Python Control Library Documentation

Release dev

RMM

Aug 16, 2019

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, LQR, H2, Hinf
- Model reduction: balanced realizations, Hankel singular values
- 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.

1.1 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 module* is available that provides many of the common functions corresponding to commands available in the MATLAB Control Systems Toolbox.

1.2 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.

1.3 Installation

The *python-control* package can be installed using pip, conda or the standard distutils/setuptools mechanisms. The package requires numpy and scipy, and the plotting routines require matplotlib. In addition, some routines require the slycot library in order to implement more advanced features (including some MIMO functionality).

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 either the *lapack* or *openplas* 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 conda-forge control
```

This installs *slycot* and *python-control* from conda-forge, including the *openblas* 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
```

1.4 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.

2.1 LTI system representation

Linear time invariant (LTI) systems are represented in python-control in state space, transfer function, or frequency response data (FRD) form. Most functions in the toolbox will operate on any of these data types and functions for converting between compatible types is provided.

2.1.1 State space systems

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

$$\frac{dx}{dt} = Ax + Bu$$
$$y = Cx + Du$$

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

To create a state space system, use the *`StateSpace`* constructor:

sys = StateSpace(A, B, C, D)

State space systems can be manipulated using standard arithmetic operations as well as the *`feedback()`*, *`parallel()`*, and *`series()`* function. A full list of functions can be found in *Function reference*.

2.1.2 Transfer functions

The *`TransferFunction`* class is used to represent input/output transfer functions

$$G(s) = \frac{\text{num}(s)}{\text{den}(s)} = \frac{a_0 s^m + a_1 s^{m-1} + \cdots + a_m}{b_0 s^n + b_1 s^{n-1} + \cdots + b_n},$$

where n is generally greater than or equal to m (for a proper transfer function).

To create a transfer function, use the `TransferFunction` constructor:

> sys = TransferFunction(num, den)

Transfer functions can be manipulated using standard arithmetic operations as well as the `feedback()`, `parallel()`, and `series()` function. A full list of functions can be found in *Function reference*.

2.1.3 FRD (frequency response data) systems

The `FrequencyResponseData` (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.

FRD systems have a somewhat more limited set of functions that are available, although all of the standard algebraic manipulations can be performed.

2.1.4 Discrete time systems

A discrete time system is created by specifying a nonzero 'timebase', dt. The timebase argument can be given when a system is constructed:

- dt = None: no timebase specified (default)
- dt = 0: continuous time system
- dt > 0: discrete time system with sampling period 'dt'
- dt = True: discrete time with unspecified sampling period

Only the `StateSpace`, `TransferFunction`, and `InputOutputSystem` classes allow explicit representation of discrete time systems.

Systems must have compatible timebases in order to be combined. A system with timebase *None* can be combined with a system having a specified timebase; the result will have the timebase of the latter system. Similarly, a discrete time system with unspecified sampling time (*dt = True*) can be combined with a system having a specified sampling time; the result will be a discrete time system with the sample time of the latter system. For continuous time systems, the `sample_system()` function or the `StateSpace.sample()` and `TransferFunction.sample()` methods can be used to create a discrete time system from a continuous time system. See *Utility functions and conversions*.

2.1.5 Conversion between representations

LTI systems can be converted between representations either by calling the constructor for the desired data type using the original system as the sole argument or using the explicit conversion functions `ss2tf()` and `tf2ss()`.

2.2 Input/output systems

The `iosys` module contains the `InputOutputSystem` class that represents (possibly nonlinear) input/output systems. The `InputOutputSystem` class is a general class that defines any continuous or discrete time dynamical system. Input/output systems can be simulated and also used to compute equilibrium points and linearizations.

An input/output system is defined as a dynamical system that has a system state as well as inputs and outputs (either inputs or states can be empty). The dynamics of the system can be in continuous or discrete time. To simulate an input/output system, use the *input_output_response()* function:

```
t, y = input_output_response(io_sys, T, U, X0, params)
```

An input/output system can be linearized around an equilibrium point to obtain a *StateSpace* linear system. Use the `find_eqpts()` function to obtain an equilibrium point and the *linearize()* function to linearize about that equilibrium point:

```
xeq, ueq = find_eqpt(io_sys, X0, U0)
ss_sys = linearize(io_sys, xeq, ueq)
```

Input/output systems can be created from state space LTI systems by using the *LinearIOSystem* class':

```
io_sys = LinearIOSystem(ss_sys)
```

Nonlinear input/output systems can be created using the `NonlinearIoSystem` class, which requires the definition of an update function (for the right hand side of the differential or different equation) and and output function (computes the outputs from the state):

```
io_sys = NonlinearIOSystem(updfcn, outfcn, inputs=M, outputs=P, states=N)
```

More complex input/output systems can be constructed by using the *InterconnectedSystem* class, which allows a collection of input/output subsystems to be combined with internal connections between the subsystems and a set of overall system inputs and outputs that link to the subsystems:

```
steering = ct.InterconnectedSystem(
    (plant, controller), name='system',
    connections=(('controller.e', '-plant.y')),
    inplist=('controller.e'), inputs='r',
    outlist=('plant.y'), outputs='y')
```

Interconnected systems can also be created using block diagram manipulations such as the *series()*, *parallel()*, and *feedback()* functions. The *InputOutputSystem* class also supports various algebraic operations such as * (series interconnection) and + (parallel interconnection).

