Python Control Documentation

Release dev

Python Control Developers

Apr 21, 2017

Contents

The Python Control Systems Library (*python-control*) is a Python package that implements basic operations for analysis and design of feedback control systems.

Features

- Linear input/output systems in state-space and frequency domain
- Block diagram algebra: serial, parallel, and feedback interconnections
- Time response: initial, step, impulse
- Frequency response: Bode and Nyquist plots
- Control analysis: stability, reachability, observability, stability margins
- Control design: eigenvalue placement, linear quadratic regulator
- Estimator design: linear quadratic estimator (Kalman filter)

Documentation

CHAPTER 1

Introduction

Welcome to the Python Control Systems Toolbox (python-control) User's Manual. This manual contains information on using the python-control package, including documentation for all functions in the package and examples illustrating their use.

Overview of the Toolbox

The python-control package is a set of python classes and functions that implement common operations for the analysis and design of feedback control systems. The initial goal is to implement all of the functionality required to work through the examples in the textbook Feedback Systems by Astrom and Murray. A MATLAB compatibility package (control.matlab) is available that provides many of the common functions corresponding to commands available in the MATLAB Control Systems Toolbox.

Some Differences from MATLAB

The python-control package makes use of NumPy and SciPy. A list of general differences between NumPy and MATLAB can be found here.

In terms of the python-control package more specifically, here are some thing to keep in mind:

- You must include commas in vectors. So [1 2 3] must be [1, 2, 3].
- Functions that return multiple arguments use tuples
- You cannot use braces for collections; use tuples instead

Installation

The *python-control* package may be installed using pip, conda or the standard distutils/setuptools mechanisms.

To install using pip:

```
pip install slycot   # optional
pip install control
```

Many parts of *python-control* will work without *slycot*, but some functionality is limited or absent, and installation of *slycot* is recommended.

Note: the *slycot* library only works on some platforms, mostly linux-based. Users should check to insure that slycot is installed correctly by running the command:

```
python -c "import slycot"
```

and verifying that no error message appears. It may be necessary to install *slycot* from source, which requires a working FORTRAN compiler and the *lapack* library. More information on the slycot package can be obtained from the slycot project page.

For users with the Anaconda distribution of Python, the following commands can be used:

```
conda install numpy scipy matplotlib    # if not yet installed
conda install -c python-control -c cyclus slycot control
```

This installs *slycot* and *python-control* from the *python-control* channel and uses the *cyclus* channel to obtain the required *lapack* package.

Alternatively, to use setuptools, first download the source and unpack it. To install in your home directory, use:

```
python setup.py install --user
```

or to install for all users (on Linux or Mac OS):

```
python setup.py build
sudo python setup.py install
```

The package requires *numpy* and *scipy*, and the plotting routines require *matplotlib*. In addition, some routines require the *slycot* module, described above.

Getting Started

There are two different ways to use the package. For the default interface described in *Function reference*, simply import the control package as follows:

```
>>> import control
```

If you want to have a MATLAB-like environment, use the *MATLAB compatibility module*:

```
>>> from control.matlab import *
```

CHAPTER 2

Library conventions

The python-control library uses a set of standard conventions for the way that different types of standard information used by the library.

Time series data

This is a convention for function arguments and return values that represent time series: sequences of values that change over time. It is used throughout the library, for example in the functions *forced_response()*, *step_response()*, *impulse_response()*, and *initial_response()*.

Note: This convention is different from the convention used in the library `scipy.signal`. In Scipy's convention the meaning of rows and columns is interchanged. Thus, all 2D values must be transposed when they are used with functions from `scipy.signal`.

Types:

- **Arguments** can be **arrays**, **matrices**, or **nested lists**.
- **Return values** are **arrays** (not matrices).

The time vector is either 1D, or 2D with shape (1, n):

```
T = [[t1,     t2,     t3,     ..., tn    ]]
```

Input, state, and output all follow the same convention. Columns are different points in time, rows are different components. When there is only one row, a 1D object is accepted or returned, which adds convenience for SISO systems:

```
U = [[u1(t1), u1(t2), u1(t3), ..., u1(tn)]
     [u2(t1), u2(t2), u2(t3), ..., u2(tn)]
     ...
     ...
     [ui(t1), ui(t2), ui(t3), ..., ui(tn)]]
```

```

Same for X, Y
```

So, U[:,2] is the system's input at the third point in time; and U[1] or U[1,:] is the sequence of values for the system's second input.

The initial conditions are either 1D, or 2D with shape (j, 1):

```
X0 = [[x1]
      [x2]
      ...
      ...
      [xj]]
```

As all simulation functions return *arrays*, plotting is convenient:

```
t, y = step(sys)
plot(t, y)
```

The output of a MIMO system can be plotted like this:

```
t, y, x = lsim(sys, u, t)
plot(t, y[0], label='y_0')
plot(t, y[1], label='y_1')
```

The convention also works well with the state space form of linear systems. If `D` is the feedthrough *matrix* of a linear system, and `U` is its input (*matrix* or *array*), then the feedthrough part of the system's response, can be computed like this:

```
ft = D * U
```

Package configuration

The python-control library can be customized to allow for different plotting conventions. The currently configurable options allow the units for Bode plots to be set as dB for gain, degrees for phase and Hertz for frequency (MATLAB conventions) or the gain can be given in magnitude units (powers of 10), corresponding to the conventions used in Feedback Systems.

Variables that can be configured, along with their default values:

- bode_dB (False): Bode plot magnitude plotted in dB (otherwise powers of 10)
- bode_deg (True): Bode plot phase plotted in degrees (otherwise radians)
- bode_Hz (False): Bode plot frequency plotted in Hertz (otherwise rad/sec)
- bode_number_of_samples (None): Number of frequency points in Bode plots
- bode_feature_periphery_decade (1.0): How many decades to include in the frequency range on both sides of features (poles, zeros).

Functions that can be used to set standard configurations:

use_fbs_defaults()	Use Astrom and Murray compatible settings
use_matlab_defaults()	Use MATLAB compatible configuration settings

control.use_fbs_defaults

`control.use_fbs_defaults()`

Use Astrom and Murray compatible settings

- Bode plots plot gain in powers of ten, phase in degrees, frequency in Hertz

control.use_matlab_defaults

`control.use_matlab_defaults()`

Use MATLAB compatible configuration settings

- Bode plots plot gain in dB, phase in degrees, frequency in Hertz

CHAPTER 3

Function reference

The Python Control Systems Library `control` provides common functions for analyzing and designing feedback control systems.

System creation

`ss`(*args)	Create a state space system.
`tf`(*args)	Create a transfer function system.
`frd`(*args)	Construct a Frequency Response Data model, or convert a system
`rss`([states, outputs, inputs])	Create a stable **continuous** random state space object.
`drss`([states, outputs, inputs])	Create a stable **discrete** random state space object.

control.ss

`control.`**`ss`**(**args*)

Create a state space system.

The function accepts either 1, 4 or 5 parameters:

`ss(sys)` Convert a linear system into space system form. Always creates a new system, even if sys is already a StateSpace object.

`ss(A, B, C, D)` Create a state space system from the matrices of its state and output equations:

$$\dot{x} = A \cdot x + B \cdot u$$
$$y = C \cdot x + D \cdot u$$

`ss(A, B, C, D, dt)` Create a discrete-time state space system from the matrices of its state and output equations:

$$x[k+1] = A \cdot x[k] + B \cdot u[k]$$
$$y[k] = C \cdot x[k] + D \cdot u[ki]$$

The matrices can be given as *array like* data types or strings. Everything that the constructor of `numpy.matrix` accepts is permissible here too.

Parameters **sys: StateSpace or TransferFunction** :

A linear system

A: array_like or string :

System matrix

B: array_like or string :

Control matrix

C: array_like or string :

Output matrix

D: array_like or string :

Feed forward matrix

dt: If present, specifies the sampling period and a discrete time :

system is created

Returns **out: :class:‘StateSpace‘** :

The new linear system

Raises **ValueError** :

if matrix sizes are not self-consistent

See also:

tf, *ss2tf*, *tf2ss*

Examples

```
>>> # Create a StateSpace object from four "matrices".
>>> sys1 = ss("1. -2; 3. -4", "5.; 7", "6. 8", "9.")
```

```
>>> # Convert a TransferFunction to a StateSpace object.
>>> sys_tf = tf([2.], [1., 3])
>>> sys2 = ss(sys_tf)
```

control.tf

control.**tf**(**args*)

Create a transfer function system. Can create MIMO systems.

The function accepts either 1 or 2 parameters:

`tf(sys)` Convert a linear system into transfer function form. Always creates a new system, even if sys is already a TransferFunction object.

`tf(num, den)` Create a transfer function system from its numerator and denominator polynomial coefficients.

If *num* and *den* are 1D array_like objects, the function creates a SISO system.

To create a MIMO system, *num* and *den* need to be 2D nested lists of array_like objects. (A 3 dimensional data structure in total.) (For details see note below.)

`tf(num, den, dt)` Create a discrete time transfer function system; dt can either be a positive number indicating the sampling time or 'True' if no specific timebase is given.

Parameters **sys: LTI (StateSpace or TransferFunction)** :

A linear system

num: array_like, or list of list of array_like :

Polynomial coefficients of the numerator

den: array_like, or list of list of array_like :

Polynomial coefficients of the denominator

Returns **out: :class:'TransferFunction'** :

The new linear system

Raises **ValueError** :

if *num* and *den* have invalid or unequal dimensions

TypeError :

if *num* or *den* are of incorrect type

See also:

`ss`, *`ss2tf`*, *`tf2ss`*

Notes

`num[i][j]` contains the polynomial coefficients of the numerator for the transfer function from the (j+1)st input to the (i+1)st output. `den[i][j]` works the same way.

The list `[2, 3, 4]` denotes the polynomial $2s^2 + 3s + 4$.

Examples

```
>>> # Create a MIMO transfer function object
>>> # The transfer function from the 2nd input to the 1st output is
>>> # (3s + 4) / (6s^2 + 5s + 4).
>>> num = [[[1., 2.], [3., 4.]], [[5., 6.], [7., 8.]]]
>>> den = [[[9., 8., 7.], [6., 5., 4.]], [[3., 2., 1.], [-1., -2., -3.]]]
>>> sys1 = tf(num, den)
>>> # Convert a StateSpace to a TransferFunction object.
>>> sys_ss = ss("1. -2; 3. -4", "5.; 7", "6. 8", "9.")
>>> sys2 = tf(sys1)
```

control.frd

`control.`**`frd`**`(`**args*`)`

Construct a Frequency Response Data model, or convert a system

frd models store the (measured) frequency response of a system.

This function can be called in different ways:

`frd(response, freqs)` Create an frd model with the given response data, in the form of complex response vector, at matching frequency freqs [in rad/s]

`frd(sys, freqs)` Convert an LTI system into an frd model with data at frequencies freqs.

Parameters **response: array_like, or list** :

complex vector with the system response

freq: array_lik or lis :

vector with frequencies

sys: LTI (StateSpace or TransferFunction) :

A linear system

Returns **sys: FRD** :

New frequency response system

See also:

`ss`, *`tf`*

control.rss

`control.`**`rss`**`(`*states=1*, *outputs=1*, *inputs=1*`)`

Create a stable **continuous** random state space object.

Parameters **states: integer** :

Number of state variables

inputs: integer :

Number of system inputs

outputs: integer :

Number of system outputs

Returns **sys: StateSpace** :

The randomly created linear system

Raises **ValueError** :

if any input is not a positive integer

See also:

`drss`

Notes

If the number of states, inputs, or outputs is not specified, then the missing numbers are assumed to be 1. The poles of the returned system will always have a negative real part.

control.drss

`control.drss`(*states=1*, *outputs=1*, *inputs=1*)

Create a stable **discrete** random state space object.

Parameters **states: integer** :

Number of state variables

inputs: integer :

Number of system inputs

outputs: integer :

Number of system outputs

Returns **sys: StateSpace** :

The randomly created linear system

Raises **ValueError** :

if any input is not a positive integer

See also:

`rss`

Notes

If the number of states, inputs, or outputs is not specified, then the missing numbers are assumed to be 1. The poles of the returned system will always have a magnitude less than 1.

System interconnections

`append`(*sys)	Group models by appending their inputs and outputs
`connect`(sys, Q, inputv, outputv)	Index-base interconnection of system
`feedback`(sys1[, sys2, sign])	Feedback interconnection between two I/O systems.
`negate`(sys)	Return the negative of a system.
`parallel`(sys1, sys2)	Return the parallel connection sys1 + sys2.
`series`(sys1, sys2)	Return the series connection sys2 * sys1 for –> sys1 –> sys2 –>.

control.append

`control.append(*sys)`

Group models by appending their inputs and outputs

Forms an augmented system model, and appends the inputs and outputs together. The system type will be the type of the first system given; if you mix state-space systems and gain matrices, make sure the gain matrices are not first.

Parameters sys1, sys2, ... sysn: StateSpace or Transferfunction :

LTI systems to combine

Returns sys: LTI system :

Combined LTI system, with input/output vectors consisting of all input/output vectors appended

Examples

```
>>> sys1 = ss("1. -2; 3. -4", "5.; 7", "6. 8", "9.")
>>> sys2 = ss("-1.", "1.", "1.", "0.")
>>> sys = append(sys1, sys2)
```

Todo

also implement for transfer function, zpk, etc.

control.connect

`control.connect(sys, Q, inputv, outputv)`

Index-base interconnection of system

The system sys is a system typically constructed with append, with multiple inputs and outputs. The inputs and outputs are connected according to the interconnection matrix Q, and then the final inputs and outputs are trimmed according to the inputs and outputs listed in inputv and outputv.

Note: to have this work, inputs and outputs start counting at 1!!!!

Parameters sys: StateSpace Transferfunction :

System to be connected

Q: 2d array :

Interconnection matrix. First column gives the input to be connected second column gives the output to be fed into this input. Negative values for the second column mean the feedback is negative, 0 means no connection is made

inputv: 1d array :

list of final external inputs

outputv: 1d array :

list of final external outputs

Returns sys: LTI system :

Connected and trimmed LTI system

Examples

```
>>> sys1 = ss("1. -2; 3. -4", "5.; 7", "6, 8", "9.")
>>> sys2 = ss("-1.", "1.", "1.", "0.")
>>> sys = append(sys1, sys2)
>>> Q = sp.mat([ [ 1, 2], [2, -1] ]) # basically feedback, output 2 in 1
>>> sysc = connect(sys, Q, [2], [1, 2])
```

control.feedback

`control.`**`feedback`**(*sys1*, *sys2=1*, *sign=-1*)

Feedback interconnection between two I/O systems.

Parameters sys1: scalar, StateSpace, TransferFunction, FRD :

The primary plant.

sys2: scalar, StateSpace, TransferFunction, FRD :

The feedback plant (often a feedback controller).

sign: scalar :

The sign of feedback. *sign* = -1 indicates negative feedback, and *sign* = 1 indicates positive feedback. *sign* is an optional argument; it assumes a value of -1 if not specified.

