also access one particular SISO transfer of a MIMO model.

```
sys_tf_MIMO = [tf(1,[1 1]), 1; tf([1 2],[2 1]), tf(5,[1 3 1])];
sys_tf_SISO21 = sys_tf_MIMO(2,1); % From input 1 to output 2: tf([1 2],[2 1])
```

Model Transformation – Continuous-Discrete Conversion

- There are various functions available for conversions between continuous-time and discrete-time. For instance, to discretize a continuous-time model:

```
sys_discrete = c2d(sys_continuous, sample_time);
```

- To create a continuous-time model from a discrete-time model:

```
sys_continuous = d2c(sys_discrete)
```

- To resample a discrete-time model:

```
sys_rs = d2d(sys, new_sample_time)
```

There are various options for all these functions, which may help improve the result in any particular case.

- To upsample a discrete-time model, i.e., to multiply the sampling rate by a factor `L`:

```
sys_us = upsample(sys, L)
```

Both `d2d` and `upsample` can be used to decrease sample time. Note that `upsample` provides better match in both time and frequency domain, but increases model order to do so. Moreover, it is only applicable if the new sampling rate is an integer multiple of the old sampling rate.

Model Transformation – Simplification

- Models can also be simplified. To eliminate the states of a state-space model that do not affect the input-output response, apply structural pole-zero cancelations:

```
msys = sminreal(sys);
```

- A minimal realization or pole-zero cancelation of any type of dynamic model can be obtained as follows. The resulting system is of minimal order and has the same response characteristics as the original.

```
msys = minreal(sys);
msys = minreal(sys, tolerance);
```

Default tolerance is `sqrt(eps)`. An increased tolerance forces additional cancellations.

- Specific states of a state-space model can be eliminated as follows:

```
rsys = modred(sys, elim);
```

Here, `elim` can be a vector of indices or a logical vector.
This function is usually used in combination with `balreal` (see next slide).

Model Transformation – Simplification

- `balreal` computes a balanced realization. Hankel singular values g are returned as well.

```
sys = zpk([-10 -20.01], [-5 -9.9 -20.1], 1);
[bsys, g] = balreal(sys);
```

- Note that balreal works with state-space models only, so the ZPK object here is automatically transformed to an SS object first.

- The system above contains two near-cancelling pole-zero pairs. This is also reflected by small values in the vector g.

```
g'
```

```
ans =
    0.1006    0.0001    0.0000
```

Other functions for simplification:
`balred`
`hsvd`
`hsvplot`

- Small values of g indicate that the corresponding states can be removed for simplification.

```
rsys = modred(sysb, g<0.001);
```

Model Transformation – State-Coordinate Transform

- The function `canon` transforms linear models into canonical state-space models. The modal form is constituted of the system modes, whereas the companion form explicitly contains the characteristic polynomial coefficients.

Modal form of a system with eigenvalues $(\lambda_1, \sigma \pm j\omega, \lambda_2)$:

$$A = \begin{bmatrix} \lambda_1 & 0 & 0 & 0 \\ 0 & \sigma & \omega & 0 \\ 0 & -\omega & \sigma & 0 \\ 0 & 0 & 0 & \lambda_2 \end{bmatrix}$$

Companion form of a system with char. polynomial $p = s^n + a_1 s^{n-1} + \cdots + a_{n-1} s + a_n$:

$$A = \begin{bmatrix} 0 & 0 & \cdots & \cdots & 0 & -a_n \\ 1 & 0 & 0 & \cdots & 0 & -a_{n-1} \\ 0 & 1 & 0 & \cdots & \vdots & \vdots \\ \vdots & 0 & \ddots & \ddots & \vdots & \vdots \\ 0 & \vdots & \ddots & 1 & 0 & -a_2 \\ 0 & \cdots & \cdots & 0 & 1 & -a_1 \end{bmatrix}$$

```
msys = canon(sys,'modal');
csys = canon(sys,'companion');
```

- The state transformation matrix `T` can also be obtained and applied.

```
[csys,T] = canon(sys,type);
csys_alt = ss2ss(sys,T);      % Note: csys_alt = csys
```

Other functions for state-coordinate transform:
`prescale`
`ss2ss`
`xperm`

Model Transformation – Modal Decomposition

- Models can be decomposed based on the characteristics of the modes:
  - Stable-unstable decomposition using `stablesep`
  - Slow-fast decomposition using `freqsep`
  - Region-based decomposition (`N` regions) using `modsep`

```
[sys_stable, sys_unstable] = stablesep(sys);
[sys_slow, sys_fast]       = freqsep(sys, f_cut);
[H, H0]                    = modsep(sys, N, REGIONFCN);
```

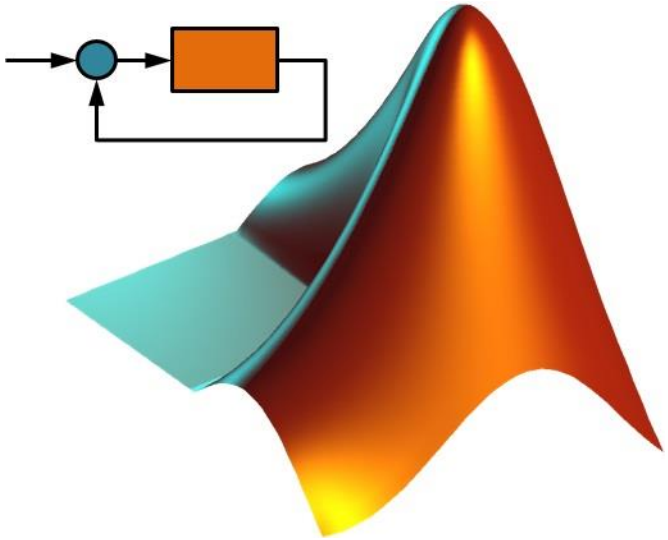
- `REGIONFCN` specifies the regions of interest. For instance, to decompose a system into two systems `H1`, `H2`, which have their poles inside and outside the unit disk, respectively:

Note:
`sys_stable + sys_unstable = sys`
`sys_slow + sys_fast = sys`

```
function r = udsep(p)
if abs(p)<1, r = 1;
else         r = 2;
end
```

```
[H, H0] = modsep(sys, 2, @udpsep);
```

5. Linear Analysis



Linear Analysis – Stability

- After linear dynamic models have been created, connected and possibly transformed, it is time to analyze them. Poles and zeros of a system can be found as follows:

```
vector_of_poles = pole(sys);
vector_of_zeros = zero(sys);
```

- The function `zero` can also return the overall gain:

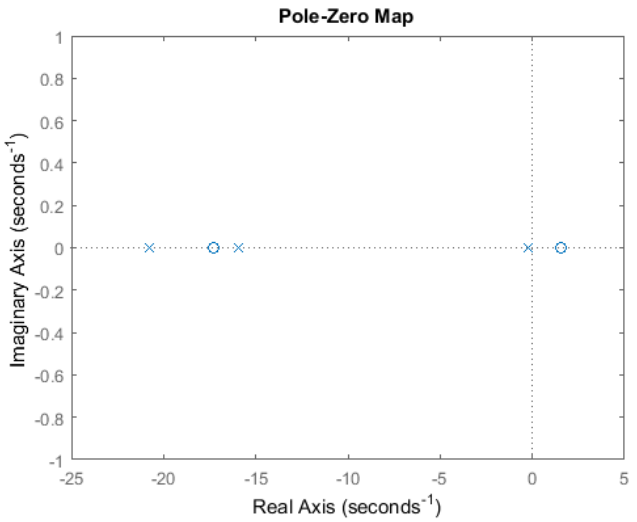
```
[vector_of_zeros, overall_gain] = zero(sys);
```

- To visualize both the poles and the zeros of a system, one of the following functions can be used:

```
pzplot(sys)
pzmap(sys)
```

Multiple different types of systems can be handed over and `pzmap` can return poles and zeros.

```
pzplot(sys1, sys2, sys3)
[poles, zeros] = pzmap(sys)
```



Linear Analysis – Stability

- More detailed information about the poles is provided by the function `damp`.

```
damp(sys) % displays a table of natural frequencies, damping ratios and time constants
[omega, zeta] = damp(sys); % returns natural frequencies and damping ratios
```

Pole	Damping	Frequency (rad/seconds)	Time Constant (seconds)
-1.20e+00	1.00e+00	1.20e+00	8.33e-01
-2.16e+00 + 8.90e-01i	9.25e-01	2.34e+00	4.63e-01
-2.16e+00 - 8.90e-01i	9.25e-01	2.34e+00	4.63e-01

- Stability margins can be determined as follows:

```
[gain_margin, phase_margin, omega_GM, omega_PM] = margin(sys)
```

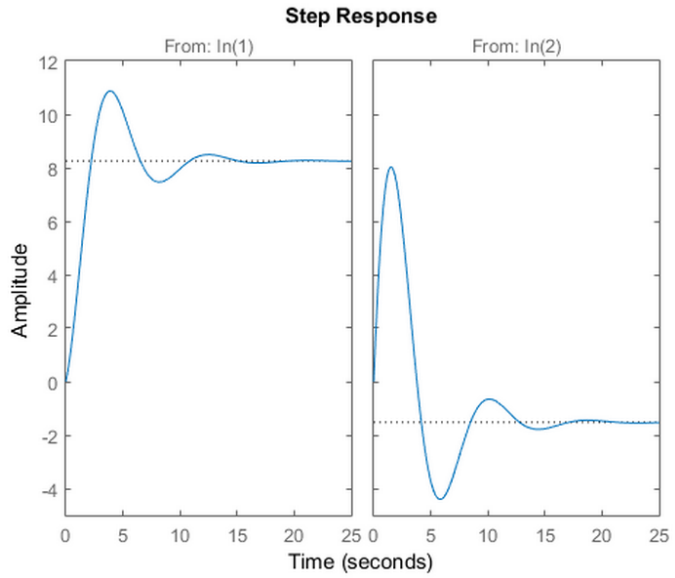
`omega_GM` and `omega_PM` are the frequencies where the gain / phase margin is measured.

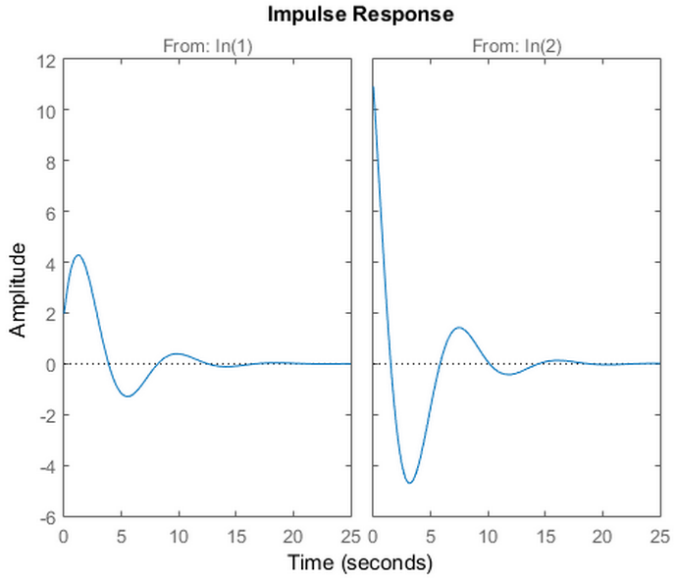
- Calling margin without output arguments creates a Bode plot (see slides below) with margins indicated.