2.3 Time series data

A variety of functions in the library return time series data: sequences of values that change over time. A common set of conventions is used for returning such data: columns represent different points in time, rows are different components (e.g., inputs, outputs or states). For return arguments, an array of times is given as the first returned argument, followed by one or more arrays of variable values. This convention is used throughout the library, for example in the functions *forced_response()*, *step_response()*, *impulse_response()*, and *initial_response()*.

Note: The convention used by python-control is different from the convention used in the scipy.signal library. 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_response(sys)
plot(t, y)
```

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

```
t, y, x = forced_response(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
```

2.4 Package configuration parameters

The python-control library can be customized to allow for different default values for selected parameters. This includes the ability to set the style for various types of plots and establishing the underlying representation for state space matrices.

To set the default value of a configuration variable, set the appropriate element of the *control.config.defaults* dictionary:

```
control.config.defaults['module.parameter'] = value
```

The *~control.config.set_defaults* function can also be used to set multiple configuration parameters at the same time:

```
control.config.set_defaults('module', param1=val1, param2=val2, ...]
```

Finally, there are also functions available set collections of variables based on standard configurations.

Selected 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.grid (True): Include grids for magnitude and phase plots
- freqplot.number_of_samples (None): Number of frequency points in Bode plots
- freqplot.feature_periphery_decade (1.0): How many decades to include in the frequency range on both sides of features (poles, zeros).
- statesp.use_numpy_matrix: set the return type for state space matrices to *numpy.matrix* (verus numpy.ndarray)

Additional parameter variables are documented in individual functions

Functions that can be used to set standard configurations:

`reset_defaults`()	Reset configuration values to their default (initial) values.
`use_fbs_defaults`()	Use Feedback Systems (FBS) compatible settings.
`use_matlab_defaults`()	Use MATLAB compatible configuration settings.
`use_numpy_matrix`([flag, warn])	Turn on/off use of Numpy *matrix* class for state space operations.

2.4.1 control.reset_defaults

control.**reset_defaults**()

Reset configuration values to their default (initial) values.

2.4.2 control.use_fbs_defaults

control.**use_fbs_defaults**()

Use Feedback Systems (FBS) compatible settings.

The following conventions are used:

- Bode plots plot gain in powers of ten, phase in degrees, frequency in Hertz, no grid

2.4.3 control.use_matlab_defaults

control.**use_matlab_defaults**()

Use MATLAB compatible configuration settings.

The following conventions are used:

- Bode plots plot gain in dB, phase in degrees, frequency in Hertz, with grids
- State space class and functions use Numpy matrix objects

2.4.4 control.use_numpy_matrix

control.**use_numpy_matrix**(*flag=True*, *warn=True*)

Turn on/off use of Numpy *matrix* class for state space operations.

Parameters

- **flag** (*bool*) – If flag is *True* (default), use the Numpy (soon to be deprecated) *matrix* class to represent matrices in the *~control.StateSpace* class and functions. If flat is *False*, then matrices are represented by a 2D *ndarray* object.
- **warn** (*bool*) – If flag is *True* (default), issue a warning when turning on the use of the Numpy *matrix* class. Set *warn* to false to omit display of the warning message.

CHAPTER 3

Function reference

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

3.1 System creation

`ss`(A, B, C, D[, dt])	Create a state space system.
`tf`(num, den[, dt])	Create a transfer function system.
`frd`(d, w)	Construct a frequency response data model
`rss`([states, outputs, inputs])	Create a stable *continuous* random state space object.
`drss`([states, outputs, inputs])	Create a stable *discrete* random state space object.

3.1.1 control.ss

`control.ss`(*A*, *B*, *C*, *D*[, *dt*])

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** – The new linear system

Return type *StateSpace*

Raises `ValueError` – if matrix sizes are not self-consistent

See also:

StateSpace(), *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)
```

3.1.2 control.tf

control.**tf**(*num*, *den*[, *dt*])

Create a transfer function system. Can create MIMO systems.