Returns out: StateSpace or TransferFunction :

Raises ValueError :

if *sys1* does not have as many inputs as *sys2* has outputs, or if *sys2* does not have as many inputs as *sys1* has outputs

NotImplementedError :

if an attempt is made to perform a feedback on a MIMO TransferFunction object

See also:

`series`, *`parallel`*

Notes

This function is a wrapper for the feedback function in the StateSpace and TransferFunction classes. It calls TransferFunction.feedback if *sys1* is a TransferFunction object, and StateSpace.feedback if *sys1* is a StateSpace object. If *sys1* is a scalar, then it is converted to *sys2*'s type, and the corresponding feedback function is used. If *sys1* and *sys2* are both scalars, then TransferFunction.feedback is used.

control.negate

`control.`**`negate`**(*sys*)

Return the negative of a system.

Parameters sys: StateSpace, TransferFunction or FRD :

Returns **out: StateSpace or TransferFunction** :

Notes

This function is a wrapper for the __neg__ function in the StateSpace and TransferFunction classes. The output type is the same as the input type.

If both systems have a defined timebase (dt = 0 for continuous time, dt > 0 for discrete time), then the timebase for both systems must match. If only one of the system has a timebase, the return timebase will be set to match it.

Examples

```
>>> sys2 = negate(sys1) # Same as sys2 = -sys1.
```

control.parallel

`control.parallel(sys1, sys2)`

Return the parallel connection sys1 + sys2.

Parameters **sys1: scalar, StateSpace, TransferFunction, or FRD** :

sys2: scalar, StateSpace, TransferFunction, or FRD :

Returns **out: scalar, StateSpace, or TransferFunction** :

Raises **ValueError** :

if *sys1* and *sys2* do not have the same numbers of inputs and outputs

See also:

series, *feedback*

Notes

This function is a wrapper for the __add__ function in the StateSpace and TransferFunction classes. The output type is usually the type of *sys1*. If *sys1* is a scalar, then the output type is the type of *sys2*.

If both systems have a defined timebase (dt = 0 for continuous time, dt > 0 for discrete time), then the timebase for both systems must match. If only one of the system has a timebase, the return timebase will be set to match it.

Examples

```
>>> sys3 = parallel(sys1, sys2) # Same as sys3 = sys1 + sys2.
```

control.series

control.**series**(*sys1*, *sys2*)

Return the series connection sys2 * sys1 for –> sys1 –> sys2 –>.

Parameters sys1: scalar, StateSpace, TransferFunction, or FRD :

sys2: scalar, StateSpace, TransferFunction, or FRD :

Returns out: scalar, StateSpace, or TransferFunction :

Raises ValueError :

if *sys2.inputs* does not equal *sys1.outputs* if *sys1.dt* is not compatible with *sys2.dt*

See also:

parallel, *feedback*

Notes

This function is a wrapper for the __mul__ function in the StateSpace and TransferFunction classes. The output type is usually the type of *sys2*. If *sys2* is a scalar, then the output type is the type of *sys1*.

If both systems have a defined timebase (dt = 0 for continuous time, dt > 0 for discrete time), then the timebase for both systems must match. If only one of the system has a timebase, the return timebase will be set to match it.

Examples

```
>>> sys3 = series(sys1, sys2) # Same as sys3 = sys2 * sys1.
```

Frequency domain plotting

bode_plot(syslist[, omega, dB, Hz, deg, ...])	Bode plot for a system
nyquist_plot(syslist[, omega, Plot, color, ...])	Nyquist plot for a system
gangof4_plot(P, C[, omega])	Plot the “Gang of 4” transfer functions for a system
nichols_plot(syslist[, omega, grid])	Nichols plot for a system

control.bode_plot

control.**bode_plot**(*syslist*, *omega=None*, *dB=None*, *Hz=None*, *deg=None*, *Plot=True*, *omega_limits=None*, *omega_num=None*, **args*, ***kwargs*)

Bode plot for a system

Plots a Bode plot for the system over a (optional) frequency range.

Parameters syslist : linsys

List of linear input/output systems (single system is OK)

omega : freq_range

Range of frequencies in rad/sec

dB : boolean

If True, plot result in dB

Hz : boolean

If True, plot frequency in Hz (omega must be provided in rad/sec)

deg : boolean

If True, plot phase in degrees (else radians)

Plot : boolean

If True, plot magnitude and phase

omega_limits: tuple, list, ... of two values :

Limits of the to generate frequency vector. If Hz=True the limits are in Hz otherwise in rad/s.

omega_num: int :

number of samples

***args, **kwargs:** :

Additional options to matplotlib (color, linestyle, etc)

Returns **mag** : array (list if len(syslist) > 1)

magnitude

phase : array (list if len(syslist) > 1)

phase in radians

omega : array (list if len(syslist) > 1)

frequency in rad/sec

Notes

1. Alternatively, you may use the lower-level method (mag, phase, freq) = sys.freqresp(freq) to generate the frequency response for a system, but it returns a MIMO response.

2. If a discrete time model is given, the frequency response is plotted along the upper branch of the unit circle, using the mapping z = exp(j omega dt) where omega ranges from 0 to pi/dt and dt is the discrete time base. If not timebase is specified (dt = True), dt is set to 1.

Examples

```
>>> sys = ss("1. -2; 3. -4", "5.; 7", "6. 8", "9.")
>>> mag, phase, omega = bode(sys)
```

control.nyquist_plot

control.**nyquist_plot**(*syslist*, *omega=None*, *Plot=True*, *color='b'*, *labelFreq=0*, **args*, ***kwargs*)

Nyquist plot for a system

Plots a Nyquist plot for the system over a (optional) frequency range.

Parameters **syslist** : list of LTI

List of linear input/output systems (single system is OK)

omega : freq_range

Range of frequencies (list or bounds) in rad/sec

Plot : boolean

If True, plot magnitude

labelFreq : int

Label every nth frequency on the plot

***args, **kwargs:** :

Additional options to matplotlib (color, linestyle, etc)

Returns **real** : array

real part of the frequency response array

imag : array

imaginary part of the frequency response array

freq : array

frequencies

Examples

```
>>> sys = ss("1. -2; 3. -4", "5.; 7", "6. 8", "9.")
>>> real, imag, freq = nyquist_plot(sys)
```

control.gangof4_plot

control.**gangof4_plot**(*P*, *C*, *omega=None*)

Plot the "Gang of 4" transfer functions for a system

Generates a 2x2 plot showing the "Gang of 4" sensitivity functions [T, PS; CS, S]

Parameters **P, C** : LTI

Linear input/output systems (process and control)

omega : array

Range of frequencies (list or bounds) in rad/sec

Returns **None** :

control.nichols_plot

control.**nichols_plot**(*syslist*, *omega=None*, *grid=True*)

Nichols plot for a system

Plots a Nichols plot for the system over a (optional) frequency range.

Parameters **syslist** : list of LTI, or LTI

List of linear input/output systems (single system is OK)

omega : array_like

Range of frequencies (list or bounds) in rad/sec

grid : boolean, optional

True if the plot should include a Nichols-chart grid. Default is True.

Returns None :

Time domain simulation

`forced_response`(sys[, T, U, X0, transpose])	Simulate the output of a linear system.
`impulse_response`(sys[, T, X0, input, ...])	Impulse response of a linear system
`initial_response`(sys[, T, X0, input, ...])	Initial condition response of a linear system
`step_response`(sys[, T, X0, input, output, ...])	Step response of a linear system
`phase_plot`(odefun[, X, Y, scale, X0, T, ...])	Phase plot for 2D dynamical systems

control.forced_response

`control.`**`forced_response`**(*sys*, *T=None*, *U=0.0*, *X0=0.0*, *transpose=False*)

Simulate the output of a linear system.

As a convenience for parameters *U*, *X0*: Numbers (scalars) are converted to constant arrays with the correct shape. The correct shape is inferred from arguments *sys* and *T*.

For information on the **shape** of parameters *U*, *T*, *X0* and return values *T*, *yout*, *xout*, see *Time series data*.

Parameters sys: LTI (StateSpace, or TransferFunction) :

LTI system to simulate

T: array-like :

Time steps at which the input is defined; values must be evenly spaced.

U: array-like or number, optional :

Input array giving input at each time *T* (default = 0).

If *U* is `None` or `0`, a special algorithm is used. This special algorithm is faster than the general algorithm, which is used otherwise.

X0: array-like or number, optional :

Initial condition (default = 0).

transpose: bool :

If True, transpose all input and output arrays (for backward compatibility with MATLAB and scipy.signal.lsim)

Returns T: array :

Time values of the output.

yout: array :

Response of the system.

xout: array :

Time evolution of the state vector.

See also:

step_response, *initial_response*, *impulse_response*

Examples

```
>>> T, yout, xout = forced_response(sys, T, u, X0)
```

See *Time series data*.

control.impulse_response

`control.`**`impulse_response`**(*sys*, *T=None*, *X0=0.0*, *input=0*, *output=None*, *transpose=False*, *return_x=False*)

Impulse response of a linear system

If the system has multiple inputs or outputs (MIMO), one input has to be selected for the simulation. Optionally, one output may be selected. The parameters *input* and *output* do this. All other inputs are set to 0, all other outputs are ignored.

For information on the **shape** of parameters *T*, *X0* and return values *T*, *yout*, see *Time series data*.

Parameters **sys: StateSpace, TransferFunction** :

LTI system to simulate

T: array-like object, optional :

Time vector (argument is autocomputed if not given)

X0: array-like object or number, optional :

Initial condition (default = 0)

Numbers are converted to constant arrays with the correct shape.

input: int :

Index of the input that will be used in this simulation.

output: int :

Index of the output that will be used in this simulation. Set to None to not trim outputs

transpose: bool :

If True, transpose all input and output arrays (for backward compatibility with MATLAB and scipy.signal.lsim)

return_x: bool :

If True, return the state vector (default = False).

Returns **T: array** :

Time values of the output

yout: array :

Response of the system

xout: array :

Individual response of each x variable

See also:

`ForcedReponse`, *initial_response*, *step_response*

Examples

```
>>> T, yout = impulse_response(sys, T, X0)
```

control.initial_response

`control.initial_response`(*sys*, *T=None*, *X0=0.0*, *input=0*, *output=None*, *transpose=False*, *return_x=False*)

Initial condition response of a linear system

If the system has multiple outputs (MIMO), optionally, one output may be selected. If no selection is made for the output, all outputs are given.

For information on the **shape** of parameters *T*, *X0* and return values *T*, *yout*, see *Time series data*.

Parameters sys: StateSpace, or TransferFunction :

LTI system to simulate

T: array-like object, optional :

Time vector (argument is autocomputed if not given)

X0: array-like object or number, optional :

Initial condition (default = 0)

Numbers are converted to constant arrays with the correct shape.

input: int :

Ignored, has no meaning in initial condition calculation. Parameter ensures compatibility with step_response and impulse_response

output: int :

Index of the output that will be used in this simulation. Set to None to not trim outputs

transpose: bool :

If True, transpose all input and output arrays (for backward compatibility with MATLAB and scipy.signal.lsim)

return_x: bool :

If True, return the state vector (default = False).

Returns T: array :

Time values of the output

yout: array :

Response of the system

xout: array :

Individual response of each x variable

See also:

forced_response, *impulse_response*, *step_response*

Examples

```
>>> T, yout = initial_response(sys, T, X0)
```

control.step_response

`control.`**`step_response`**(*sys*, *T=None*, *X0=0.0*, *input=None*, *output=None*, *transpose=False*, *return_x=False*)

Step response of a linear system

If the system has multiple inputs or outputs (MIMO), one input has to be selected for the simulation. Optionally, one output may be selected. The parameters *input* and *output* do this. All other inputs are set to 0, all other outputs are ignored.

For information on the **shape** of parameters *T*, *X0* and return values *T*, *yout*, see *Time series data*.

Parameters sys: StateSpace, or TransferFunction :

LTI system to simulate

T: array-like object, optional :

Time vector (argument is autocomputed if not given)

X0: array-like or number, optional :

Initial condition (default = 0)

Numbers are converted to constant arrays with the correct shape.

input: int :

Index of the input that will be used in this simulation.

output: int :

Index of the output that will be used in this simulation. Set to None to not trim outputs

transpose: bool :

If True, transpose all input and output arrays (for backward compatibility with MATLAB and scipy.signal.lsim)

return_x: bool :

If True, return the state vector (default = False).

Returns T: array :

Time values of the output

yout: array :

Response of the system

xout: array :

Individual response of each x variable

See also:

forced_response, *initial_response*, *impulse_response*

Examples

```
>>> T, yout = step_response(sys, T, X0)
```

control.phase_plot

`control.phase_plot`(*odefun*, *X=None*, *Y=None*, *scale=1*, *X0=None*, *T=None*, *lingrid=None*, *lintime=None*, *logtime=None*, *timepts=None*, *parms=()*, *verbose=True*)

Phase plot for 2D dynamical systems

Produces a vector field or stream line plot for a planar system.

Call signatures: phase_plot(func, X, Y, ...) - display vector field on meshgrid phase_plot(func, X, Y, scale, ...) - scale arrows phase_plot(func. X0=(...), T=Tmax, ...) - display stream lines phase_plot(func, X, Y, X0=[...], T=Tmax, ...) - plot both phase_plot(func, X0=[...], T=Tmax, lingrid=N, ...) - plot both phase_plot(func, X0=[...], lintime=N, ...) - stream lines with arrows

Parameters **func** : callable(x, t, ...)

Computes the time derivative of y (compatible with odeint). The function should be the same for as used for scipy.integrate. Namely, it should be a function of the form dxdt = F(x, t) that accepts a state x of dimension 2 and returns a derivative dx/dt of dimension 2.

X, Y: ndarray, optional :

Two 1-D arrays representing x and y coordinates of a grid. These arguments are passed to meshgrid and generate the lists of points at which the vector field is plotted. If absent (or None), the vector field is not plotted.

scale: float, optional :

Scale size of arrows; default = 1

X0: ndarray of initial conditions, optional :

List of initial conditions from which streamlines are plotted. Each initial condition should be a pair of numbers.

T: array-like or number, optional :

Length of time to run simulations that generate streamlines. If a single number, the same simulation time is used for all initial conditions. Otherwise, should be a list of length len(X0) that gives the simulation time for each initial condition. Default value = 50.

lingrid = N or (N, M): integer or 2-tuple of integers, optional :

If X0 is given and X, Y are missing, a grid of arrows is produced using the limits of the initial conditions, with N grid points in each dimension or N grid points in x and M grid points in y.

lintime = N: integer, optional :

Draw N arrows using equally space time points

logtime = (N, lambda): (integer, float), optional :

Draw N arrows using exponential time constant lambda

timepts = [t1, t2, ...]: array-like, optional :

Draw arrows at the given list times

parms: tuple, optional :

List of parameters to pass to vector field: *func(x, t, *parms)*

See also:

`box_grid`, `Y`

Block diagram algebra

`series`(sys1, sys2)	Return the series connection sys2 * sys1 for –> sys1 –> sys2 –>.
`parallel`(sys1, sys2)	Return the parallel connection sys1 + sys2.
`feedback`(sys1[, sys2, sign])	Feedback interconnection between two I/O systems.
`negate`(sys)	Return the negative of a system.