Linear Analysis – Time Domain

- Step and impulse response of linear systems can be easily visualized:

```
a = [-0.5572, -0.7814; 0.7814, 0];
b = [1, -1; 0, 2]; c = [1.9691, 6.4493];
sys = ss(a,b,c,0);
step(sys)
impulse(sys)
```

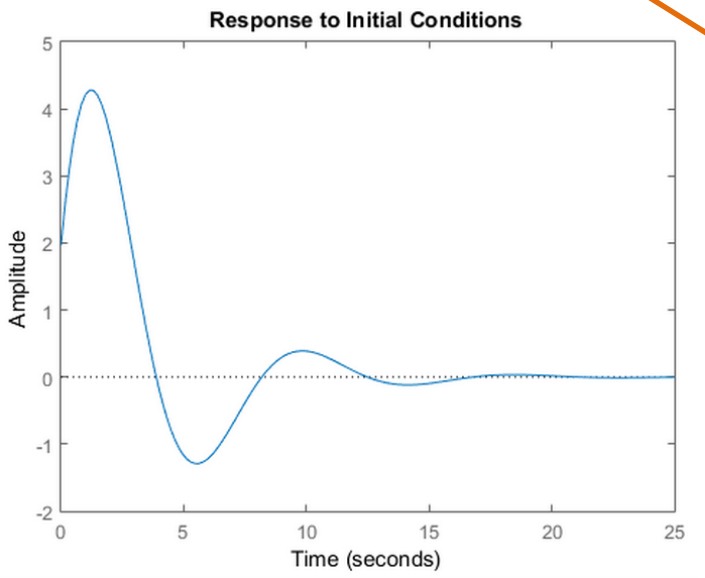




Linear Analysis – Time Domain

- The response to initial conditions can be investigated as follows.

```
a = [-0.5572, -0.7814; 0.7814, 0];
c = [1.9691, 6.4493];
sys = ss(a, [0;0], c, 0);
x0 = [1; 0];
initial(sys, x0)
```

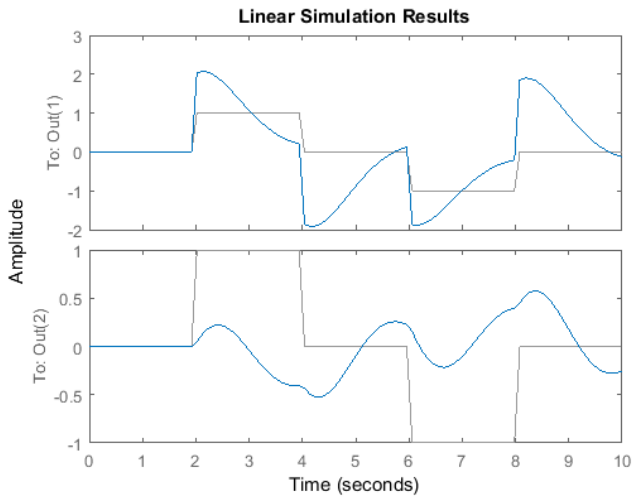


Note: system inputs are irrelevant for the response to initial conditions. Therefore, the input matrix can be set to zero here. However, the dimension of the input matrix has to match

Linear Analysis – Time Domain

- Apart from the common step-, impulse- and initial condition response analysis, the Control System Toolbox also provides a function that returns the time response to arbitrary inputs.

```
sys = [tf([2 5 1],[1 2 3]); tf([1 -1],[1 1 5])];
t = linspace(0,10,100);
L = length(t)/5;
u = [zeros(1,L), ones(1,L), zeros(1,L), -ones(1,L), zeros(1,L)];
lsim(sys,u,t)
```



- The system outputs, simulations time stamps (and states in the case of state-space models) can be retrieved as follows:

```
y = lsim(sys,u,t);
[y,t,x] = lsim(sys,u,t);
```

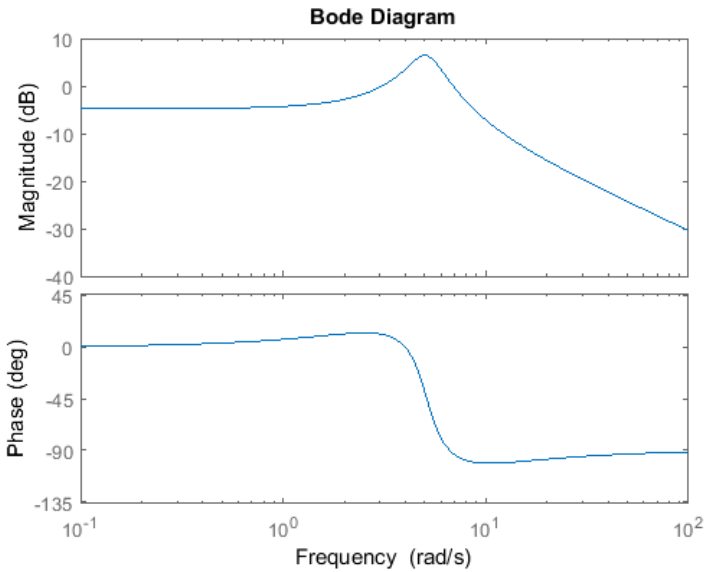
- Initial conditions of state-space models can be specified as a fourth argument:

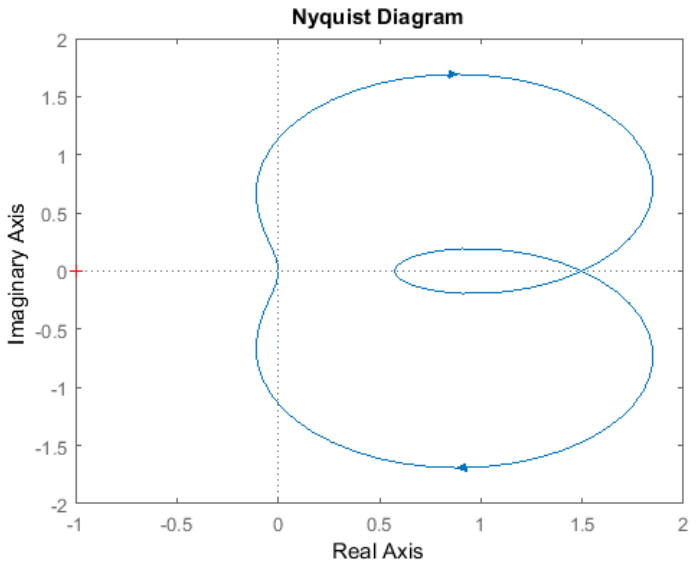
```
lsim(sys,u,t,x0);
```

Linear Analysis – Frequency Domain

- In the frequency domain, models can be analyzed using Bode or Nyquist diagrams.

```
sys = tf([3, 15], [1, 2, 26]);
bode(sys)
nyquist(sys)
```