The function accepts either 1, 2, or 3 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.

tf('s') or tf('z') Create a transfer function representing the differential operator ('s') or delay operator ('z').

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** – The new linear system

Return type *TransferFunction*

Raises

- ValueError – if *num* and *den* have invalid or unequal dimensions
- TypeError – if *num* or *den* are of incorrect type

See also:

TransferFunction(), *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 $2s^2 + 3s + 4$.

The special forms `tf('s')` and `tf('z')` can be used to create transfer functions for differentiation and unit delays.

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)
```

```
>>> # Create a variable 's' to allow algebra operations for SISO systems
>>> s = tf('s')
>>> G  = (s + 1)/(s**2 + 2*s + 1)
```

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

3.1.3 control.frd

`control.`**`frd`**`(d, w)`

Construct a frequency response data model

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 – New frequency response system

Return type FRD

See also:

`FRD()`, `ss()`, `tf()`

3.1.4 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 – The randomly created linear system

Return type *StateSpace*

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.

3.1.5 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** – The randomly created linear system

Return type *StateSpace*

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.

3.2 System interconnections

append(sys1, sys2, . . . , sysn)	Group models by appending their inputs and outputs
connect(sys, Q, inputv, outputv)	Index-based interconnection of an LTI system.
feedback(sys1[, sys2, sign])	Feedback interconnection between two I/O systems.
negate(sys)	Return the negative of a system.
parallel(sys1, *sysn)	Return the parallel connection sys1 + sys2 (+ sys3 + . . .)
series(sys1, *sysn)	Return the series connection (. . .

3.2.1 control.append

control.**append**(*sys1*, *sys2*, *...*, *sysn*)

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 **sys2, .. sysn** (*sys1,*) – LTI systems to combine

Returns **sys** – Combined LTI system, with input/output vectors consisting of all input/output vectors appended

Return type LTI system

Examples

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

3.2.2 control.connect

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

Index-based interconnection of an LTI 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: Inputs and outputs are indexed starting at 1 and negative values correspond to a negative feedback interconnection.

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. Inputs and outputs are indexed starting at 1.
- **inputv** (*1D array*) – list of final external inputs
- **outputv** (*1D array*) – list of final external outputs

Returns **sys** – Connected and trimmed LTI system

Return type 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 = [[1, 2], [2, -1]]  # negative feedback interconnection
>>> sysc = connect(sys, Q, [2], [1, 2])
```

3.2.3 control.feedback

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

Feedback interconnection between two I/O systems.

Parameters

- **sys1** (*scalar, StateSpace, TransferFunction, FRD*) – The primary process.
- **sys2** (*scalar, StateSpace, TransferFunction, FRD*) – The feedback process (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**

Return type *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.

3.2.4 control.negate

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

Return the negative of a system.

Parameters **sys** (StateSpace, TransferFunction *or* FRD) –

Returns out

Return type *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.

Examples

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

3.2.5 control.parallel

control.**parallel**(*sys1*, **sysn*)

Return the parallel connection sys1 + sys2 (+ sys3 + . . .)

Parameters

- **sys1** (*scalar*, StateSpace, TransferFunction, *or* FRD) –
- ***sysn** (*other scalars, StateSpaces, TransferFunctions, or FRDs*) –

Returns out

Return type 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
```

```
>>> sys5 = parallel(sys1, sys2, sys3, sys4) # More systems
```

3.2.6 control.series

`control.series(sys1, *sysn)`

Return the series connection (. . . * sys3 *) sys2 * sys1

Parameters

- **sys1** (*scalar, StateSpace, TransferFunction, or FRD*) –
- **sysn** (*other scalars, StateSpaces, TransferFunctions, or FRDs*) –

Returns out

Return type 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
```

```
>>> sys5 = series(sys1, sys2, sys3, sys4) # More systems
```

3.3 Frequency domain plotting

`bode_plot`(syslist[, omega, Plot, . . .])	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`(sys_list[, omega, grid])	Nichols plot for a system

3.3.1 control.bode_plot

control.**bode_plot**(*syslist*, *omega=None*, *Plot=True*, *omega_limits=None*, *omega_num=None*, *margins=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** (*list*) – List of frequencies in rad/sec to be used for frequency response
- **dB** (*bool*) – If True, plot result in dB. Default is false.
- **Hz** (*bool*) – If True, plot frequency in Hz (omega must be provided in rad/sec). Default value (False) set by config.defaults['bode.Hz']
- **deg** (*bool*) – If True, plot phase in degrees (else radians). Default value (True) config.defaults['bode.deg']
- **Plot** (*bool*) – 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 to plot. Defaults to config.defaults['freqplot.number_of_samples'].
- **margins** (*bool*) – If True, plot gain and phase margin.
- ****kwargs** (**args,*) – 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