Control system analysis

`dcgain`(sys)	Return the zero-frequency (or DC) gain of the given system
`evalfr`(sys, x)	Evaluate the transfer function of an LTI system for a single complex number x.
`freqresp`(sys, omega)	Frequency response of an LTI system at multiple angular frequencies.
`margin`(*args)	Calculate gain and phase margins and associated crossover frequencies
`stability_margins`(sysdata[, returnall, epsw])	Calculate stability margins and associated crossover frequencies.
`phase_crossover_frequencies`(sys)	Compute frequencies and gains at intersections with real axis in Nyquist plot.
`pole`(sys)	Compute system poles.
`zero`(sys)	Compute system zeros.
`pzmap`(sys[, Plot, title])	Plot a pole/zero map for a linear system.
`root_locus`(sys[, kvect, xlim, ylim, ...])	Root locus plot

control.dcgain

`control.`**`dcgain`**(*sys*)

Return the zero-frequency (or DC) gain of the given system

Returns gain : ndarray

The zero-frequency gain, or np.nan if the system has a pole at the origin

control.evalfr

control.**evalfr**(*sys*, *x*)

Evaluate the transfer function of an LTI system for a single complex number x.

To evaluate at a frequency, enter x = omega*j, where omega is the frequency in radians

Parameters **sys: StateSpace or TransferFunction** :

Linear system

x: scalar :

Complex number

Returns **fresp: ndarray** :

See also:

freqresp, bode

Notes

This function is a wrapper for StateSpace.evalfr and TransferFunction.evalfr.

Examples

```
>>> sys = ss("1. -2; 3. -4", "5.; 7", "6. 8", "9.")
>>> evalfr(sys, 1j)
array([[ 44.8-21.4j]])
>>> # This is the transfer function matrix evaluated at s = i.
```

Todo

Add example with MIMO system

control.freqresp

control.**freqresp**(*sys*, *omega*)

Frequency response of an LTI system at multiple angular frequencies.

Parameters **sys: StateSpace or TransferFunction** :

Linear system

omega: array_like :

List of frequencies

Returns **mag: ndarray** :

phase: ndarray :

omega: list, tuple, or ndarray :

See also:

evalfr, bode

Notes

This function is a wrapper for StateSpace.freqresp and TransferFunction.freqresp. The output omega is a sorted version of the input omega.

Examples

```
>>> sys = ss("1. -2; 3. -4", "5.; 7", "6. 8", "9.")
>>> mag, phase, omega = freqresp(sys, [0.1, 1., 10.])
>>> mag
array([[[ 58.8576682 ,  49.64876635,  13.40825927]]])
>>> phase
array([[[-0.05408304, -0.44563154, -0.66837155]]])
```

Todo

Add example with MIMO system

#>>> sys = rss(3, 2, 2) #>>> mag, phase, omega = freqresp(sys, [0.1, 1., 10.]) #>>> mag[0, 1, :] #array([55.43747231, 42.47766549, 1.97225895]) #>>> phase[1, 0, :] #array([-0.12611087, -1.14294316, 2.5764547]) #>>> # This is the magnitude of the frequency response from the 2nd #>>> # input to the 1st output, and the phase (in radians) of the #>>> # frequency response from the 1st input to the 2nd output, for #>>> # s = 0.1i, i, 10i.

control.margin

`control.`**`margin`**`(`*`*args`*`)`

Calculate gain and phase margins and associated crossover frequencies

Parameters **sysdata: LTI system or (mag, phase, omega) sequence** :

sys [StateSpace or TransferFunction] Linear SISO system

mag, phase, omega [sequence of array_like] Input magnitude, phase (in deg.), and frequencies (rad/sec) from bode frequency response data

Returns **gm** : float

Gain margin

pm [float] Phase margin (in degrees)

Wcg [float] Gain crossover frequency (corresponding to phase margin)

Wcp [float] Phase crossover frequency (corresponding to gain margin) (in rad/sec)

Margins are of SISO open-loop. If more than one crossover frequency is :

detected, returns the lowest corresponding margin. :

Examples

```
>>> sys = tf(1, [1, 2, 1, 0])
>>> gm, pm, Wcg, Wcp = margin(sys)
```

control.stability_margins

`control.`**`stability_margins`**(*sysdata*, *returnall=False*, *epsw=1e-08*)

Calculate stability margins and associated crossover frequencies.

Parameters **sysdata: LTI system or (mag, phase, omega) sequence** :

sys [LTI system] Linear SISO system

mag, phase, omega [sequence of array_like] Arrays of magnitudes (absolute values, not dB), phases (degrees), and corresponding frequencies. Crossover frequencies returned are in the same units as those in *omega* (e.g., rad/sec or Hz).

returnall: bool, optional :

If true, return all margins found. If false (default), return only the minimum stability margins. For frequency data or FRD systems, only one margin is found and returned.

epsw: float, optional :

Frequencies below this value (default 1e-8) are considered static gain, and not returned as margin.

Returns **gm: float or array_like** :

Gain margin

pm: float or array_loke :

Phase margin

sm: float or array_like :

Stability margin, the minimum distance from the Nyquist plot to -1

wg: float or array_like :

Gain margin crossover frequency (where phase crosses -180 degrees)

wp: float or array_like :

Phase margin crossover frequency (where gain crosses 0 dB)

ws: float or array_like :

Stability margin frequency (where Nyquist plot is closest to -1)

control.phase_crossover_frequencies

`control.`**`phase_crossover_frequencies`**(*sys*)

Compute frequencies and gains at intersections with real axis in Nyquist plot.

Call as: omega, gain = phase_crossover_frequencies()

Returns **omega: 1d array of (non-negative) frequencies where Nyquist plot** :

intersects the real axis :

gain: 1d array of corresponding gains :

Examples

```
>>> tf = TransferFunction([1], [1, 2, 3, 4])
>>> PhaseCrossoverFrequenies(tf)
(array([ 1.73205081,  0.        ]), array([-0.5 ,  0.25]))
```

control.pole

`control.`**`pole`**(*sys*)

Compute system poles.

Parameters sys: StateSpace or TransferFunction :

Linear system

Returns poles: ndarray :

Array that contains the system's poles.

Raises NotImplementedError :

when called on a TransferFunction object

See also:

`zero`, *`TransferFunction.pole`*, *`StateSpace.pole`*

control.zero

`control.`**`zero`**(*sys*)

Compute system zeros.

Parameters sys: StateSpace or TransferFunction :

Linear system

Returns zeros: ndarray :

Array that contains the system's zeros.

Raises NotImplementedError :

when called on a MIMO system

See also:

`pole`, *`StateSpace.zero`*, *`TransferFunction.zero`*

control.pzmap

`control.`**`pzmap`**(*sys*, *Plot=True*, *title='Pole Zero Map'*)

Plot a pole/zero map for a linear system.

Parameters sys: LTI (StateSpace or TransferFunction) :

Linear system for which poles and zeros are computed.

Plot: bool :

If `True` a graph is generated with Matplotlib, otherwise the poles and zeros are only computed and returned.

Returns pole: array :

The systems poles

zeros: array :

The system's zeros.

control.root_locus

`control.`**`root_locus`**(*sys*, *kvect=None*, *xlim=None*, *ylim=None*, *plotstr='-'*, *Plot=True*, *PrintGain=True*)

Root locus plot

Calculate the root locus by finding the roots of 1+k*TF(s) where TF is self.num(s)/self.den(s) and each k is an element of kvect.

Parameters sys : LTI object

Linear input/output systems (SISO only, for now)

kvect : list or ndarray, optional

List of gains to use in computing diagram

xlim : tuple or list, optional

control of x-axis range, normally with tuple (see matplotlib.axes)

ylim : tuple or list, optional

control of y-axis range

Plot : boolean, optional (default = True)

If True, plot magnitude and phase

PrintGain: boolean (default = True) :

If True, report mouse clicks when close to the root-locus branches, calculate gain, damping and print

Returns rlist : ndarray

Computed root locations, given as a 2d array

klist : ndarray or list

Gains used. Same as klist keyword argument if provided.

Matrix computations

care(A, B, Q[, R, S, E])	(X,L,G) = care(A,B,Q,R=None) solves the continuous-time algebraic Riccati
dare(A, B, Q, R[, S, E])	(X,L,G) = dare(A,B,Q,R) solves the discrete-time algebraic Riccati

Continued on next page

Table 3.7 – continued from previous page

`lyap`(A, Q[, C, E])	X = lyap(A,Q) solves the continuous-time Lyapunov equation
`dlyap`(A, Q[, C, E])	dlyap(A,Q) solves the discrete-time Lyapunov equation
`ctrb`(A, B)	Controllabilty matrix
`obsv`(A, C)	Observability matrix
`gram`(sys, type)	Gramian (controllability or observability)

control.care

`control.`**`care`**(*A*, *B*, *Q*, *R=None*, *S=None*, *E=None*)

(X,L,G) = care(A,B,Q,R=None) solves the continuous-time algebraic Riccati equation

$$A^T X + XA - XBR^{-1}B^T X + Q = 0$$

where A and Q are square matrices of the same dimension. Further, Q and R are a symmetric matrices. If R is None, it is set to the identity matrix. The function returns the solution X, the gain matrix G = B^T X and the closed loop eigenvalues L, i.e., the eigenvalues of A - B G.

(X,L,G) = care(A,B,Q,R,S,E) solves the generalized continuous-time algebraic Riccati equation

$$A^T XE + E^T XA - (E^T XB + S)R^{-1}(B^T XE + S^T) + Q = 0$$

where A, Q and E are square matrices of the same dimension. Further, Q and R are symmetric matrices. If R is None, it is set to the identity matrix. The function returns the solution X, the gain matrix G = R^-1 (B^T X E + S^T) and the closed loop eigenvalues L, i.e., the eigenvalues of A - B G , E.

control.dare

`control.`**`dare`**(*A*, *B*, *Q*, *R*, *S=None*, *E=None*)

(X,L,G) = dare(A,B,Q,R) solves the discrete-time algebraic Riccati equation

$$A^T XA - X - A^T XB(B^T XB + R)^{-1}B^T XA + Q = 0$$

where A and Q are square matrices of the same dimension. Further, Q is a symmetric matrix. The function returns the solution X, the gain matrix G = (B^T X B + R)^-1 B^T X A and the closed loop eigenvalues L, i.e., the eigenvalues of A - B G.

(X,L,G) = dare(A,B,Q,R,S,E) solves the generalized discrete-time algebraic Riccati equation

$$A^T XA - E^T XE - (A^T XB + S)(B^T XB + R)^{-1}(B^T XA + S^T) + Q = 0$$

where A, Q and E are square matrices of the same dimension. Further, Q and R are symmetric matrices. The function returns the solution X, the gain matrix $G = (B^T XB + R)^{-1}(B^T XA + S^T)$ and the closed loop eigenvalues L, i.e., the eigenvalues of A - B G , E.

control.lyap

`control.`**`lyap`**(*A*, *Q*, *C=None*, *E=None*)

X = lyap(A,Q) solves the continuous-time Lyapunov equation

$$AX + XA^T + Q = 0$$

where A and Q are square matrices of the same dimension. Further, Q must be symmetric.

X = lyap(A,Q,C) solves the Sylvester equation

$$AX + XQ + C = 0$$

where A and Q are square matrices.

X = lyap(A,Q,None,E) solves the generalized continuous-time Lyapunov equation

$$AXE^T + EXA^T + Q = 0$$

where Q is a symmetric matrix and A, Q and E are square matrices of the same dimension.

control.dlyap

`control.`**`dlyap`**(*A*, *Q*, *C=None*, *E=None*)

dlyap(A,Q) solves the discrete-time Lyapunov equation

$$AXA^T - X + Q = 0$$

where A and Q are square matrices of the same dimension. Further Q must be symmetric.

dlyap(A,Q,C) solves the Sylvester equation

$$AXQ^T - X + C = 0$$

where A and Q are square matrices.

dlyap(A,Q,None,E) solves the generalized discrete-time Lyapunov equation

$$AXA^T - EXE^T + Q = 0$$

where Q is a symmetric matrix and A, Q and E are square matrices of the same dimension.

control.ctrb

`control.`**`ctrb`**(*A*, *B*)

Controllabilty matrix

Parameters **A, B: array_like or string** :

Dynamics and input matrix of the system

Returns **C: matrix** :

Controllability matrix

Examples

```
>>> C = ctrb(A, B)
```

control.obsv

`control.`**`obsv`**(*A*, *C*)

Observability matrix

Parameters **A, C: array_like or string** :

Dynamics and output matrix of the system

Returns **O: matrix** :

Observability matrix

Examples

```
>>> O = obsv(A, C)
```

control.gram

control.**gram**(*sys*, *type*)

Gramian (controllability or observability)

Parameters **sys: StateSpace** :

State-space system to compute Gramian for

type: String :

Type of desired computation. *type* is either 'c' (controllability) or 'o' (observability). To compute the Cholesky factors of gramians use 'cf' (controllability) or 'of' (observability)

Returns **gram: array** :

Gramian of system

Raises **ValueError** :

- if system is not instance of StateSpace class
- if *type* is not 'c', 'o', 'cf' or 'of'
- if system is unstable (sys.A has eigenvalues not in left half plane)

ImportError :

if slycot routine sb03md cannot be found if slycot routine sb03od cannot be found

Examples

```
>>> Wc = gram(sys,'c')
>>> Wo = gram(sys,'o')
>>> Rc = gram(sys,'cf'), where Wc=Rc'*Rc
>>> Ro = gram(sys,'of'), where Wo=Ro'*Ro
```

Control system synthesis

`acker`(A, B, poles)	Pole placement using Ackermann method
`h2syn`(P, nmeas, ncon)	H_2 control synthesis for plant P.
`hinfsyn`(P, nmeas, ncon)	H_{inf} control synthesis for plant P.
`lqr`(*args, **keywords)	Linear quadratic regulator design
`place`(A, B, p)	Place closed loop eigenvalues

control.acker

`control.acker(A, B, poles)`

Pole placement using Ackermann method

Call: K = acker(A, B, poles)

Parameters **A, B** : 2-d arrays

State and input matrix of the system

poles: 1-d list :

Desired eigenvalue locations

Returns **K: matrix** :

Gains such that A - B K has given eigenvalues

control.h2syn

`control.h2syn(P, nmeas, ncon)`

H_2 control synthesis for plant P.

Parameters **P: partitioned lti plant (State-space sys)** :

nmeas: number of measurements (input to controller) :

ncon: number of control inputs (output from controller) :

Returns **K: controller to stabilize P (State-space sys)** :

Raises **ImportError** :

if slycot routine sb10hd is not loaded

See also:

StateSpace

Examples

```
>>> K = h2syn(P,nmeas,ncon)
```

control.hinfsyn

`control.hinfsyn(P, nmeas, ncon)`

H_{inf} control synthesis for plant P.