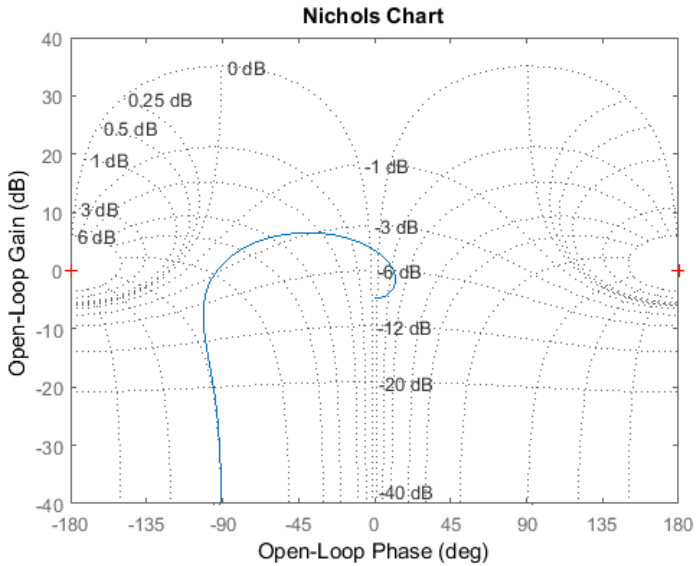




Linear Analysis – Frequency Domain

- A Nichols chart with its grid lines can be created as follows:

```
sys = tf([3, 15], [1, 2, 26]);
nichols(sys)
ngrid
```



- The low-frequency gain of a system, i.e., the transfer function value at s = 0, can be determined as follows.

```
k = dcgain(sys);
```

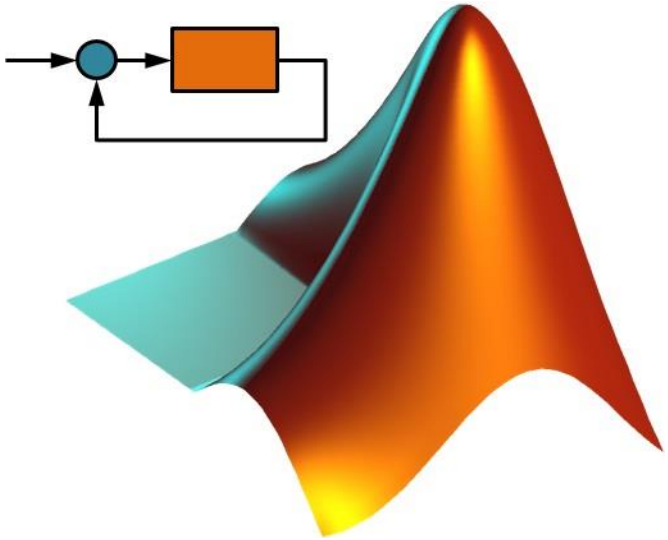
- To find all frequencies where a system has a certain principal gain g

```
omega_c = getGainCrossover(sys, g);
```

- The peak gain of a system is given by the following function.

```
g_peak = getPeakGain(sys);
```

6. Control Design



Control Design – PID Controller Tuning

- The Control System Toolbox provides an automatic PID tuning algorithm:

```
controller = pidtune(sys, 'pi');
```

- The second argument specifies the controller type (`p`, `i`, `pi`, `pd`, `pdf`, `pid` or `pidf`). When handing over an existing controller as second argument, a controller of the same type and form is generated.
- The algorithm tries to balance performance (response time) and robustness (stability margins). You can specify a target crossover frequency or hand over additional options:

```
controller = pidtune(sys, 'p', omega_c_target);
controller = pidtune(sys, 'p', options);
```

- Options need to be created using the following function.

```
options = pidtuneOptions('PhaseMargin', 45, 'DesignFocus', 'reference-tracking');
```

Design focus is either:
`balanced` (default)
`reference-tracking`
`disturbance-rejection`