Other Parameters

- **grid** (*bool*) – If True, plot grid lines on gain and phase plots. Default is set by config.defaults['bode.grid'].
- **The default values for Bode plot configuration parameters can be reset**
- **using the 'config.defaults' dictionary, with module name 'bode'.**

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 timebase. 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)
```

3.3.2 control.nyquist_plot

`control.`**`nyquist_plot`**(*syslist*, *omega=None*, *Plot=True*, *color=None*, *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
- **`color`** (*`string`*) – Used to specify the color of the plot
- **`labelFreq`** (*`int`*) – Label every nth frequency on the plot
- **`**kwargs`** (*`*args,`*) – 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)
```

3.3.3 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

- **C** (*P,*) – Linear input/output systems (process and control)
- **omega** (*array*) – Range of frequencies (list or bounds) in rad/sec

Returns

Return type None

3.3.4 control.nichols_plot

control.**nichols_plot**(*sys_list*, *omega=None*, *grid=None*)

Nichols plot for a system

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

Parameters

- **sys_list** (*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

Return type None

Note: For plotting commands that create multiple axes on the same plot, the individual axes can be retrieved using the axes label (retrieved using the *get_label* method for the matplotliib axes object). The following labels are currently defined:

- Bode plots: *control-bode-magnitude*, *control-bode-phase*
- Gang of 4 plots: *control-gangof4-s*, *control-gangof4-cs*, *control-gangof4-ps*, *control-gangof4-t*

3.4 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
input_output_response(sys, T[, U, X0, . . .])	Compute the output response of a system to a given input.
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

3.4.1 control.forced_response

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

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, optional for discrete LTI *sys*) – 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, optional (default=False)*) – If True, transpose all input and output arrays (for backward compatibility with MATLAB and scipy.signal.lsim)
- **interpolate** (*bool, optional (default=False)*) – If True and system is a discrete time system, the input will be interpolated between the given time steps and the output will be given at system sampling rate. Otherwise, only return the output at the times given in *T*. No effect on continuous time simulations (default = False).
- **squeeze** (*bool, optional (default=True)*) – If True, remove single-dimensional entries from the shape of the output. For single output systems, this converts the output response to a 1D array.

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*.

3.4.2 control.impulse_response

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

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).
- **squeeze** (*bool, optional (default=True)*) – If True, remove single-dimensional entries from the shape of the output. For single output systems, this converts the output response to a 1D array.

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()*, *step_response()*

Examples

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

3.4.3 control.initial_response

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

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).
- **squeeze** (*bool, optional (default=True)*) – If True, remove single-dimensional entries from the shape of the output. For single output systems, this converts the output response to a 1D array.

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)
```

3.4.4 control.input_output_response

control.**input_output_response**(*sys, T, U=0.0, X0=0, params={}, method='RK45', return_x=False, squeeze=True*)

Compute the output response of a system to a given input.

Simulate a dynamical system with a given input and return its output and state values.

Parameters

- **sys** (InputOutputSystem) – Input/output 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).
- **X0** (*array-like or number, optional*) – Initial condition (default = 0).
- **return_x** (*bool, optional*) – If True, return the values of the state at each time (default = False).
- **squeeze** (*bool, optional*) – If True (default), squeeze unused dimensions out of the output response. In particular, for a single output system, return a vector of shape (nsteps) instead of (nsteps, 1).

Returns

- **T** (*array*) – Time values of the output.
- **yout** (*array*) – Response of the system.

- **xout** (*array*) – Time evolution of the state vector (if return_x=True)

Raises

- `TypeError` – If the system is not an input/output system.
- `ValueError` – If time step does not match sampling time (for discrete time systems)

3.4.5 control.step_response

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

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).
- **squeeze** (*bool, optional (default=True)*) – If True, remove single-dimensional entries from the shape of the output. For single output systems, this converts the output response to a 1D array.

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)
```

3.4.6 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.
- **Y** (*X,*) – Two 3-element sequences specifying x and y coordinates of a grid. These arguments are passed to linspace and meshgrid to generate the 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.
- **= N or (N, M)** (*lingrid*) – 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.
- **= N** (*lintime*) – Draw N arrows using equally space time points
- **= (N, lambda)** (*logtime*) – Draw N arrows using exponential time constant lambda
- **= [t1, t2, ..]** (*timepts*) – 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() construct box-shaped grid of initial conditions

Examples

3.5 Block diagram algebra

series(sys1, *sysn)	Return the series connection (…
parallel(sys1, *sysn)	Return the parallel connection sys1 + sys2 (+ sys3 + …)

Continued on next page

Table 5 – continued from previous page

feedback(sys1[, sys2, sign])	Feedback interconnection between two I/O systems.
negate(sys)	Return the negative of a system.