Parameters **P: partitioned lti plant** :

nmeas: number of measurements (input to controller) :

ncon: number of control inputs (output from controller) :

Returns **K: controller to stabilize P (State-space sys)** :

CL: closed loop system (State-space sys) :

gam: infinity norm of closed loop system :

rcond: 4-vector, reciprocal condition estimates of: :

1: control transformation matrix 2: measurement transformation matrix 3: X-Ricatti equation 4: Y-Ricatti equation

TODO: document significance of rcond

Raises ImportError :

if slycot routine sb10ad is not loaded

See also:

StateSpace

Examples

```
>>> K, CL, gam, rcond = hinfsyn(P,nmeas,ncon)
```

control.lqr

`control.lqr(*args, **keywords)`

Linear quadratic regulator design

The lqr() function computes the optimal state feedback controller that minimizes the quadratic cost

$$J = \int_0^\infty (x'Qx + u'Ru + 2x'Nu)dt$$

The function can be called with either 3, 4, or 5 arguments:

- `lqr(sys, Q, R)`
- `lqr(sys, Q, R, N)`
- `lqr(A, B, Q, R)`
- `lqr(A, B, Q, R, N)`

where *sys* is an *LTI* object, and *A*, *B*, *Q*, *R*, and *N* are 2d arrays or matrices of appropriate dimension.

Parameters A, B: 2-d array :

Dynamics and input matrices

sys: LTI (StateSpace or TransferFunction) :

Linear I/O system

Q, R: 2-d array :

State and input weight matrices

N: 2-d array, optional :

Cross weight matrix

Returns K: 2-d array :

State feedback gains

S: 2-d array :

Solution to Riccati equation

E: 1-d array :

Eigenvalues of the closed loop system

Examples

```
>>> K, S, E = lqr(sys, Q, R, [N])
>>> K, S, E = lqr(A, B, Q, R, [N])
```

control.place

control.**place**(*A*, *B*, *p*)

Place closed loop eigenvalues

Parameters **A** : 2-d array

Dynamics matrix

B : 2-d array

Input matrix

p : 1-d list

Desired eigenvalue locations

Returns **K** : 2-d array

Gains such that A - B K has given eigenvalues

Examples

```
>>> A = [[-1, -1], [0, 1]]
>>> B = [[0], [1]]
>>> K = place(A, B, [-2, -5])
```

Model simplification tools

`minreal`(sys[, tol, verbose])	Eliminates uncontrollable or unobservable states in state-space models or cancelling pole-zero pairs in transfer functions.
`balred`(sys, orders[, method, alpha])	Balanced reduced order model of sys of a given order.
`hsvd`(sys)	Calculate the Hankel singular values.
`modred`(sys, ELIM[, method])	Model reduction of *sys* by eliminating the states in *ELIM* using a given method.
`era`(YY, m, n, nin, nout, r)	Calculate an ERA model of order *r* based on the impulse-response data *YY*.
`markov`(Y, U, M)	Calculate the first *M* Markov parameters [D CB CAB ...] from input *U*, output *Y*.

control.minreal

`control.`**`minreal`**(*sys*, *tol=None*, *verbose=True*)

Eliminates uncontrollable or unobservable states in state-space models or cancelling pole-zero pairs in transfer functions. The output sysr has minimal order and the same response characteristics as the original model sys.

Parameters **sys: StateSpace or TransferFunction** :

Original system

tol: real :

Tolerance

verbose: bool :

Print results if True

Returns **rsys: StateSpace or TransferFunction** :

Cleaned model

control.balred

`control.`**`balred`**(*sys*, *orders*, *method='truncate'*, *alpha=None*)

Balanced reduced order model of sys of a given order. States are eliminated based on Hankel singular value. If sys has unstable modes, they are removed, the balanced realization is done on the stable part, then reinserted in accordance with the reference below.

Reference: Hsu,C.S., and Hou,D., 1991, Reducing unstable linear control systems via real Schur transformation. Electronics Letters, 27, 984-986.

Parameters **sys: StateSpace** :

Original system to reduce

orders: integer or array of integer :

Desired order of reduced order model (if a vector, returns a vector of systems)

method: string :

Method of removing states, either `'truncate'` or `'matchdc'`.

alpha: float :

Redefines the stability boundary for eigenvalues of the system matrix A. By default for continuous-time systems, alpha <= 0 defines the stability boundary for the real part of A's eigenvalues and for discrete-time systems, 0 <= alpha <= 1 defines the stability boundary for the modulus of A's eigenvalues. See SLICOT routines AB09MD and AB09ND for more information.

Returns **rsys: StateSpace** :

A reduced order model or a list of reduced order models if orders is a list

Raises **ValueError** :

- if *method* is not `'truncate'` or `'matchdc'`

ImportError :

if slycot routine ab09ad, ab09md, or ab09nd is not found

ValueError :

if there are more unstable modes than any value in orders

Examples

```
>>> rsys = balred(sys, orders, method='truncate')
```

control.hsvd

`control.`**`hsvd`**(*sys*)

Calculate the Hankel singular values.

Parameters **sys** : StateSpace

A state space system

Returns **H** : Matrix

A list of Hankel singular values

See also:

`gram`

Notes

The Hankel singular values are the singular values of the Hankel operator. In practice, we compute the square root of the eigenvalues of the matrix formed by taking the product of the observability and controllability gramians. There are other (more efficient) methods based on solving the Lyapunov equation in a particular way (more details soon).

Examples

```
>>> H = hsvd(sys)
```

control.modred

`control.`**`modred`**(*sys*, *ELIM*, *method='matchdc'*)

Model reduction of *sys* by eliminating the states in *ELIM* using a given method.

Parameters **sys: StateSpace** :

Original system to reduce

ELIM: array :

Vector of states to eliminate

method: string :

Method of removing states in *ELIM*: either `'truncate'` or `'matchdc'`.

Returns **rsys: StateSpace** :

A reduced order model

Raises ValueError :

- if *method* is not either `'matchdc'` or `'truncate'`
- if eigenvalues of *sys.A* are not all in left half plane (*sys* must be stable)

Examples

```
>>> rsys = modred(sys, ELIM, method='truncate')
```

control.era

`control.`**`era`**(*YY*, *m*, *n*, *nin*, *nout*, *r*)

Calculate an ERA model of order *r* based on the impulse-response data *YY*.

Note: This function is not implemented yet.

Parameters YY: array :

nout x *nin* dimensional impulse-response data

m: integer :

Number of rows in Hankel matrix

n: integer :

Number of columns in Hankel matrix

nin: integer :

Number of input variables

nout: integer :

Number of output variables

r: integer :

Order of model

Returns sys: StateSpace :

A reduced order model sys=ss(Ar,Br,Cr,Dr)

Examples

```
>>> rsys = era(YY, m, n, nin, nout, r)
```

control.markov

`control.`**`markov`**(*Y*, *U*, *M*)

Calculate the first *M* Markov parameters [D CB CAB ...] from input *U*, output *Y*.

Parameters Y: array_like :

Output data

U: array_like :

Input data

M: integer :

Number of Markov parameters to output

Returns H: matrix :

First M Markov parameters

Notes

Currently only works for SISO

Examples

```
>>> H = markov(Y, U, M)
```

Utility functions and conversions

`unwrap`(angle[, period])	Unwrap a phase angle to give a continuous curve
`db2mag`(db)	Convert a gain in decibels (dB) to a magnitude
`mag2db`(mag)	Convert a magnitude to decibels (dB)
`damp`(sys[, doprint])	Compute natural frequency, damping ratio, and poles of a system
`isctime`(sys[, strict])	Check to see if a system is a continuous-time system
`isdtime`(sys[, strict])	Check to see if a system is a discrete time system
`issiso`(sys[, strict])	
`issys`(obj)	Return True if an object is a system, otherwise False
`pade`(T[, n, numdeg])	Create a linear system that approximates a delay.
`sample_system`(sysc, Ts[, method, alpha])	Convert a continuous time system to discrete time
`canonical_form`(xsys[, form])	Convert a system into canonical form
`observable_form`(xsys)	Convert a system into observable canonical form
`reachable_form`(xsys)	Convert a system into reachable canonical form
`ss2tf`(*args)	Transform a state space system to a transfer function.
`ssdata`(sys)	Return state space data objects for a system
`tf2ss`(*args)	Transform a transfer function to a state space system.
`tfdata`(sys)	Return transfer function data objects for a system
`timebase`(sys[, strict])	Return the timebase for an LTI system
`timebaseEqual`(sys1, sys2)	Check to see if two systems have the same timebase

control.unwrap

`control.`**`unwrap`**(*angle*, *period=6.28*)

Unwrap a phase angle to give a continuous curve

Parameters angle : array_like

Array of angles to be unwrapped

period : float, optional

Period (defaults to *2*pi*)

Returns angle_out : array_like

Output array, with jumps of period/2 eliminated

Examples

```
>>> import numpy as np
>>> theta = [5.74, 5.97, 6.19, 0.13, 0.35, 0.57]
>>> unwrap(theta, period=2 * np.pi)
[5.74, 5.97, 6.19, 6.413185307179586, 6.633185307179586, 6.8531853071795865]
```

control.db2mag

control.**db2mag**(*db*)

Convert a gain in decibels (dB) to a magnitude

If A is magnitude,

db = 20 * log10(A)

Parameters db : float or ndarray

input value or array of values, given in decibels

Returns mag : float or ndarray

corresponding magnitudes

control.mag2db

control.**mag2db**(*mag*)

Convert a magnitude to decibels (dB)

If A is magnitude,

db = 20 * log10(A)

Parameters mag : float or ndarray

input magnitude or array of magnitudes

Returns db : float or ndarray

corresponding values in decibels

control.damp

control.**damp**(*sys*, *doprint=True*)

Compute natural frequency, damping ratio, and poles of a system

The function takes 1 or 2 parameters

Parameters **sys: LTI (StateSpace or TransferFunction)** :

A linear system object

doprint: :

if true, print table with values

Returns **wn: array** :

Natural frequencies of the poles

damping: array :

Damping values

poles: array :

Pole locations

See also:

pole

control.isctime

`control.`**`isctime`**(*sys*, *strict=False*)

Check to see if a system is a continuous-time system

Parameters **sys** : LTI system

System to be checked

strict: bool (default = False) :

If strict is True, make sure that timebase is not None

control.isdtime

`control.`**`isdtime`**(*sys*, *strict=False*)

Check to see if a system is a discrete time system

Parameters **sys** : LTI system

System to be checked

strict: bool (default = False) :

If strict is True, make sure that timebase is not None

control.issiso

`control.`**`issiso`**(*sys*, *strict=False*)

control.issys

`control.`**`issys`**(*obj*)

Return True if an object is a system, otherwise False

control.pade

`control.pade(T, n=1, numdeg=None)`

Create a linear system that approximates a delay.

Return the numerator and denominator coefficients of the Pade approximation.

Parameters **T** : number

time delay

n : positive integer

degree of denominator of approximation

numdeg: integer, or None (the default) :

If None, numerator degree equals denominator degree If >= 0, specifies degree of numerator If < 0, numerator degree is n+numdeg

Returns **num, den** : array

Polynomial coefficients of the delay model, in descending powers of s.

Notes

Based on:

1. Algorithm 11.3.1 in Golub and van Loan, "Matrix Computation" 3rd. Ed. pp. 572-574
2. M. Vajta, "Some remarks on Padé-approximations", 3rd TEMPUS-INTCOM Symposium

control.sample_system

`control.sample_system(sysc, Ts, method='zoh', alpha=None)`

Convert a continuous time system to discrete time

Creates a discrete time system from a continuous time system by sampling. Multiple methods of conversion are supported.

Parameters **sysc** : linsys

Continuous time system to be converted

Ts : real

Sampling period

method : string

Method to use for conversion: 'matched', 'tustin', 'zoh' (default)

Returns **sysd** : linsys

Discrete time system, with sampling rate Ts

Notes

See *TransferFunction.sample* and *StateSpace.sample* for further details.

Examples

```
>>> sysc = TransferFunction([1], [1, 2, 1])
>>> sysd = sample_system(sysc, 1, method='matched')
```

control.canonical_form

`control.`**`canonical_form`**(*xsys*, *form='reachable'*)
Convert a system into canonical form

Parameters **xsys** : StateSpace object

System to be transformed, with state 'x'

form : String

Canonical form for transformation. Chosen from:

- 'reachable' - reachable canonical form
- 'observable' - observable canonical form
- 'modal' - modal canonical form [not implemented]

Returns **zsys** : StateSpace object

System in desired canonical form, with state 'z'

T : matrix

Coordinate transformation matrix, z = T * x

control.observable_form

`control.`**`observable_form`**(*xsys*)
Convert a system into observable canonical form

Parameters **xsys** : StateSpace object

System to be transformed, with state *x*

Returns **zsys** : StateSpace object

System in observable canonical form, with state *z*

T : matrix

Coordinate transformation: z = T * x

control.reachable_form

`control.`**`reachable_form`**(*xsys*)
Convert a system into reachable canonical form

Parameters **xsys** : StateSpace object

System to be transformed, with state *x*

Returns **zsys** : StateSpace object

System in reachable canonical form, with state *z*

T : matrix

Coordinate transformation: z = T * x

control.ss2tf

control.**ss2tf**(**args*)

Transform a state space system to a transfer function.

The function accepts either 1 or 4 parameters:

ss2tf(sys) Convert a linear system into space system form. Always creates a new system, even if sys is already a StateSpace object.

ss2tf(A, B, C, D) Create a state space system from the matrices of its state and output equations.

For details see: *ss()*

Parameters **sys: StateSpace** :

A linear system

A: array_like or string :

System matrix

B: array_like or string :

Control matrix

C: array_like or string :

Output matrix

D: array_like or string :

Feedthrough matrix

Returns **out: TransferFunction** :

New linear system in transfer function form

Raises **ValueError** :

if matrix sizes are not self-consistent, or if an invalid number of arguments is passed in

TypeError :

if *sys* is not a StateSpace object

See also:

tf, *ss*, *tf2ss*

Examples

```
>>> A = [[1., -2], [3, -4]]
>>> B = [[5.], [7]]
>>> C = [[6., 8]]
>>> D = [[9.]]
>>> sys1 = ss2tf(A, B, C, D)
>>> sys_ss = ss(A, B, C, D)
>>> sys2 = ss2tf(sys_ss)
```

control.ssdata

`control.`**`ssdata`**`(`*sys*`)`

Return state space data objects for a system

Parameters sys: LTI (StateSpace, or TransferFunction) :

LTI system whose data will be returned

Returns (A, B, C, D): list of matrices :

State space data for the system

control.tf2ss

`control.`**`tf2ss`**`(`**args*`)`

Transform a transfer function to a state space system.

The function accepts either 1 or 2 parameters:

`tf2ss(sys)` Convert a linear system into transfer function form. Always creates a new system, even if sys is already a TransferFunction object.

`tf2ss(num, den)` Create a transfer function system from its numerator and denominator polynomial coefficients.