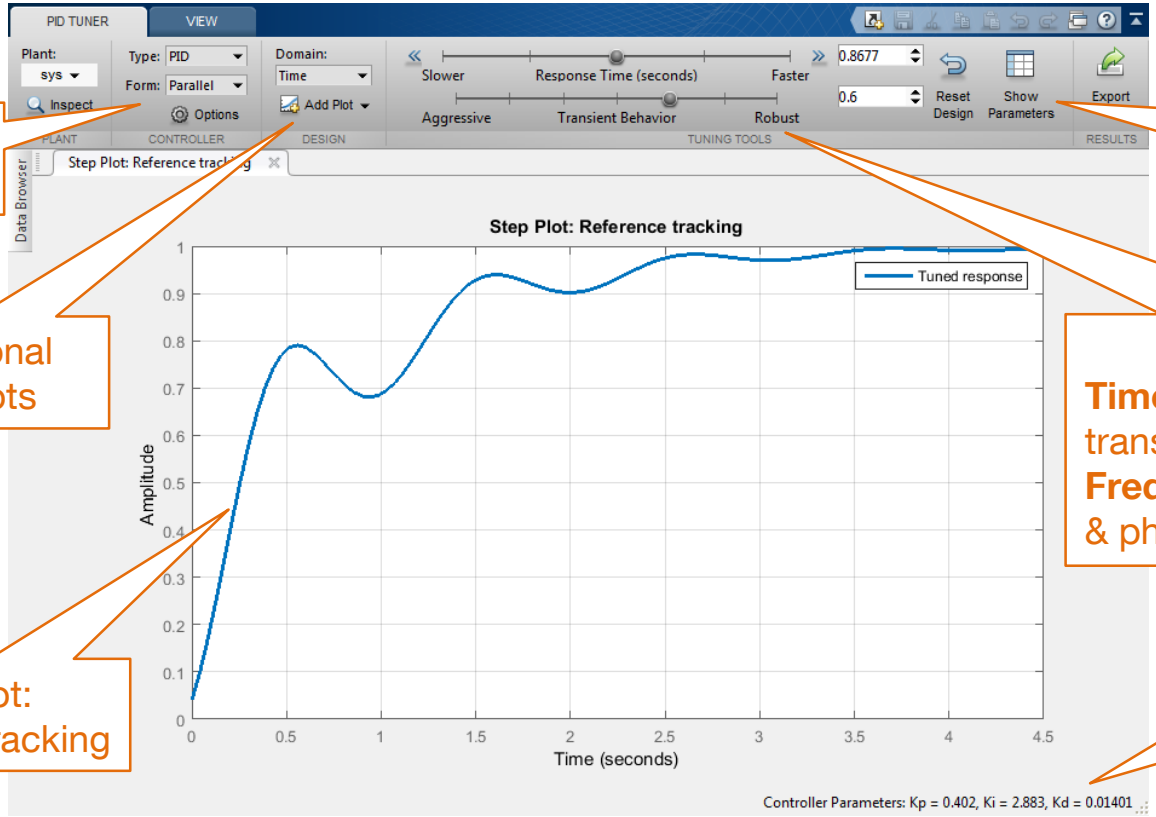
- More information on the resulting closed-loop system can be retrieved as follows.

```
[controller, info] = pidtune(sys, 'pid');
```

Control Design – PID Controller Tuning

- PID controllers can also be tuned using a graphical user interface (GUI)

```
pidTuner(sys, 'pid')
```



Controller type and form

Add additional analysis plots

Analysis plot: reference tracking

New window with all relevant parameters (incl. margins)

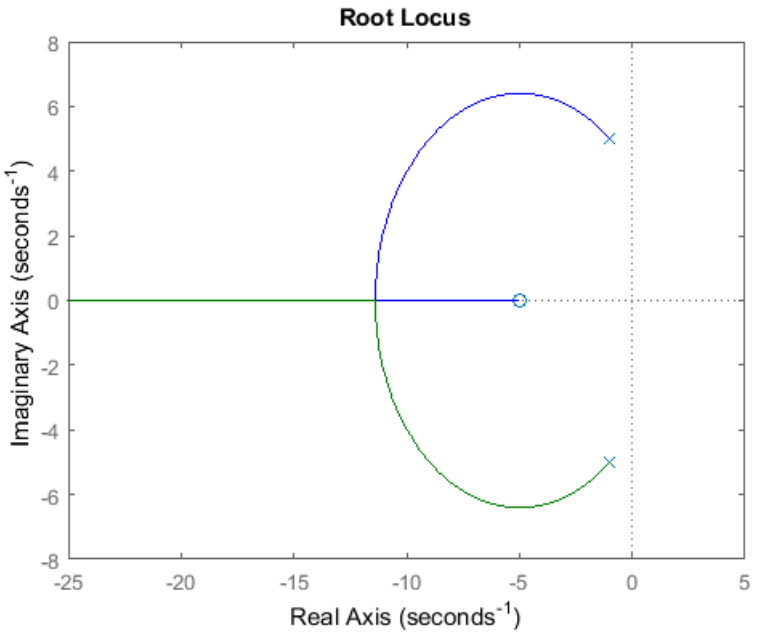
Sliders
Time domain: response time & transient behavior
Frequency domain: bandwidth & phase margin

Current controller parameters

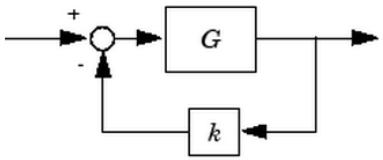
Control Design – SISO Feedback Loops: Root Locus

- The following root locus function may be considered an analysis tool, but can certainly be useful for control design.

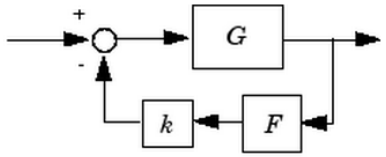
```
rlocus(sys)
```



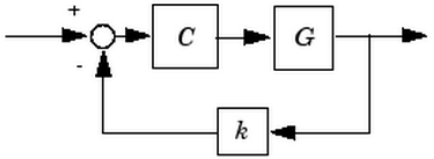
This function can be applied to any of the following **negative** feedback loops:



sys = G



sys = F * G

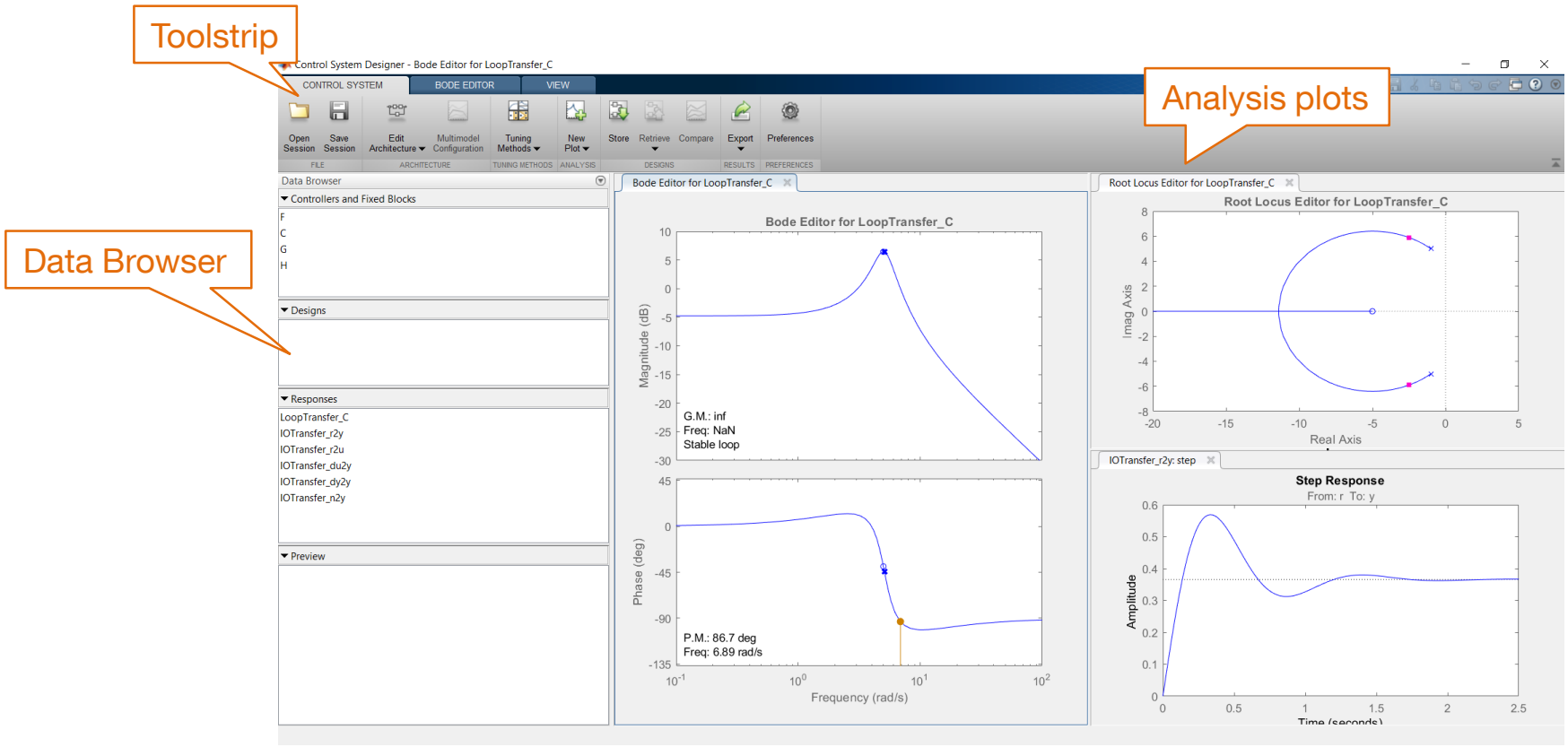


sys = G * C

Control Design – SISO Feedback Loops: SISO Design Tool

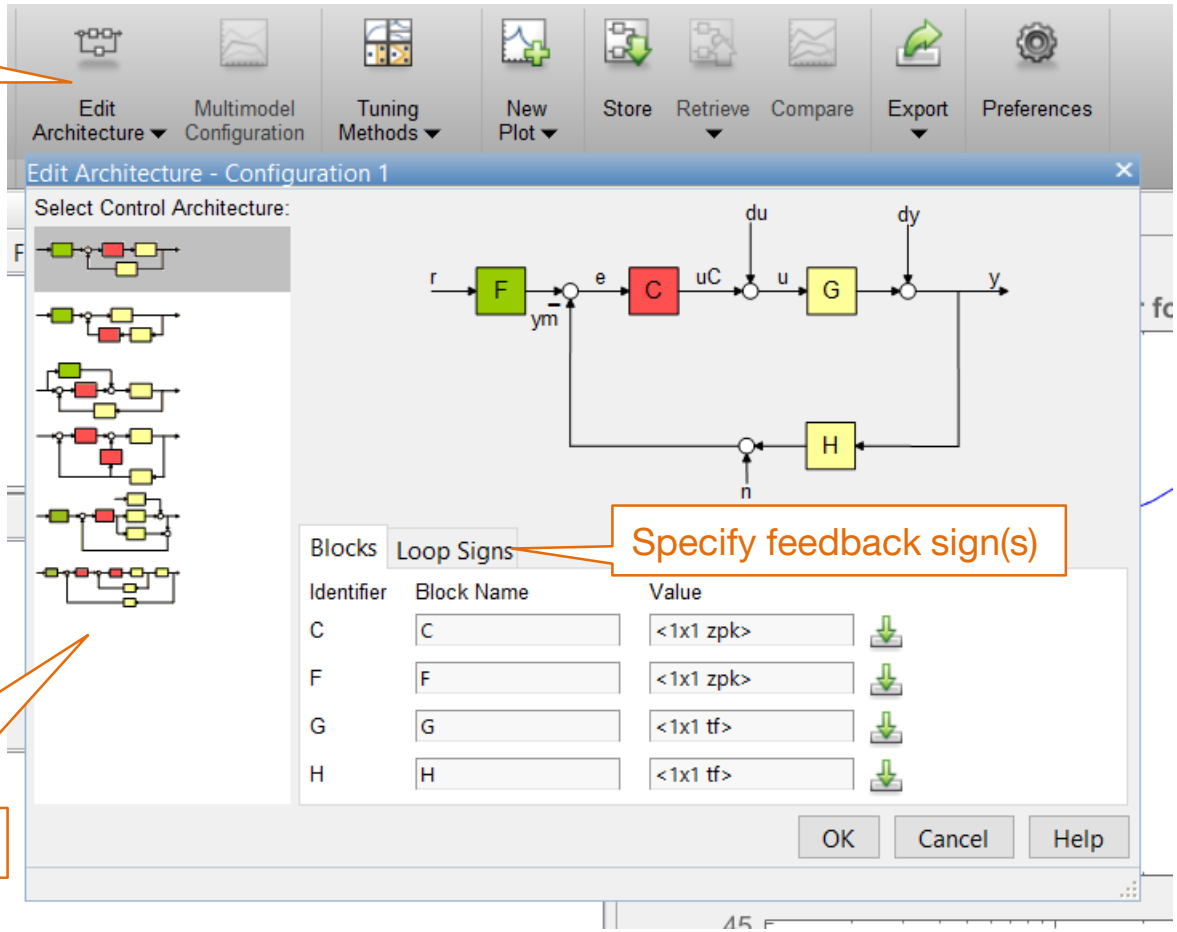
- A comfortable GUI for the design of various controllers is the SISO Design Tool:

```
controlSystemDesigner(sys)
```