3.6 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(sysdata)	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, grid, title])	Plot a pole/zero map for a linear system.
root_locus(sys[, kvect, xlim, ylim, . . .])	Root locus plot
sisotool(sys[, kvect, xlim_rlocus, . . .])	Sisotool style collection of plots inspired by MATLAB's sisotool.

3.6.1 control.dcgain

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

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

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

Return type ndarray

3.6.2 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

Return type 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

3.6.3 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.

3.6.4 control.margin

control.**margin**(*sysdata*)

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)
- **wg** (*float*) – Frequency for gain margin (at phase crossover, phase = -180 degrees)
- **wp** (*float*) – Frequency for phase margin (at gain crossover, gain = 1)
- *Margins are calculated for a SISO open-loop system.*
- *If there is more than one gain crossover, the one at the smallest*
- *margin (deviation from gain = 1), in absolute sense, is*
- *returned. Likewise the smallest phase margin (in absolute sense)*
- *is returned.*

Examples

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

3.6.5 control.stability_margins

control.**stability_margins**(*sysdata*, *returnall=False*, *epsw=0.0*)

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 margins in the given frequency region can be found and returned.
- **epsw** (*float, optional*) – Frequencies below this value (default 0.0) 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*) – Frequency for gain margin (at phase crossover, phase = -180 degrees)
- **wp** (*float or array_like*) – Frequency for phase margin (at gain crossover, gain = 1)
- **ws** (*float or array_like*) – Frequency for stability margin (complex gain closest to -1)

3.6.6 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]))
```

3.6.7 control.pole

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

Compute system poles.

Parameters **sys** (*StateSpace or TransferFunction*) – Linear system

Returns **poles** – Array that contains the system's poles.

Return type ndarray

Raises NotImplementedError – when called on a TransferFunction object

See also:

zero(), *TransferFunction.pole()*, *StateSpace.pole()*

3.6.8 control.zero

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

Compute system zeros.

Parameters **sys** (StateSpace *or* TransferFunction) – Linear system

Returns **zeros** – Array that contains the system's zeros.

Return type ndarray

Raises NotImplementedError – when called on a MIMO system

See also:

pole(), *StateSpace.zero()*, *TransferFunction.zero()*

3.6.9 control.pzmap

control.**pzmap**(*sys*, *Plot=True*, *grid=False*, *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.
- **grid** (*boolean (default = False)*) – If True plot omega-damping grid.

Returns

- **pole** (*array*) – The systems poles
- **zeros** (*array*) – The system's zeros.

3.6.10 control.root_locus

control.**root_locus**(*sys*, *kvect=None*, *xlim=None*, *ylim=None*, *plotstr=None*, *Plot=True*, *PrintGain=None*, *grid=None*, ***kwargs*)

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*) – Set limits of x axis, normally with tuple (see matplotlib.axes).
- **ylim** (*tuple or list, optional*) – Set limits of y axis, normally with tuple (see matplotlib.axes).
- **Plot** (*boolean, optional*) – If True (default), plot root locus diagram.
- **PrintGain** (*bool*) – If True (default), report mouse clicks when close to the root locus branches, calculate gain, damping and print.

- **grid** (*bool*) – If True plot omega-damping grid. Default is False.

Returns

- **rlist** (*ndarray*) – Computed root locations, given as a 2D array
- **klist** (*ndarray or list*) – Gains used. Same as klist keyword argument if provided.

3.6.11 control.sisotool

control.**sisotool**(*sys*, *kvect=None*, *xlim_rlocus=None*, *ylim_rlocus=None*, *plotstr_rlocus='C0'*, *rlocus_grid=False*, *omega=None*, *dB=None*, *Hz=None*, *deg=None*, *omega_limits=None*, *omega_num=None*, *margins_bode=True*, *tvect=None*)

Sisotool style collection of plots inspired by MATLAB's sisotool. The left two plots contain the bode magnitude and phase diagrams. The top right plot is a clickable root locus plot, clicking on the root locus will change the gain of the system. The bottom left plot shows a closed loop time response.

Parameters

- **sys** (*LTI object*) – Linear input/output systems (SISO only)
- **kvect** (*list or ndarray, optional*) – List of gains to use for plotting root locus
- **xlim_rlocus** (*tuple or list, optional*) – control of x-axis range, normally with tuple (see matplotlib.axes)
- **ylim_rlocus** (*tuple or list, optional*) – control of y-axis range
- **plotstr_rlocus** (*Additional options to matplotlib*) – plotting style for the root locus plot(color, linestyle, etc)
- **rlocus_grid** (*boolean (default = False)*) – If True plot s-plane grid.
- **omega** (*freq_range*) – Range of frequencies in rad/sec for the bode plot
- **dB** (*boolean*) – If True, plot result in dB for the bode plot
- **Hz** (*boolean*) – If True, plot frequency in Hz for the bode plot (omega must be provided in rad/sec)
- **deg** (*boolean*) – If True, plot phase in degrees for the bode plot (else radians)
- **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
- **margins_bode** (*boolean*) – If True, plot gain and phase margin in the bode plot
- **tvect** (*list or ndarray, optional*) – List of timesteps to use for closed loop step response

Examples