For details see: `tf()`

Parameters sys: LTI (StateSpace or TransferFunction) :

A linear system

num: array_like, or list of list of array_like :

Polynomial coefficients of the numerator

den: array_like, or list of list of array_like :

Polynomial coefficients of the denominator

Returns out: StateSpace :

New linear system in state space form

Raises ValueError :

if *num* and *den* have invalid or unequal dimensions, or if an invalid number of arguments is passed in

TypeError :

if *num* or *den* are of incorrect type, or if sys is not a TransferFunction object

See also:

`ss`, `tf`, `ss2tf`

Examples

```
>>> num = [[[1., 2.], [3., 4.]], [[5., 6.], [7., 8.]]]
>>> den = [[[9., 8., 7.], [6., 5., 4.]], [[3., 2., 1.], [-1., -2., -3.]]]
>>> sys1 = tf2ss(num, den)
```

```
>>> sys_tf = tf(num, den)
>>> sys2 = tf2ss(sys_tf)
```

control.tfdata

`control.`**`tfdata`**(*sys*)

Return transfer function data objects for a system

Parameters **sys: LTI (StateSpace, or TransferFunction)** :

LTI system whose data will be returned

Returns **(num, den): numerator and denominator arrays** :

Transfer function coefficients (SISO only)

control.timebase

`control.`**`timebase`**(*sys*, *strict=True*)

Return the timebase for an LTI system

dt = timebase(sys)

returns the timebase for a system 'sys'. If the strict option is set to False, dt = True will be returned as 1.

control.timebaseEqual

`control.`**`timebaseEqual`**(*sys1*, *sys2*)

Check to see if two systems have the same timebase

timebaseEqual(sys1, sys2)

returns True if the timebases for the two systems are compatible. By default, systems with timebase 'None' are compatible with either discrete or continuous timebase systems. If two systems have a discrete timebase (dt > 0) then their timebases must be equal.

CHAPTER 4

LTI system classes

The classes listed below are used to represent models of linear time-invariant (LTI) systems. They are usually created from factory functions such as `tf()` and `ss()`, so the user should normally not need to instantiate these directly.

`TransferFunction`(*args)	A class for representing transfer functions
`StateSpace`(*args)	A class for representing state-space models
`FRD`(*args, **kwargs)	A class for models defined by Frequency Response Data (FRD)

control.TransferFunction

class `control.`**`TransferFunction`**(**args*)

A class for representing transfer functions

The TransferFunction class is used to represent systems in transfer function form.

The main data members are 'num' and 'den', which are 2-D lists of arrays containing MIMO numerator and denominator coefficients. For example,

```
>>> num[2][5] = numpy.array([1., 4., 8.])
```

means that the numerator of the transfer function from the 6th input to the 3rd output is set to s^2 + 4s + 8.

Discrete-time transfer functions are implemented by using the 'dt' instance variable and setting it to something other than 'None'. If 'dt' has a non-zero value, then it must match whenever two transfer functions are combined. If 'dt' is set to True, the system will be treated as a discrete time system with unspecified sampling time.

`__init__`(**args*)

Construct a transfer function.

The default constructor is TransferFunction(num, den), where num and den are lists of lists of arrays containing polynomial coefficients. To crete a discrete time transfer funtion, use TransferFunction(num, den, dt). To call the copy constructor, call TransferFunction(sys), where sys is a TransferFunction object (continuous or discrete).

Methods

`__init__`(*args)	Construct a transfer function.
`damp`()	
`dcgain`()	Return the zero-frequency (or DC) gain
`evalfr`(omega)	Evaluate a transfer function at a single angular frequency.
`feedback`([other, sign])	Feedback interconnection between two LTI objects.
`freqresp`(omega)	Evaluate a transfer function at a list of angular frequencies.
`horner`(s)	Evaluate the systems's transfer function for a complex variable
`isctime`([strict])	Check to see if a system is a continuous-time system
`isdtime`([strict])	Check to see if a system is a discrete-time system
`issiso`()	
`minreal`([tol])	Remove cancelling pole/zero pairs from a transfer function
`pole`()	Compute the poles of a transfer function.
`returnScipySignalLTI`()	Return a list of a list of scipy.signal.lti objects.
`sample`(Ts[, method, alpha])	Convert a continuous-time system to discrete time
`zero`()	Compute the zeros of a transfer function.

dcgain()

Return the zero-frequency (or DC) gain

For a continous-time transfer function G(s), the DC gain is G(0) For a discrete-time transfer function G(z), the DC gain is G(1)

Returns gain : ndarray

The zero-frequency gain

evalfr(*omega*)

Evaluate a transfer function at a single angular frequency.

self.evalfr(omega) returns the value of the transfer function matrix with input value s = i * omega.

feedback(*other=1*, *sign=-1*)

Feedback interconnection between two LTI objects.

freqresp(*omega*)

Evaluate a transfer function at a list of angular frequencies.

mag, phase, omega = self.freqresp(omega)

reports the value of the magnitude, phase, and angular frequency of the transfer function matrix evaluated at s = i * omega, where omega is a list of angular frequencies, and is a sorted version of the input omega.

horner(*s*)

Evaluate the systems's transfer function for a complex variable

Returns a matrix of values evaluated at complex variable s.

isctime(*strict=False*)

Check to see if a system is a continuous-time system

Parameters sys : LTI system

System to be checked

strict: bool (default = False) :

If strict is True, make sure that timebase is not None

isdtime(*strict=False*)

Check to see if a system is a discrete-time system

Parameters strict: bool (default = False) :

If strict is True, make sure that timebase is not None

minreal(*tol=None*)

Remove cancelling pole/zero pairs from a transfer function

pole()

Compute the poles of a transfer function.

returnScipySignalLTI()

Return a list of a list of scipy.signal.lti objects.

For instance,

```
>>> out = tfobject.returnScipySignalLTI()
>>> out[3][5]
```

is a signal.scipy.lti object corresponding to the transfer function from the 6th input to the 4th output.

sample(*Ts*, *method='zoh'*, *alpha=None*)

Convert a continuous-time system to discrete time

Creates a discrete-time system from a continuous-time system by sampling. Multiple methods of conversion are supported.

Parameters Ts : float

Sampling period

method : {"gbt", "bilinear", "euler", "backward_diff", "zoh", "matched"}

Which method to use:

- gbt: generalized bilinear transformation
- bilinear: Tustin's approximation ("gbt" with alpha=0.5)
- euler: Euler (or forward differencing) method ("gbt" with alpha=0)
- backward_diff: Backwards differencing ("gbt" with alpha=1.0)
- zoh: zero-order hold (default)

alpha : float within [0, 1]

The generalized bilinear transformation weighting parameter, which should only be specified with method="gbt", and is ignored otherwise

Returns sysd : StateSpace system

Discrete time system, with sampling rate Ts

Notes

1. Available only for SISO systems
2. Uses the command *cont2discrete* from *scipy.signal*

Examples

```
>>> sys = TransferFunction(1, [1,1])
>>> sysd = sys.sample(0.5, method='bilinear')
```

zero()

Compute the zeros of a transfer function.

control.StateSpace

class control.**StateSpace**(**args*)

A class for representing state-space models

The StateSpace class is used to represent state-space realizations of linear time-invariant (LTI) systems:

dx/dt = A x + B u y = C x + D u

where u is the input, y is the output, and x is the state.

The main data members are the A, B, C, and D matrices. The class also keeps track of the number of states (i.e., the size of A).

Discrete-time state space system are implemented by using the 'dt' instance variable and setting it to the sampling period. If 'dt' is not None, then it must match whenever two state space systems are combined. Setting dt = 0 specifies a continuous system, while leaving dt = None means the system timebase is not specified. If 'dt' is set to True, the system will be treated as a discrete time system with unspecified sampling time.

__init__(**args*)

Construct a state space object.

The default constructor is StateSpace(A, B, C, D), where A, B, C, D are matrices or equivalent objects. To call the copy constructor, call StateSpace(sys), where sys is a StateSpace object.

Methods

__init__(*args)	Construct a state space object.
append(other)	Append a second model to the present model.
damp()	
dcgain()	Return the zero-frequency gain
evalfr(omega)	Evaluate a SS system's transfer function at a single frequency.
feedback([other, sign])	Feedback interconnection between two LTI systems.
freqresp(omega)	Evaluate the system's transfer func.
horner(s)	Evaluate the systems's transfer function for a complex variable
isctime([strict])	Check to see if a system is a continuous-time system
isdtime([strict])	Check to see if a system is a discrete-time system
issiso()	
minreal([tol])	Calculate a minimal realization, removes unobservable and
pole()	Compute the poles of a state space system.
returnScipySignalLTI()	Return a list of a list of scipy.signal.lti objects.

Continued on next page

Table 4.3 – continued from previous page

sample(Ts[, method, alpha])	Convert a continuous time system to discrete time
zero()	Compute the zeros of a state space system.

append(*other*)
Append a second model to the present model. The second model is converted to state-space if necessary, inputs and outputs are appended and their order is preserved

dcgain()
Return the zero-frequency gain

The zero-frequency gain of a continuous-time state-space system is given by:

and of a discrete-time state-space system by:

Returns gain : ndarray

An array of shape (outputs,inputs); the array will either be the zero-frequency (or DC) gain, or, if the frequency response is singular, the array will be filled with np.nan.

evalfr(*omega*)
Evaluate a SS system's transfer function at a single frequency.

self.evalfr(omega) returns the value of the transfer function matrix with input value s = i * omega.

feedback(*other=1*, *sign=-1*)
Feedback interconnection between two LTI systems.

freqresp(*omega*)
Evaluate the system's transfer func. at a list of ang. frequencies.

mag, phase, omega = self.freqresp(omega)

reports the value of the magnitude, phase, and angular frequency of the system's transfer function matrix evaluated at s = i * omega, where omega is a list of angular frequencies, and is a sorted version of the input omega.

horner(*s*)
Evaluate the systems's transfer function for a complex variable

Returns a matrix of values evaluated at complex variable s.

isctime(*strict=False*)
Check to see if a system is a continuous-time system

Parameters sys : LTI system

System to be checked

strict: bool (default = False) :

If strict is True, make sure that timebase is not None

isdtime(*strict=False*)
Check to see if a system is a discrete-time system

Parameters strict: bool (default = False) :

If strict is True, make sure that timebase is not None

minreal(*tol=0.0*)
Calculate a minimal realization, removes unobservable and uncontrollable states

pole()
Compute the poles of a state space system.

returnScipySignalLTI()

Return a list of a list of scipy.signal.lti objects.

For instance,

```
>>> out = ssobject.returnScipySignalLTI()
>>> out[3][5]
```

is a signal.scipy.lti object corresponding to the transfer function from the 6th input to the 4th output.

sample(*Ts*, *method='zoh'*, *alpha=None*)

Convert a continuous time system to discrete time

Creates a discrete-time system from a continuous-time system by sampling. Multiple methods of conversion are supported.

Parameters **Ts** : float

Sampling period

method : {"gbt", "bilinear", "euler", "backward_diff", "zoh"}

Which method to use:

- gbt: generalized bilinear transformation
- bilinear: Tustin's approximation ("gbt" with alpha=0.5)
- euler: Euler (or forward differencing) method ("gbt" with alpha=0)
- backward_diff: Backwards differencing ("gbt" with alpha=1.0)
- zoh: zero-order hold (default)

alpha : float within [0, 1]

The generalized bilinear transformation weighting parameter, which should only be specified with method="gbt", and is ignored otherwise

Returns **sysd** : StateSpace system

Discrete time system, with sampling rate Ts

Notes

Uses the command 'cont2discrete' from scipy.signal

Examples

```
>>> sys = StateSpace(0, 1, 1, 0)
>>> sysd = sys.sample(0.5, method='bilinear')
```

zero()

Compute the zeros of a state space system.

control.FRD

class control.**FRD**(**args*, ***kwargs*)

A class for models defined by Frequency Response Data (FRD)

The FRD class is used to represent systems in frequency response data form.

The main data members are 'omega' and 'fresp', where *omega* is a 1D array with the frequency points of the response, and *fresp* is a 3D array, with the first dimension corresponding to the output index of the FRD, the second dimension corresponding to the input index, and the 3rd dimension corresponding to the frequency points in omega. For example,

```
>>> frdata[2,5,:] = numpy.array([1., 0.8-0.2j, 0.2-0.8j])
```

means that the frequency response from the 6th input to the 3rd output at the frequencies defined in omega is set to the array above, i.e. the rows represent the outputs and the columns represent the inputs.

`__init__`(**args*, ***kwargs*)

Construct an FRD object

The default constructor is FRD(d, w), where w is an iterable of frequency points, and d is the matching frequency data.

If d is a single list, 1d array, or tuple, a SISO system description is assumed. d can also be

To call the copy constructor, call FRD(sys), where sys is a FRD object.

To construct frequency response data for an existing LTI object, other than an FRD, call FRD(sys, omega)

Methods

`__init__`(*args, **kwargs)	Construct an FRD object
`damp`()	
`dcgain`()	Return the zero-frequency gain
`evalfr`(omega)	Evaluate a transfer function at a single angular frequency.
`feedback`([other, sign])	Feedback interconnection between two FRD objects.
`freqresp`(omega)	Evaluate a transfer function at a list of angular frequencies.
`isctime`([strict])	Check to see if a system is a continuous-time system
`isdtime`([strict])	Check to see if a system is a discrete-time system
`issiso`()	

Attributes

`epsw`

`dcgain`()

Return the zero-frequency gain

`evalfr`(*omega*)

Evaluate a transfer function at a single angular frequency.

self.evalfr(omega) returns the value of the frequency response at frequency omega.

Note that a "normal" FRD only returns values for which there is an entry in the omega vector. An interpolating FRD can return intermediate values.

`feedback`(*other=1*, *sign=-1*)

Feedback interconnection between two FRD objects.

`freqresp`(*omega*)

Evaluate a transfer function at a list of angular frequencies.

mag, phase, omega = self.freqresp(omega)

reports the value of the magnitude, phase, and angular frequency of the transfer function matrix evaluated at s = i * omega, where omega is a list of angular frequencies, and is a sorted version of the input omega.

`isctime`(*strict=False*)

Check to see if a system is a continuous-time system

Parameters **sys** : LTI system

System to be checked

strict: bool (default = False) :

If strict is True, make sure that timebase is not None

`isdtime`(*strict=False*)

Check to see if a system is a discrete-time system

Parameters **strict: bool (default = False)** :

If strict is True, make sure that timebase is not None

CHAPTER 5

MATLAB compatibility module

The `control.matlab` module contains a number of functions that emulate some of the functionality of MATLAB. The intent of these functions is to provide a simple interface to the python control systems library (python-control) for people who are familiar with the MATLAB Control Systems Toolbox (tm).

Creating linear models

`tf`(*args)	Create a transfer function system.
`ss`(*args)	Create a state space system.
`frd`(*args)	Construct a Frequency Response Data model, or convert a system
`rss`([states, outputs, inputs])	Create a stable **continuous** random state space object.
`drss`([states, outputs, inputs])	Create a stable **discrete** random state space object.

control.matlab.tf

control.matlab.**tf**(**args*)

Create a transfer function system. Can create MIMO systems.

The function accepts either 1 or 2 parameters:

`tf(sys)` Convert a linear system into transfer function form. Always creates a new system, even if sys is already a TransferFunction object.

`tf(num, den)` Create a transfer function system from its numerator and denominator polynomial coefficients.