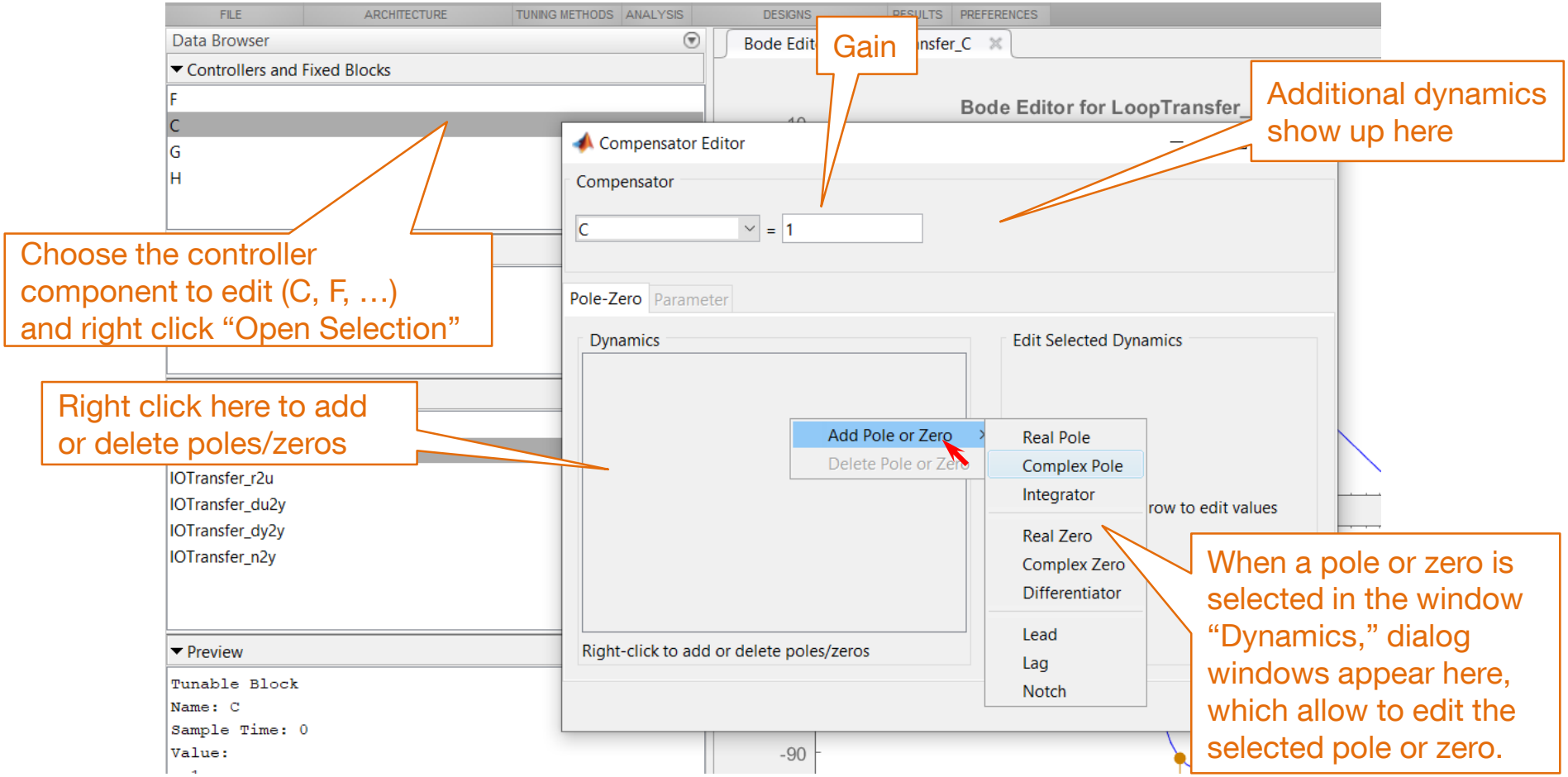
Control Design – SISO Feedback Loops: SISO Design Tool

Click here to change the control architecture…



Choose an architecture!

Control Design – SISO Feedback Loops: SISO Design Tool



Control Design – Pole Placement, LQR, LQG, Kalman Filter

- The Control System Toolbox also comprises functions for more advanced control design. The pole placement gain for a state feedback $u = -Kx$ can be computed as follows.

```
K = place(A,B,p); % A and B are system and input matrix, p is the vector of poles.
```

- A Linear-Quadratic Regulator (LQR) consists of a state feedback gain K that minimizes $J(u) = \int_0^\infty (x^T Q x + u^T R u + 2x^T N u) dt$. The `lqr` function returns the optimal gain `K`, the solution to the Riccati equation `S` and the closed-loop eigenvalues `e`.

```
[K,S,e] = lqr(sys, Q, R, N);
```

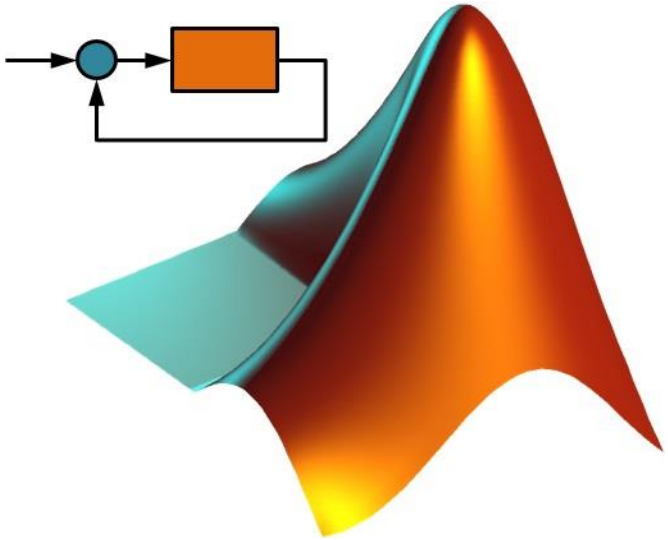
- A Linear-Quadratic-Gaussian (LQG) regulator minimizes a different cost function and can be applied to systems affected by process and measurement noise. With the weighting matrices `QXU` and `QWV` given, the `lqg` function returns this regulator.

```
regulator = lqg(sys, QXU, QWV);
```