```
>>> sys = tf([1000], [1,25,100,0])
>>> sisotool(sys)
```

3.7 Matrix computations

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

3.7.1 control.care

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

(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.

3.7.2 control.dare

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

(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.

3.7.3 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.

3.7.4 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.

3.7.5 control.ctrb

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

Controllabilty matrix

Parameters **B** (*A,*) – Dynamics and input matrix of the system

Returns **C** – Controllability matrix

Return type matrix

Examples

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

3.7.6 control.obsv

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

Observability matrix

Parameters **C** (*A,*) – Dynamics and output matrix of the system

Returns **O** – Observability matrix

Return type matrix

Examples

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

3.7.7 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** – Gramian of system

Return type array

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
```

3.8 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(A, B, Q, R[, N])	Linear quadratic regulator design
mixsyn(g[, w1, w2, w3])	Mixed-sensitivity H-infinity synthesis.
place(A, B, p)	Place closed loop eigenvalues K = place(A, B, p)

3.8.1 control.acker

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

Pole placement using Ackermann method

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

Parameters

- **B** (*A,*) – State and input matrix of the system
- **poles** (*1-d list*) – Desired eigenvalue locations

Returns **K** – Gains such that A - B K has given eigenvalues

Return type matrix

3.8.2 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**

Return type 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)
```

3.8.3 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)
```

3.8.4 control.lqr

`control.lqr`(*A*, *B*, *Q*, *R*[, *N*])

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

- **B** (*A,*) – Dynamics and input matrices
- **sys** (*LTI (StateSpace or TransferFunction)*) – Linear I/O system
- **R** (*Q,*) – State and input weight matrices
- **N** (*2-d array, optional*) – Cross weight matrix

Returns

- **K** (*2D array*) – State feedback gains
- **S** (*2D array*) – Solution to Riccati equation
- **E** (*1D 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])
```

3.8.5 control.mixsyn

`control.mixsyn`(*g*, *w1=None*, *w2=None*, *w3=None*)

Mixed-sensitivity H-infinity synthesis.

mixsyn(g,w1,w2,w3) -> k,cl,info

Parameters

- **g** (*LTI; the plant for which controller must be synthesized*) –
- **w1** (*weighting on s = (1+g*k)**-1; None, or scalar or k1-by-ny LTI*) –
- **w2** (*weighting on k*s; None, or scalar or k2-by-nu LTI*) –
- **w3** (*weighting on t = g*k*(1+g*k)**-1; None, or scalar or k3-by-ny LTI*) –
- **least one of w1, w2, and w3 must not be None.** (*At*) –

Returns

- **k** (*synthesized controller; StateSpace object*)
- **cl** (*closed system mapping evaluation inputs to evaluation outputs; if*)
- *p is the augmented plant, with* – [z] = [p11 p12] [w], [y] [p21 g] [u]
- *then cl is the system from w->z with u=-k*y. StateSpace object.*
- **info** (*tuple with entries, in order,*) –
  - gamma: scalar; H-infinity norm of cl
  - rcond: array; estimates of reciprocal condition numbers computed during synthesis. See hinfsyn for details
- *If a weighting w is scalar, it will be replaced by I*w, where I is*
- *ny-by-ny for w1 and w3, and nu-by-nu for w2.*

See also:

`hinfsyn()`, `augw()`

3.8.6 control.place

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

Place closed loop eigenvalues K = place(A, B, p)

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*) – Gain such that A - B K has eigenvalues given in p
- *Algorithm*
- ———
- *This is a wrapper function for scipy.signal.place_poles, which*
- *implements the Tits and Yang algorithm [1]. It will handle SISO,*
- *MISO, and MIMO systems. If you want more control over the algorithm,*
- *use scipy.signal.place_poles directly.*

- *[1] A.L. Tits and Y. Yang, "Globally convergent algorithms for robust*
- *pole assignment by state feedback, IEEE Transactions on Automatic*
- *Control, Vol. 41, pp. 1432-1452, 1996.*
- *Limitations*
- *———*
- *The algorithm will not place poles at the same location more*
- *than rank(B) times.*

Examples

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

See also:

`place_varga()`, `acker()`

3.9 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*.

3.9.1 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** – Cleaned model

Return type *StateSpace* or *TransferFunction*

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

Return type *StateSpace*

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')
```

3.9.3 control.hsvd

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

Calculate the Hankel singular values.