If *num* and *den* are 1D array_like objects, the function creates a SISO system.

To create a MIMO system, *num* and *den* need to be 2D nested lists of array_like objects. (A 3 dimensional data structure in total.) (For details see note below.)

tf(num, den, dt) Create a discrete time transfer function system; dt can either be a positive number indicating the sampling time or 'True' if no specific timebase is given.

Parameters sys: LTI (StateSpace or TransferFunction) :

A linear system

num: array_like, or list of list of array_like :

Polynomial coefficients of the numerator

den: array_like, or list of list of array_like :

Polynomial coefficients of the denominator

Returns out: :class:'TransferFunction' :

The new linear system

Raises ValueError :

if *num* and *den* have invalid or unequal dimensions

TypeError :

if *num* or *den* are of incorrect type

See also:

`ss`, `ss2tf`, `tf2ss`

Notes

`num[i][j]` contains the polynomial coefficients of the numerator for the transfer function from the (j+1)st input to the (i+1)st output. `den[i][j]` works the same way.

The list `[2, 3, 4]` denotes the polynomial $2s^2 + 3s + 4$.

Examples

```
>>> # Create a MIMO transfer function object
>>> # The transfer function from the 2nd input to the 1st output is
>>> # (3s + 4) / (6s^2 + 5s + 4).
>>> num = [[[1., 2.], [3., 4.]], [[5., 6.], [7., 8.]]]
>>> den = [[[9., 8., 7.], [6., 5., 4.]], [[3., 2., 1.], [-1., -2., -3.]]]
>>> sys1 = tf(num, den)
```

```
>>> # Convert a StateSpace to a TransferFunction object.
>>> sys_ss = ss("1. -2; 3. -4", "5.; 7", "6. 8", "9.")
>>> sys2 = tf(sys1)
```

control.matlab.ss

`control.matlab.`**`ss`**`(*args)`

Create a state space system.

The function accepts either 1, 4 or 5 parameters:

ss(sys) Convert a linear system into space system form. Always creates a new system, even if sys is already a StateSpace object.

ss(A, B, C, D) Create a state space system from the matrices of its state and output equations:

$$\dot{x} = A \cdot x + B \cdot u$$
$$y = C \cdot x + D \cdot u$$

ss(A, B, C, D, dt) Create a discrete-time state space system from the matrices of its state and output equations:

$$x[k+1] = A \cdot x[k] + B \cdot u[k]$$
$$y[k] = C \cdot x[k] + D \cdot u[ki]$$

The matrices can be given as *array like* data types or strings. Everything that the constructor of `numpy.matrix` accepts is permissible here too.

Parameters **sys: StateSpace or TransferFunction** :

A linear system

A: array_like or string :

System matrix

B: array_like or string :

Control matrix

C: array_like or string :

Output matrix

D: array_like or string :

Feed forward matrix

dt: If present, specifies the sampling period and a discrete time :

system is created

Returns **out: :class:'StateSpace'** :

The new linear system

Raises **ValueError** :

if matrix sizes are not self-consistent

See also:

tf, *ss2tf*, *tf2ss*

Examples

```
>>> # Create a StateSpace object from four "matrices".
>>> sys1 = ss("1. -2; 3. -4", "5.; 7", "6. 8", "9.")
```

```
>>> # Convert a TransferFunction to a StateSpace object.
>>> sys_tf = tf([2.], [1., 3])
>>> sys2 = ss(sys_tf)
```

control.matlab.frd

`control.matlab.`**`frd`**`(`**args*`)`

Construct a Frequency Response Data model, or convert a system

frd models store the (measured) frequency response of a system.

This function can be called in different ways:

`frd(response, freqs)` Create an frd model with the given response data, in the form of complex response vector, at matching frequency freqs [in rad/s]

`frd(sys, freqs)` Convert an LTI system into an frd model with data at frequencies freqs.

Parameters **response: array_like, or list** :

complex vector with the system response

freq: array_lik or lis :

vector with frequencies

sys: LTI (StateSpace or TransferFunction) :

A linear system

Returns **sys: FRD** :

New frequency response system

See also:

`ss`, *`tf`*

control.matlab.rss

`control.matlab.`**`rss`**`(`*states=1*, *outputs=1*, *inputs=1*`)`

Create a stable **continuous** random state space object.

Parameters **states: integer** :

Number of state variables

inputs: integer :

Number of system inputs

outputs: integer :

Number of system outputs

Returns **sys: StateSpace** :

The randomly created linear system

Raises **ValueError** :

if any input is not a positive integer

See also:

`drss`

Notes

If the number of states, inputs, or outputs is not specified, then the missing numbers are assumed to be 1. The poles of the returned system will always have a negative real part.

control.matlab.drss

`control.matlab.`**`drss`**`(`*states=1*, *outputs=1*, *inputs=1*`)`

Create a stable **discrete** random state space object.

Parameters states: integer :

Number of state variables

inputs: integer :

Number of system inputs

outputs: integer :

Number of system outputs

Returns sys: StateSpace :

The randomly created linear system

Raises ValueError :

if any input is not a positive integer

See also:

`rss`

Notes

If the number of states, inputs, or outputs is not specified, then the missing numbers are assumed to be 1. The poles of the returned system will always have a magnitude less than 1.

Utility functions and conversions

`mag2db`(mag)	Convert a magnitude to decibels (dB)
`db2mag`(db)	Convert a gain in decibels (dB) to a magnitude
`c2d`(sysc, Ts[, method])	Return a discrete-time system
`ss2tf`(*args)	Transform a state space system to a transfer function.
`tf2ss`(*args)	Transform a transfer function to a state space system.
`tfdata`(sys)	Return transfer function data objects for a system

control.matlab.mag2db

`control.matlab.`**`mag2db`**`(`*mag*`)`

Convert a magnitude to decibels (dB)

If A is magnitude,

db = 20 * log10(A)

Parameters **mag** : float or ndarray

input magnitude or array of magnitudes

Returns **db** : float or ndarray

corresponding values in decibels

control.matlab.db2mag

`control.matlab.db2mag(db)`

Convert a gain in decibels (dB) to a magnitude

If A is magnitude,

db = 20 * log10(A)

Parameters **db** : float or ndarray

input value or array of values, given in decibels

Returns **mag** : float or ndarray

corresponding magnitudes

control.matlab.c2d

`control.matlab.c2d(sysc, Ts, method='zoh')`

Return a discrete-time system

Parameters **sysc: LTI (StateSpace or TransferFunction), continuous** :

System to be converted

Ts: number :

Sample time for the conversion

method: string, optional :

Method to be applied, 'zoh' Zero-order hold on the inputs (default) 'foh' First-order hold, currently not implemented 'impulse' Impulse-invariant discretization, currently not implemented 'tustin' Bilinear (Tustin) approximation, only SISO 'matched' Matched pole-zero method, only SISO

control.matlab.ss2tf

`control.matlab.ss2tf(*args)`

Transform a state space system to a transfer function.

The function accepts either 1 or 4 parameters:

`ss2tf(sys)` Convert a linear system into space system form. Always creates a new system, even if sys is already a StateSpace object.

`ss2tf(A, B, C, D)` Create a state space system from the matrices of its state and output equations.

For details see: `ss()`

Parameters sys: StateSpace :

A linear system

A: array_like or string :

System matrix

B: array_like or string :

Control matrix

C: array_like or string :

Output matrix

D: array_like or string :

Feedthrough matrix

Returns out: TransferFunction :

New linear system in transfer function form

Raises ValueError :

if matrix sizes are not self-consistent, or if an invalid number of arguments is passed in

TypeError :

if *sys* is not a StateSpace object

See also:

`tf`, `ss`, `tf2ss`

Examples

```
>>> A = [[1., -2], [3, -4]]
>>> B = [[5.], [7]]
>>> C = [[6., 8]]
>>> D = [[9.]]
>>> sys1 = ss2tf(A, B, C, D)
```

```
>>> sys_ss = ss(A, B, C, D)
>>> sys2 = ss2tf(sys_ss)
```

control.matlab.tf2ss

`control.matlab.`**`tf2ss`**`(*args)`

Transform a transfer function to a state space system.

The function accepts either 1 or 2 parameters:

`tf2ss(sys)` Convert a linear system into transfer function form. Always creates a new system, even if sys is already a TransferFunction object.

`tf2ss(num, den)` Create a transfer function system from its numerator and denominator polynomial coefficients.

For details see: `tf()`

Parameters sys: LTI (StateSpace or TransferFunction) :

A linear system

num: array_like, or list of list of array_like :

Polynomial coefficients of the numerator

den: array_like, or list of list of array_like :

Polynomial coefficients of the denominator

Returns out: StateSpace :

New linear system in state space form

Raises ValueError :

if *num* and *den* have invalid or unequal dimensions, or if an invalid number of arguments is passed in

TypeError :

if *num* or *den* are of incorrect type, or if sys is not a TransferFunction object

See also:

`ss`, `tf`, `ss2tf`

Examples

```
>>> num = [[[1., 2.], [3., 4.]], [[5., 6.], [7., 8.]]]
>>> den = [[[9., 8., 7.], [6., 5., 4.]], [[3., 2., 1.], [-1., -2., -3.]]]
>>> sys1 = tf2ss(num, den)
```

```
>>> sys_tf = tf(num, den)
>>> sys2 = tf2ss(sys_tf)
```

control.matlab.tfdata

`control.matlab.`**`tfdata`**(*sys*)

Return transfer function data objects for a system

Parameters sys: LTI (StateSpace, or TransferFunction) :

LTI system whose data will be returned

Returns (num, den): numerator and denominator arrays :

Transfer function coefficients (SISO only)

System interconnections

`series`(sys1, sys2)	Return the series connection sys2 * sys1 for –> sys1 –> sys2 –>.
`parallel`(sys1, sys2)	Return the parallel connection sys1 + sys2.

Continued on next page

Table 5.3 – continued from previous page

`feedback`(sys1[, sys2, sign])	Feedback interconnection between two I/O systems.
`negate`(sys)	Return the negative of a system.
`connect`(sys, Q, inputv, outputv)	Index-base interconnection of system
`append`(*sys)	Group models by appending their inputs and outputs

control.matlab.series

`control.matlab.series(sys1, sys2)`

Return the series connection sys2 * sys1 for –> sys1 –> sys2 –>.

Parameters sys1: scalar, StateSpace, TransferFunction, or FRD :

sys2: scalar, StateSpace, TransferFunction, or FRD :

Returns out: scalar, StateSpace, or TransferFunction :

Raises ValueError :

if *sys2.inputs* does not equal *sys1.outputs* if *sys1.dt* is not compatible with *sys2.dt*

See also:

`parallel`, `feedback`

Notes

This function is a wrapper for the __mul__ function in the StateSpace and TransferFunction classes. The output type is usually the type of *sys2*. If *sys2* is a scalar, then the output type is the type of *sys1*.

If both systems have a defined timebase (dt = 0 for continuous time, dt > 0 for discrete time), then the timebase for both systems must match. If only one of the system has a timebase, the return timebase will be set to match it.

Examples

```
>>> sys3 = series(sys1, sys2) # Same as sys3 = sys2 * sys1.
```

control.matlab.parallel

`control.matlab.parallel(sys1, sys2)`

Return the parallel connection sys1 + sys2.

Parameters sys1: scalar, StateSpace, TransferFunction, or FRD :

sys2: scalar, StateSpace, TransferFunction, or FRD :

Returns out: scalar, StateSpace, or TransferFunction :

Raises ValueError :

if *sys1* and *sys2* do not have the same numbers of inputs and outputs

See also:

`series`, `feedback`

Notes

This function is a wrapper for the __add__ function in the StateSpace and TransferFunction classes. The output type is usually the type of *sys1*. If *sys1* is a scalar, then the output type is the type of *sys2*.

If both systems have a defined timebase (dt = 0 for continuous time, dt > 0 for discrete time), then the timebase for both systems must match. If only one of the system has a timebase, the return timebase will be set to match it.

Examples

```
>>> sys3 = parallel(sys1, sys2) # Same as sys3 = sys1 + sys2.
```

control.matlab.feedback

`control.matlab.`**`feedback`**(*sys1*, *sys2=1*, *sign=-1*)

Feedback interconnection between two I/O systems.

Parameters sys1: scalar, StateSpace, TransferFunction, FRD :

The primary plant.

sys2: scalar, StateSpace, TransferFunction, FRD :

The feedback plant (often a feedback controller).

sign: scalar :

The sign of feedback. *sign* = -1 indicates negative feedback, and *sign* = 1 indicates positive feedback. *sign* is an optional argument; it assumes a value of -1 if not specified.

Returns out: StateSpace or TransferFunction :

Raises ValueError :

if *sys1* does not have as many inputs as *sys2* has outputs, or if *sys2* does not have as many inputs as *sys1* has outputs

NotImplementedError :

if an attempt is made to perform a feedback on a MIMO TransferFunction object

See also:

`series`, *`parallel`*

Notes

This function is a wrapper for the feedback function in the StateSpace and TransferFunction classes. It calls TransferFunction.feedback if *sys1* is a TransferFunction object, and StateSpace.feedback if *sys1* is a StateSpace object. If *sys1* is a scalar, then it is converted to *sys2*'s type, and the corresponding feedback function is used. If *sys1* and *sys2* are both scalars, then TransferFunction.feedback is used.

control.matlab.negate

`control.matlab.negate(sys)`

Return the negative of a system.

Parameters sys: StateSpace, TransferFunction or FRD :

Returns out: StateSpace or TransferFunction :

Notes

This function is a wrapper for the __neg__ function in the StateSpace and TransferFunction classes. The output type is the same as the input type.

If both systems have a defined timebase (dt = 0 for continuous time, dt > 0 for discrete time), then the timebase for both systems must match. If only one of the system has a timebase, the return timebase will be set to match it.

Examples

```
>>> sys2 = negate(sys1) # Same as sys2 = -sys1.
```

control.matlab.connect

`control.matlab.connect(sys, Q, inputv, outputv)`

Index-base interconnection of system

The system sys is a system typically constructed with append, with multiple inputs and outputs. The inputs and outputs are connected according to the interconnection matrix Q, and then the final inputs and outputs are trimmed according to the inputs and outputs listed in inputv and outputv.

Note: to have this work, inputs and outputs start counting at 1!!!!

Parameters sys: StateSpace Transferfunction :

System to be connected

Q: 2d array :

Interconnection matrix. First column gives the input to be connected second column gives the output to be fed into this input. Negative values for the second column mean the feedback is negative, 0 means no connection is made

inputv: 1d array :

list of final external inputs

outputv: 1d array :

list of final external outputs

Returns sys: LTI system :

Connected and trimmed LTI system

Examples

```
>>> sys1 = ss("1. -2; 3. -4", "5.; 7", "6, 8", "9.")
>>> sys2 = ss("-1.", "1.", "1.", "0.")
>>> sys = append(sys1, sys2)
>>> Q = sp.mat([ [ 1, 2], [2, -1] ]) # basically feedback, output 2 in 1
>>> sysc = connect(sys, Q, [2], [1, 2])
```

control.matlab.append

`control.matlab.`**`append`**`(`*`*sys`*`)`

Group models by appending their inputs and outputs

Forms an augmented system model, and appends the inputs and outputs together. The system type will be the type of the first system given; if you mix state-space systems and gain matrices, make sure the gain matrices are not first.