- Similarly, a Kalman filter or state estimator can be designed.

```
[k_est,L,P] = kalman(sys, Qn, Rn, Nn)
```

7. Matrix Computations



Matrix Computations

- For state-space models, the controllability and observability matrices can be found by

```
M_ctr = ctrb(sys);
M_obs = obsv(sys);
```

- The H_2 norm of a linear system is returned by

```
N_H2 = norm(sys);
```

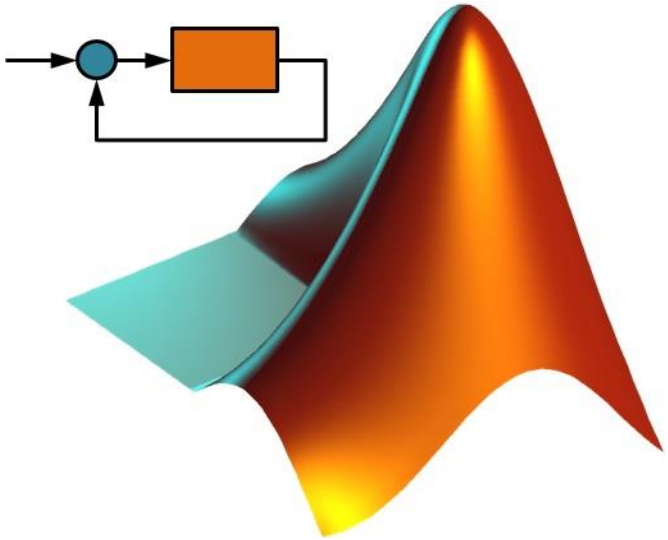
whereas the following command computes the H_∞ norm

```
N_Hinf = norm(sys,inf);
```

- The Lyapunov equation $AX + XA^T + Q = 0$ can be solved using `lyap`.

```
X = lyap(A,Q)
```

8. List of Commands



List of Commands

Command	Explanation	Slide #
`tf`	Transfer function	7
`zpk`	Zero-pole-gain model	7
`ss`	State-space model	7
`dss`	Descriptor state-space model	7
`filt`	Digital filter model	7
`frd`	Freq. response data model	7
`pade`	Padé approximation	12
`absorbDelay`	Replace delays with poles	12
`realp`	Real tunable parameter	13
`genss`	Generalized SS model	13
`genfrd`	Generalized FRD model	14
`pid`	PID controller	15
`pidstd`	PID controller (standard form)	15
`isct`	Check if system is continuous	16
`isdt`	Check if system is discrete	16

Command	Explanation	Slide #
`isstable`	Check if system is stable	16
`isproper`	Check if system is proper	16
`issiso`	Check if system is SISO	16
`hasdelay`	Check if system has delays	16
`order`	Determine system order	16
`series`	Series connection	19
`parallel`	Parallel connection	20
`feedback`	Feedback connection	21
`connect`	Block diagram connection	-
`sumblk`	Summing-junction	-
`append`	Group models	-
`blkdiag`	Block-diagonal concatenat°	-
`im2exp`	Implicit to explicit	-
`inv`	Invert models	-
`lft`	Generalized feedback con.	-

List of Commands

Command	Explanation	Slide #
`c2d`	Continuous to discrete conv.	25
`d2c`	Discrete to continuous conv.	25
`d2d`	Resample	25
`upsample`	Upsample	25
`sminreal`	Structural minimal realization	26
`minreal`	Minimal realization	26
`modred`	Model order reduction	26
`balreal`	Balanced Realization	27
`balred`	Model order reduction	-
`hsvd`	Hankel singular values	-
`hsvplot`	Hankel singular value plot	-
`canon`	Canonical realization	28
`prescale`	Optimal scaling	-
`ss2ss`	State coordinate transform	28
`xperm`	Reorder states	-

Command	Explanation	Slide #
`stablesep`	Stable/unstable decompos.	29
`freqsep`	Slow/fast decomposition	29
`modsep`	Custom mode separation	29
`pole`	System poles	31
`zero`	System zeros	31
`pzplot`	Pole-zero plot	31
`pzmap`	Pole-zero map	31
`damp`	System pole characteristics	32
`margin`	Stability margins	32
`step`	Step response plot	33
`impulse`	Impulse response plot	33
`initial`	Initial value response plot	34
`lsim`	Linear simulation	35

List of Commands

Command	Explanation	Slide #
`bode`	Create Bode diagram	36
`nyquist`	Create Nyquist diagram	36
`nichols`	Create Nichols chart	37
`ngrid`	Plot Nichols grid lines	37
`dcgain`	Compute DC gain	37
`getGainCrossover`	Get crossover frequencies	37
`getPeakGain`	Get peak gain	37
`pidtune`	Tune PID controller	39
`pidtuneOptions`	Options for pidtune	39
`pidTuner`	PID tune GUI	40
`rlocus`	Plot root locus	41
`controlSystemDesigner`	Open SISO Design Tool	42
`place`	Pole placement	45

Command	Explanation	Slide #
`lqr`	Design LQR regulator	45
`lqg`	Design LQG regulator	45
`kalman`	Design Kalman filter	45
`ctrb`	Controllability matrix	47
`obsv`	Observability matrix	47
`norm`	System norm	47
`lyap`	Solve Lyapunov equation	47