Parameters **sys** (`StateSpace`) – A state space system

Returns **H** – A list of Hankel singular values

Return type array

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)
```

3.9.4 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** – A reduced order model

Return type *StateSpace*

Raises `ValueError` – Raised under the following conditions:

- 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')
```

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

Return type *StateSpace*

Examples

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

3.9.6 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** (*int*) – Number of Markov parameters to output

Returns **H** – First m Markov parameters

Return type ndarray

Notes

Currently only works for SISO

Examples

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

3.10 Nonlinear system support

find_eqpt(sys, x0[, u0, y0, t, params, iu, . . .])	Find the equilibrium point for an input/output system.
linearize(sys, xeq[, ueq, t, params])	Linearize an input/output system at a given state and input.

3.10.1 control.find_eqpt

control.**find_eqpt**(*sys*, *x0*, *u0=[]*, *y0=None*, *t=0*, *params={}*, *iu=None*, *iy=None*, *ix=None*, *idx=None*, *dx0=None*, *return_y=False*, *return_result=False*, ***kw*)

Find the equilibrium point for an input/output system.

Returns the value of an equlibrium point given the initial state and either input value or desired output value for the equilibrium point.

Parameters

- **x0** (*list of initial state values*) – Initial guess for the value of the state near the equilibrium point.
- **u0** (*list of input values, optional*) – If *y0* is not specified, sets the equilibrium value of the input. If *y0* is given, provides an initial guess for the value of the input. Can be omitted if the system does not have any inputs.
- **y0** (*list of output values, optional*) – If specified, sets the desired values of the outputs at the equilibrium point.
- **t** (*float, optional*) – Evaluation time, for time-varying systems
- **params** (*dict, optional*) – Parameter values for the system. Passed to the evaluation functions for the system as default values, overriding internal defaults.
- **iu** (*list of input indices, optional*) – If specified, only the inputs with the given indices will be fixed at the specified values in solving for an equilibrium point. All other inputs will be varied. Input indices can be listed in any order.
- **iy** (*list of output indices, optional*) – If specified, only the outputs with the given indices will be fixed at the specified values in solving for an equilibrium point. All other outputs will be varied. Output indices can be listed in any order.
- **ix** (*list of state indices, optional*) – If specified, states with the given indices will be fixed at the specified values in solving for an equilibrium point. All other states will be varied. State indices can be listed in any order.
- **dx0** (*list of update values, optional*) – If specified, the value of update map must match the listed value instead of the default value of 0.
- **idx** (*list of state indices, optional*) – If specified, state updates with the given indices will have their update maps fixed at the values given in *dx0*. All other update values will be ignored in solving for an equilibrium point. State indices can be listed in any order. By default, all updates will be fixed at *dx0* in searching for an equilibrium point.
- **return_y** (*bool, optional*) – If True, return the value of output at the equilibrium point.
- **return_result** (*bool, optional*) – If True, return the *result* option from the scipy root function used to compute the equilibrium point.

Returns

- **xeq** (*array of states*) – Value of the states at the equilibrium point, or *None* if no equilibrium point was found and *return_result* was False.
- **ueq** (*array of input values*) – Value of the inputs at the equilibrium point, or *None* if no equilibrium point was found and *return_result* was False.
- **yeq** (*array of output values, optional*) – If *return_y* is True, returns the value of the outputs at the equilibrium point, or *None* if no equilibrium point was found and *return_result* was False.
- **result** (*scipy root() result object, optional*) – If *return_result* is True, returns the *result* from the scipy root function.

3.10.2 control.linearize

control.**linearize**(*sys*, *xeq*, *ueq=[]*, *t=0*, *params={}*, ***kw*)

Linearize an input/output system at a given state and input.

This function computes the linearization of an input/output system at a given state and input value and returns a `control.StateSpace` object. The eavaluation point need not be an equilibrium point.

Parameters

- **sys** (`InputOutputSystem`) – The system to be linearized
- **xeq** (*array*) – The state at which the linearization will be evaluated (does not need to be an equlibrium state).
- **ueq** (*array*) – The input at which the linearization will be evaluated (does not need to correspond to an equlibrium state).
- **t** (*float, optional*) – The time at which the linearization will be computed (for time-varying systems).
- **params** (*dict, optional*) – Parameter values for the systems. Passed to the evaluation functions for the system as default values, overriding internal defaults.

Returns **ss_sys** – The linearization of the system, as a `LinearIOSystem` object (which is also a `StateSpace` object.