Parameters **sys1, sys2, ... sysn: StateSpace or Transferfunction** :

LTI systems to combine

Returns **sys: LTI system** :

Combined LTI system, with input/output vectors consisting of all input/output vectors appended

Examples

```
>>> sys1 = ss("1. -2; 3. -4", "5.; 7", "6. 8", "9.")
>>> sys2 = ss("-1.", "1.", "1.", "0.")
>>> sys = append(sys1, sys2)
```

Todo

also implement for transfer function, zpk, etc.

System gain and dynamics

dcgain(*args)	Compute the gain of the system in steady state.
pole(sys)	Compute system poles.
zero(sys)	Compute system zeros.
damp(sys[, doprint])	Compute natural frequency, damping ratio, and poles of a system
pzmap(sys[, Plot, title])	Plot a pole/zero map for a linear system.

control.matlab.dcgain

`control.matlab.`**`dcgain`**`(`*`*args`*`)`

Compute the gain of the system in steady state.

The function takes either 1, 2, 3, or 4 parameters:

Parameters **A, B, C, D: array-like** :

A linear system in state space form.

Z, P, k: array-like, array-like, number :

A linear system in zero, pole, gain form.

num, den: array-like :

A linear system in transfer function form.

sys: LTI (StateSpace or TransferFunction) :

A linear system object.

Returns **gain: ndarray** :

The gain of each output versus each input: $y = gain \cdot u$

Notes

This function is only useful for systems with invertible system matrix `A`.

All systems are first converted to state space form. The function then computes:

$$gain = -C \cdot A^{-1} \cdot B + D$$

control.matlab.pole

`control.matlab.`**`pole`**`(`*sys*`)`

Compute system poles.

Parameters **sys: StateSpace or TransferFunction** :

Linear system

Returns **poles: ndarray** :

Array that contains the system's poles.

Raises **NotImplementedError** :

when called on a TransferFunction object

See also:

`zero`, `TransferFunction.pole`, `StateSpace.pole`

control.matlab.zero

`control.matlab.`**`zero`**`(`*sys*`)`

Compute system zeros.

Parameters **sys: StateSpace or TransferFunction** :

Linear system

Returns **zeros: ndarray** :

Array that contains the system's zeros.

Raises NotImplementedError :

when called on a MIMO system

See also:

pole, `StateSpace.zero`, `TransferFunction.zero`

control.matlab.damp

`control.matlab.`**`damp`**(*sys*, *doprint=True*)

Compute natural frequency, damping ratio, and poles of a system

The function takes 1 or 2 parameters

Parameters sys: LTI (StateSpace or TransferFunction) :

A linear system object

doprint: :

if true, print table with values

Returns wn: array :

Natural frequencies of the poles

damping: array :

Damping values

poles: array :

Pole locations

See also:

pole

control.matlab.pzmap

`control.matlab.`**`pzmap`**(*sys*, *Plot=True*, *title='Pole Zero Map'*)

Plot a pole/zero map for a linear system.

Parameters sys: LTI (StateSpace or TransferFunction) :

Linear system for which poles and zeros are computed.

Plot: bool :

If `True` a graph is generated with Matplotlib, otherwise the poles and zeros are only computed and returned.

Returns pole: array :

The systems poles

zeros: array :

The system's zeros.

Time-domain analysis

step(sys[, T, X0, input, output, return_x])	Step response of a linear system
impulse(sys[, T, X0, input, output, return_x])	Impulse response of a linear system
initial(sys[, T, X0, input, output, return_x])	Initial condition response of a linear system
lsim(sys[, U, T, X0])	Simulate the output of a linear system.

control.matlab.step

control.matlab.**step**(*sys*, *T=None*, *X0=0.0*, *input=0*, *output=None*, *return_x=False*)

Step response of a linear system

If the system has multiple inputs or outputs (MIMO), one input has to be selected for the simulation. Optionally, one output may be selected. If no selection is made for the output, all outputs are given. The parameters *input* and *output* do this. All other inputs are set to 0, all other outputs are ignored.

Parameters sys: StateSpace, or TransferFunction :

LTI system to simulate

T: array-like object, optional :

Time vector (argument is autocomputed if not given)

X0: array-like or number, optional :

Initial condition (default = 0)

Numbers are converted to constant arrays with the correct shape.

input: int :

Index of the input that will be used in this simulation.

output: int :

If given, index of the output that is returned by this simulation.

Returns yout: array :

Response of the system

T: array :

Time values of the output

xout: array (if selected) :

Individual response of each x variable

See also:

lsim, *initial*, *impulse*

Examples

```
>>> yout, T = step(sys, T, X0)
```

control.matlab.impulse

`control.matlab.`**`impulse`**(*sys*, *T=None*, *X0=0.0*, *input=0*, *output=None*, *return_x=False*)

Impulse response of a linear system

If the system has multiple inputs or outputs (MIMO), one input has to be selected for the simulation. Optionally, one output may be selected. If no selection is made for the output, all outputs are given. The parameters *input* and *output* do this. All other inputs are set to 0, all other outputs are ignored.

Parameters **sys: StateSpace, TransferFunction** :

LTI system to simulate

T: array-like object, optional :

Time vector (argument is autocomputed if not given)

X0: array-like or number, optional :

Initial condition (default = 0)

Numbers are converted to constant arrays with the correct shape.

input: int :

Index of the input that will be used in this simulation.

output: int :

Index of the output that will be used in this simulation.

Returns **yout: array** :

Response of the system

T: array :

Time values of the output

xout: array (if selected) :

Individual response of each x variable

See also:

`lsim`, *`step`*, *`initial`*

Examples

```
>>> yout, T = impulse(sys, T)
```

control.matlab.initial

`control.matlab.`**`initial`**(*sys*, *T=None*, *X0=0.0*, *input=None*, *output=None*, *return_x=False*)

Initial condition response of a linear system

If the system has multiple outputs (?IMO), optionally, one output may be selected. If no selection is made for the output, all outputs are given.

Parameters **sys: StateSpace, or TransferFunction** :

LTI system to simulate

T: array-like object, optional :

Time vector (argument is autocomputed if not given)

X0: array-like object or number, optional :

Initial condition (default = 0)

Numbers are converted to constant arrays with the correct shape.

input: int :

This input is ignored, but present for compatibility with step and impulse.

output: int :

If given, index of the output that is returned by this simulation.

Returns yout: array :

Response of the system

T: array :

Time values of the output

xout: array (if selected) :

Individual response of each x variable

See also:

lsim, *step*, *impulse*

Examples

```
>>> yout, T = initial(sys, T, X0)
```

control.matlab.lsim

`control.matlab.lsim`(*sys*, *U=0.0*, *T=None*, *X0=0.0*)

Simulate the output of a linear system.

As a convenience for parameters *U*, *X0*: Numbers (scalars) are converted to constant arrays with the correct shape. The correct shape is inferred from arguments *sys* and *T*.

Parameters sys: LTI (StateSpace, or TransferFunction) :

LTI system to simulate

U: array-like or number, optional :

Input array giving input at each time *T* (default = 0).

If *U* is `None` or `0`, a special algorithm is used. This special algorithm is faster than the general algorithm, which is used otherwise.

T: array-like :

Time steps at which the input is defined, numbers must be (strictly monotonic) increasing.

X0: array-like or number, optional :

Initial condition (default = 0).

Returns yout: array :

Response of the system.

T: array :

Time values of the output.

xout: array :

Time evolution of the state vector.

See also:

`step`, `initial`, `impulse`

Examples

```
>>> yout, T, xout = lsim(sys, U, T, X0)
```

Frequency-domain analysis

`bode`(*args, **keywords)	Bode plot of the frequency response
`nyquist`(syslist[, omega, Plot, color, labelFreq])	Nyquist plot for a system
`nichols`(syslist[, omega, grid])	Nichols plot for a system
`margin`(*args)	Calculate gain and phase margins and associated crossover frequencies
`freqresp`(sys, omega)	Frequency response of an LTI system at multiple angular frequencies.
`evalfr`(sys, x)	Evaluate the transfer function of an LTI system for a single complex number x.

control.matlab.bode

`control.matlab.`**`bode`**`(`*`*args`*`, `*`**keywords`*`)`

Bode plot of the frequency response

Plots a bode gain and phase diagram

Parameters sys : LTI, or list of LTI

System for which the Bode response is plotted and give. Optionally a list of systems can be entered, or several systems can be specified (i.e. several parameters). The sys arguments may also be interspersed with format strings. A frequency argument (array_like) may also be added, some examples: * >>> bode(sys, w) # one system, freq vector * >>> bode(sys1, sys2, ..., sysN) # several systems * >>> bode(sys1, sys2, ..., sysN, w) * >>> bode(sys1, 'plotstyle1', ..., sysN, 'plotstyleN') # + plot formats

omega: freq_range :

Range of frequencies in rad/s

dB : boolean

If True, plot result in dB

Hz : boolean

If True, plot frequency in Hz (omega must be provided in rad/sec)

deg : boolean

If True, return phase in degrees (else radians)

Plot : boolean

If True, plot magnitude and phase

Examples

```
>>> sys = ss("1. -2; 3. -4", "5.; 7", "6. 8", "9.")
>>> mag, phase, omega = bode(sys)
```

Todo

Document these use cases

-
  ```
  >>> bode(sys, w)
  ```

-
  ```
  >>> bode(sys1, sys2, ..., sysN)
  ```

-
  ```
  >>> bode(sys1, sys2, ..., sysN, w)
  ```

-
  ```
  >>> bode(sys1, 'plotstyle1', ..., sysN, 'plotstyleN')
  ```

control.matlab.nyquist

`control.matlab.`**`nyquist`**(*syslist*, *omega=None*, *Plot=True*, *color='b'*, *labelFreq=0*, **args*, ***kwargs*)

Nyquist plot for a system

Plots a Nyquist plot for the system over a (optional) frequency range.

Parameters **syslist** : list of LTI

List of linear input/output systems (single system is OK)

omega : freq_range

Range of frequencies (list or bounds) in rad/sec

Plot : boolean

If True, plot magnitude

labelFreq : int

Label every nth frequency on the plot

***args, **kwargs:** :

Additional options to matplotlib (color, linestyle, etc)

Returns **real** : array

real part of the frequency response array

imag : array

imaginary part of the frequency response array

freq : array

frequencies

Examples

```
>>> sys = ss("1. -2; 3. -4", "5.; 7", "6. 8", "9.")
>>> real, imag, freq = nyquist_plot(sys)
```

control.matlab.nichols

`control.matlab.`**`nichols`**`(`*syslist*, *omega=None*, *grid=True*`)`

Nichols plot for a system

Plots a Nichols plot for the system over a (optional) frequency range.

Parameters **syslist** : list of LTI, or LTI

List of linear input/output systems (single system is OK)

omega : array_like

Range of frequencies (list or bounds) in rad/sec

grid : boolean, optional

True if the plot should include a Nichols-chart grid. Default is True.

Returns **None** :

control.matlab.margin

`control.matlab.`**`margin`**`(`**args*`)`

Calculate gain and phase margins and associated crossover frequencies

Parameters **sysdata: LTI system or (mag, phase, omega) sequence** :

sys [StateSpace or TransferFunction] Linear SISO system

mag, phase, omega [sequence of array_like] Input magnitude, phase (in deg.), and frequencies (rad/sec) from bode frequency response data

Returns **gm** : float

Gain margin

pm [float] Phase margin (in degrees)

Wcg [float] Gain crossover frequency (corresponding to phase margin)

Wcp [float] Phase crossover frequency (corresponding to gain margin) (in rad/sec)

Margins are of SISO open-loop. If more than one crossover frequency is :

detected, returns the lowest corresponding margin. :

Examples

```
>>> sys = tf(1, [1, 2, 1, 0])
>>> gm, pm, Wcg, Wcp = margin(sys)
```

control.matlab.freqresp

`control.matlab.freqresp(sys, omega)`

Frequency response of an LTI system at multiple angular frequencies.

Parameters **sys: StateSpace or TransferFunction** :

Linear system

omega: array_like :

List of frequencies

Returns **mag: ndarray** :

phase: ndarray :

omega: list, tuple, or ndarray :

See also:

evalfr, *bode*

Notes

This function is a wrapper for StateSpace.freqresp and TransferFunction.freqresp. The output omega is a sorted version of the input omega.

Examples

```
>>> sys = ss("1. -2; 3. -4", "5.; 7", "6. 8", "9.")
>>> mag, phase, omega = freqresp(sys, [0.1, 1., 10.])
>>> mag
array([[[ 58.8576682 ,  49.64876635,  13.40825927]]])
>>> phase
array([[[-0.05408304, -0.44563154, -0.66837155]]])
```

Todo

Add example with MIMO system

#>>> sys = rss(3, 2, 2) #>>> mag, phase, omega = freqresp(sys, [0.1, 1., 10.]) #>>> mag[0, 1, :] #array([55.43747231, 42.47766549, 1.97225895]) #>>> phase[1, 0, :] #array([-0.12611087, -1.14294316, 2.5764547]) #>>> # This is the magnitude of the frequency response from the 2nd #>>> # input to the 1st output, and the phase (in radians) of the #>>> # frequency response from the 1st input to the 2nd output, for #>>> # s = 0.1i, i, 10i.

control.matlab.evalfr

control.matlab.**evalfr**(*sys*, *x*)

Evaluate the transfer function of an LTI system for a single complex number x.

To evaluate at a frequency, enter x = omega*j, where omega is the frequency in radians

Parameters **sys: StateSpace or TransferFunction** :

Linear system

x: scalar :

Complex number

Returns **fresp: ndarray** :

See also:

freqresp, *bode*

Notes

This function is a wrapper for StateSpace.evalfr and TransferFunction.evalfr.

Examples

```
>>> sys = ss("1. -2; 3. -4", "5.; 7", "6. 8", "9.")
>>> evalfr(sys, 1j)
array([[ 44.8-21.4j]])
>>> # This is the transfer function matrix evaluated at s = i.
```

Todo

Add example with MIMO system

Compensator design

rlocus(sys[, kvect, xlim, ylim, plotstr, ...])	Root locus plot
place(A, B, p)	Place closed loop eigenvalues
lqr(*args, **keywords)	Linear quadratic regulator design

control.matlab.rlocus

control.matlab.**rlocus**(*sys*, *kvect=None*, *xlim=None*, *ylim=None*, *plotstr='-'*, *Plot=True*, *PrintGain=True*)

Root locus plot

Calculate the root locus by finding the roots of 1+k*TF(s) where TF is self.num(s)/self.den(s) and each k is an element of kvect.

Parameters **sys** : LTI object

Linear input/output systems (SISO only, for now)

kvect : list or ndarray, optional

List of gains to use in computing diagram

xlim : tuple or list, optional

control of x-axis range, normally with tuple (see matplotlib.axes)

ylim : tuple or list, optional

control of y-axis range

Plot : boolean, optional (default = True)

If True, plot magnitude and phase

PrintGain: boolean (default = True) :

If True, report mouse clicks when close to the root-locus branches, calculate gain, damping and print

Returns **rlist** : ndarray

Computed root locations, given as a 2d array

klist : ndarray or list

Gains used. Same as klist keyword argument if provided.

control.matlab.place

`control.matlab.place(A, B, p)`

Place closed loop eigenvalues

Parameters **A** : 2-d array

Dynamics matrix

B : 2-d array

Input matrix

p : 1-d list

Desired eigenvalue locations

Returns **K** : 2-d array

Gains such that A - B K has given eigenvalues

Examples

```
>>> A = [[-1, -1], [0, 1]]
>>> B = [[0], [1]]
>>> K = place(A, B, [-2, -5])
```

control.matlab.lqr

`control.matlab.lqr(*args, **keywords)`

Linear quadratic regulator design

The lqr() function computes the optimal state feedback controller that minimizes the quadratic cost

$$J = \int_0^\infty (x'Qx + u'Ru + 2x'Nu)dt$$

The function can be called with either 3, 4, or 5 arguments:

- `lqr(sys, Q, R)`
- `lqr(sys, Q, R, N)`
- `lqr(A, B, Q, R)`
- `lqr(A, B, Q, R, N)`

where *sys* is an *LTI* object, and *A*, *B*, *Q*, *R*, and *N* are 2d arrays or matrices of appropriate dimension.

Parameters **A, B: 2-d array** :

Dynamics and input matrices

sys: LTI (StateSpace or TransferFunction) :

Linear I/O system

Q, R: 2-d array :

State and input weight matrices

N: 2-d array, optional :

Cross weight matrix

Returns **K: 2-d array** :

State feedback gains

S: 2-d array :

Solution to Riccati equation

E: 1-d array :

Eigenvalues of the closed loop system

Examples

```
>>> K, S, E = lqr(sys, Q, R, [N])
>>> K, S, E = lqr(A, B, Q, R, [N])
```

State-space (SS) models

rss([states, outputs, inputs])	Create a stable **continuous** random state space object.
drss([states, outputs, inputs])	Create a stable **discrete** random state space object.

Continued on next page

Table 5.8 – continued from previous page

ctrb(A, B)	Controllabilty matrix
obsv(A, C)	Observability matrix
gram(sys, type)	Gramian (controllability or observability)

control.matlab.ctrb

`control.matlab.ctrb(A, B)`

Controllabilty matrix

Parameters A, B: array_like or string :

Dynamics and input matrix of the system

Returns C: matrix :

Controllability matrix

Examples

```
>>> C = ctrb(A, B)
```

control.matlab.obsv

`control.matlab.obsv(A, C)`

Observability matrix

Parameters A, C: array_like or string :

Dynamics and output matrix of the system

Returns O: matrix :

Observability matrix

Examples

```
>>> O = obsv(A, C)
```

control.matlab.gram

`control.matlab.gram(sys, type)`

Gramian (controllability or observability)

Parameters sys: StateSpace :

State-space system to compute Gramian for

type: String :

Type of desired computation. *type* is either 'c' (controllability) or 'o' (observability). To compute the Cholesky factors of gramians use 'cf' (controllability) or 'of' (observability)

Returns gram: array :

Gramian of system

Raises ValueError :

- if system is not instance of StateSpace class
- if *type* is not 'c', 'o', 'cf' or 'of'
- if system is unstable (sys.A has eigenvalues not in left half plane)

ImportError :

if slycot routine sb03md cannot be found if slycot routine sb03od cannot be found

Examples

```
>>> Wc = gram(sys,'c')
>>> Wo = gram(sys,'o')
>>> Rc = gram(sys,'cf'), where Wc=Rc'*Rc
>>> Ro = gram(sys,'of'), where Wo=Ro'*Ro
```

Model simplification

`minreal`(sys[, tol, verbose])	Eliminates uncontrollable or unobservable states in state-space models or cancelling pole-zero pairs in transfer functions.
`hsvd`(sys)	Calculate the Hankel singular values.
`balred`(sys, orders[, method, alpha])	Balanced reduced order model of sys of a given order.
`modred`(sys, ELIM[, method])	Model reduction of *sys* by eliminating the states in *ELIM* using a given method.
`era`(YY, m, n, nin, nout, r)	Calculate an ERA model of order *r* based on the impulse-response data *YY*.
`markov`(Y, U, M)	Calculate the first *M* Markov parameters [D CB CAB ...] from input *U*, output *Y*.

control.matlab.minreal

control.matlab.**minreal**(*sys*, *tol=None*, *verbose=True*)

Eliminates uncontrollable or unobservable states in state-space models or cancelling pole-zero pairs in transfer functions. The output sysr has minimal order and the same response characteristics as the original model sys.

Parameters sys: StateSpace or TransferFunction :

Original system

tol: real :

Tolerance

verbose: bool :

Print results if True

Returns rsys: StateSpace or TransferFunction :

Cleaned model

control.matlab.hsvd

`control.matlab.hsvd(sys)`

Calculate the Hankel singular values.

Parameters **sys** : StateSpace

A state space system

Returns **H** : Matrix

A list of Hankel singular values

See also:

gram

Notes

The Hankel singular values are the singular values of the Hankel operator. In practice, we compute the square root of the eigenvalues of the matrix formed by taking the product of the observability and controllability gramians. There are other (more efficient) methods based on solving the Lyapunov equation in a particular way (more details soon).

Examples

```
>>> H = hsvd(sys)
```

control.matlab.balred

`control.matlab.balred(sys, orders, method='truncate', alpha=None)`

Balanced reduced order model of sys of a given order. States are eliminated based on Hankel singular value. If sys has unstable modes, they are removed, the balanced realization is done on the stable part, then reinserted in accordance with the reference below.

Reference: Hsu,C.S., and Hou,D., 1991, Reducing unstable linear control systems via real Schur transformation. Electronics Letters, 27, 984-986.

Parameters **sys: StateSpace** :

Original system to reduce

orders: integer or array of integer :

Desired order of reduced order model (if a vector, returns a vector of systems)

method: string :

Method of removing states, either `'truncate'` or `'matchdc'`.

alpha: float :

Redefines the stability boundary for eigenvalues of the system matrix A. By default for continuous-time systems, alpha <= 0 defines the stability boundary for the real part of A's eigenvalues and for discrete-time systems, 0 <= alpha <= 1 defines the stability boundary for the modulus of A's eigenvalues. See SLICOT routines AB09MD and AB09ND for more information.

Returns rsys: StateSpace :

A reduced order model or a list of reduced order models if orders is a list

Raises ValueError :

- if *method* is not `'truncate'` or `'matchdc'`

ImportError :

if slycot routine ab09ad, ab09md, or ab09nd is not found

ValueError :

if there are more unstable modes than any value in orders

Examples

```
>>> rsys = balred(sys, orders, method='truncate')
```

control.matlab.modred

`control.matlab.`**`modred`**(*sys*, *ELIM*, *method='matchdc'*)

Model reduction of *sys* by eliminating the states in *ELIM* using a given method.

Parameters sys: StateSpace :

Original system to reduce

ELIM: array :

Vector of states to eliminate

method: string :

Method of removing states in *ELIM*: either `'truncate'` or `'matchdc'`.

Returns rsys: StateSpace :

A reduced order model

Raises ValueError :

- if *method* is not either `'matchdc'` or `'truncate'`
- if eigenvalues of *sys.A* are not all in left half plane (*sys* must be stable)

Examples

```
>>> rsys = modred(sys, ELIM, method='truncate')
```

control.matlab.era

`control.matlab.era(YY, m, n, nin, nout, r)`

Calculate an ERA model of order *r* based on the impulse-response data *YY*.

Note: This function is not implemented yet.

Parameters **YY: array** :

nout x *nin* dimensional impulse-response data

m: integer :

Number of rows in Hankel matrix

n: integer :

Number of columns in Hankel matrix

nin: integer :

Number of input variables

nout: integer :

Number of output variables

r: integer :

Order of model

Returns **sys: StateSpace** :

A reduced order model sys=ss(Ar,Br,Cr,Dr)

Examples

```
>>> rsys = era(YY, m, n, nin, nout, r)
```

control.matlab.markov

`control.matlab.markov(Y, U, M)`

Calculate the first *M* Markov parameters [D CB CAB ...] from input *U*, output *Y*.

Parameters **Y: array_like** :

Output data

U: array_like :

Input data

M: integer :

Number of Markov parameters to output

Returns **H: matrix** :

First M Markov parameters

Notes

Currently only works for SISO

Examples

```
>>> H = markov(Y, U, M)
```

Time delays

pade(T[, n, numdeg])	Create a linear system that approximates a delay.

control.matlab.pade

`control.matlab.pade`(*T*, *n=1*, *numdeg=None*)

Create a linear system that approximates a delay.

Return the numerator and denominator coefficients of the Pade approximation.

Parameters **T** : number

time delay

n : positive integer

degree of denominator of approximation

numdeg: integer, or None (the default) :

If None, numerator degree equals denominator degree If >= 0, specifies degree of numerator If < 0, numerator degree is n+numdeg

Returns **num, den** : array

Polynomial coefficients of the delay model, in descending powers of s.

Notes

Based on:

1. Algorithm 11.3.1 in Golub and van Loan, "Matrix Computation" 3rd. Ed. pp. 572-574
2. M. Vajta, "Some remarks on Padé-approximations", 3rd TEMPUS-INTCOM Symposium

Matrix equation solvers and linear algebra

lyap(A, Q[, C, E])	X = lyap(A,Q) solves the continuous-time Lyapunov equation
dlyap(A, Q[, C, E])	dlyap(A,Q) solves the discrete-time Lyapunov equation

Continued on next page

Table 5.11 – continued from previous page

`care`(A, B, Q[, R, S, E])	(X,L,G) = care(A,B,Q,R=None) solves the continuous-time algebraic Riccati
`dare`(A, B, Q, R[, S, E])	(X,L,G) = dare(A,B,Q,R) solves the discrete-time algebraic Riccati

control.matlab.lyap

`control.matlab.lyap(A, Q, C=None, E=None)`

X = lyap(A,Q) solves the continuous-time Lyapunov equation

$$AX + XA^T + Q = 0$$

where A and Q are square matrices of the same dimension. Further, Q must be symmetric.

X = lyap(A,Q,C) solves the Sylvester equation

$$AX + XQ + C = 0$$

where A and Q are square matrices.

X = lyap(A,Q,None,E) solves the generalized continuous-time Lyapunov equation

$$AXE^T + EXA^T + Q = 0$$

where Q is a symmetric matrix and A, Q and E are square matrices of the same dimension.

control.matlab.dlyap

`control.matlab.dlyap(A, Q, C=None, E=None)`

dlyap(A,Q) solves the discrete-time Lyapunov equation

$$AXA^T - X + Q = 0$$

where A and Q are square matrices of the same dimension. Further Q must be symmetric.

dlyap(A,Q,C) solves the Sylvester equation

$$AXQ^T - X + C = 0$$

where A and Q are square matrices.

dlyap(A,Q,None,E) solves the generalized discrete-time Lyapunov equation

$$AXA^T - EXE^T + Q = 0$$

where Q is a symmetric matrix and A, Q and E are square matrices of the same dimension.

control.matlab.care

`control.matlab.care(A, B, Q, R=None, S=None, E=None)`

(X,L,G) = care(A,B,Q,R=None) solves the continuous-time algebraic Riccati equation

$$A^T X + XA - XBR^{-1}B^T X + Q = 0$$

where A and Q are square matrices of the same dimension. Further, Q and R are a symmetric matrices. If R is None, it is set to the identity matrix. The function returns the solution X, the gain matrix G = B^T X and the closed loop eigenvalues L, i.e., the eigenvalues of A - B G.

(X,L,G) = care(A,B,Q,R,S,E) solves the generalized continuous-time algebraic Riccati equation

$$A^T X E + E^T X A - (E^T X B + S) R^{-1} (B^T X E + S^T) + Q = 0$$

where A, Q and E are square matrices of the same dimension. Further, Q and R are symmetric matrices. If R is None, it is set to the identity matrix. The function returns the solution X, the gain matrix G = R^-1 (B^T X E + S^T) and the closed loop eigenvalues L, i.e., the eigenvalues of A - B G , E.

control.matlab.dare

`control.matlab.dare`(*A*, *B*, *Q*, *R*, *S=None*, *E=None*)

(X,L,G) = dare(A,B,Q,R) solves the discrete-time algebraic Riccati equation

$$A^T X A - X - A^T X B (B^T X B + R)^{-1} B^T X A + Q = 0$$

where A and Q are square matrices of the same dimension. Further, Q is a symmetric matrix. The function returns the solution X, the gain matrix G = (B^T X B + R)^-1 B^T X A and the closed loop eigenvalues L, i.e., the eigenvalues of A - B G.

(X,L,G) = dare(A,B,Q,R,S,E) solves the generalized discrete-time algebraic Riccati equation

$$A^T X A - E^T X E - (A^T X B + S)(B^T X B + R)^{-1}(B^T X A + S^T) + Q = 0$$

where A, Q and E are square matrices of the same dimension. Further, Q and R are symmetric matrices. The function returns the solution X, the gain matrix $G = (B^T X B + R)^{-1}(B^T X A + S^T)$ and the closed loop eigenvalues L, i.e., the eigenvalues of A - B G , E.

Additional functions

`gangof4`(P, C[, omega])	Plot the "Gang of 4" transfer functions for a system
`unwrap`(angle[, period])	Unwrap a phase angle to give a continuous curve

control.matlab.gangof4

`control.matlab.gangof4`(*P*, *C*, *omega=None*)

Plot the "Gang of 4" transfer functions for a system

Generates a 2x2 plot showing the "Gang of 4" sensitivity functions [T, PS; CS, S]

Parameters **P, C** : LTI

Linear input/output systems (process and control)

omega : array

Range of frequencies (list or bounds) in rad/sec

Returns **None** :

control.matlab.unwrap

`control.matlab.unwrap`(*angle*, *period=6.28*)

Unwrap a phase angle to give a continuous curve

Parameters **angle** : array_like

Array of angles to be unwrapped

period : float, optional

Period (defaults to *2*pi*)

Returns **angle_out** : array_like

Output array, with jumps of period/2 eliminated

Examples

```
>>> import numpy as np
>>> theta = [5.74, 5.97, 6.19, 0.13, 0.35, 0.57]
>>> unwrap(theta, period=2 * np.pi)
[5.74, 5.97, 6.19, 6.413185307179586, 6.633185307179586, 6.8531853071795865]
```

Functions imported from other modules

linspace
logspace
ss2zpk
tf2zpk
zpk2ss
zpk2tf

- genindex

Development

You can check out the latest version of the source code with the command:

```
git clone https://github.com/python-control/python-control.git
```

You can run a set of unit tests to make sure that everything is working correctly. After installation, run:

```
python setup.py test
```

Your contributions are welcome! Simply fork the GitHub repository and send a pull request.

Links

- Issue tracker: https://github.com/python-control/python-control/issues
- Mailing list: http://sourceforge.net/p/python-control/mailman/

Python Module Index

Index