Return type *LinearIOSystem*

3.11 Utility functions and conversions

`augw`(g[, w1, w2, w3])	Augment plant for mixed sensitivity problem.
`canonical_form`(xsys[, form])	Convert a system into canonical form
`damp`(sys[, doprint])	Compute natural frequency, damping ratio, and poles of a system
`db2mag`(db)	Convert a gain in decibels (dB) to a magnitude
`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])	Check to see if a system is single input, single output
`issys`(obj)	Return True if an object is a system, otherwise False
`mag2db`(mag)	Convert a magnitude to decibels (dB)
`observable_form`(xsys)	Convert a system into observable canonical form
`pade`(T[, n, numdeg])	Create a linear system that approximates a delay.
`reachable_form`(xsys)	Convert a system into reachable canonical form
`reset_defaults`()	Reset configuration values to their default (initial) values.
`sample_system`(sysc, Ts[, method, alpha])	Convert a continuous time system to discrete time
`ss2tf`(sys)	Transform a state space system to a transfer function.
`ssdata`(sys)	Return state space data objects for a system
`tf2ss`(sys)	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
`unwrap`(angle[, period])	Unwrap a phase angle to give a continuous curve

Continued on next page

Table 11 – continued from previous page

use_fbs_defaults()	Use Feedback Systems (FBS) compatible settings.
use_matlab_defaults()	Use MATLAB compatible configuration settings.
use_numpy_matrix([flag, warn])	Turn on/off use of Numpy *matrix* class for state space operations.

3.11.1 control.augw

control.**augw**(*g*, *w1=None*, *w2=None*, *w3=None*)

Augment plant for mixed sensitivity problem.

Parameters

- **g** (*LTI object, ny-by-nu*) –
- **w1** (*weighting on S; None, scalar, or k1-by-ny LTI object*) –
- **w2** (*weighting on KS; None, scalar, or k2-by-nu LTI object*) –
- **w3** (*weighting on T; None, scalar, or k3-by-ny LTI object*) –
- **p** (*augmented plant; StateSpace object*) –
- **a weighting is None, no augmentation is done for it. At least** (*If*) –
- **weighting must not be None.** (*one*) –
- **a weighting w is scalar, it will be replaced by I*w, where I is** (*If*) –
- **for w1 and w3, and nu-by-nu for w2.** (*ny-by-ny*) –

Returns p

Return type plant augmented with weightings, suitable for submission to hinfsyn or h2syn.

Raises ValueError – - if all weightings are None

See also:

h2syn(), *hinfsyn()*, *mixsyn()*

3.11.2 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

Returns

- **zsys** (*StateSpace object*) – System in desired canonical form, with state 'z'

- **T** (*matrix*) – Coordinate transformation matrix, z = T * x

3.11.3 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
- *Algorithm*
- ———
- *If the system is continuous,* – wn = abs(poles) Z = -real(poles)/poles.
- *If the system is discrete, the discrete poles are mapped to their*
- *equivalent location in the s-plane via* – s = log10(poles)/dt
- *and* – wn = abs(s) Z = -real(s)/wn.

See also:

pole()

3.11.4 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** – corresponding magnitudes

Return type float or ndarray

3.11.5 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

3.11.6 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

3.11.7 control.issiso

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

Check to see if a system is single input, single output

Parameters

- **sys** (*LTI system*) – System to be checked
- **strict** (*bool (default = False)*) – If strict is True, do not treat scalars as SISO

3.11.8 control.issys

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

Return True if an object is a system, otherwise False

3.11.9 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** – corresponding values in decibels

Return type float or ndarray

3.11.10 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

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

Return type array

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

3.11.12 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

3.11.13 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** – Discrete time system, with sampling rate Ts

Return type linsys

Notes

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

Examples

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

3.11.14 control.ss2tf

`control.`**`ss2tf`**(*sys*)

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** – New linear system in transfer function form

Return type *TransferFunction*

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)
```

3.11.15 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)** – State space data for the system

Return type list of matrices

3.11.16 control.tf2ss

`control.`**`tf2ss`**(*sys*)

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** – New linear system in state space form

Return type *StateSpace*

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)
```

3.11.17 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) – Transfer function coefficients (SISO only)

Return type numerator and denominator arrays

3.11.18 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.

3.11.19 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.

3.11.20 control.unwrap

control.**unwrap**(*angle*, *period=6.283185307179586*)

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 – Output array, with jumps of period/2 eliminated

Return type array_like

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]
```

CHAPTER 4

Control 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`(num, den[, dt])	A class for representing transfer functions
`StateSpace`(A, B, C, D[, dt])	A class for representing state-space models
`FrequencyResponseData`(d, w)	A class for models defined by frequency response data (FRD)
`InputOutputSystem`([inputs, outputs, states, . . .])	A class for representing input/output systems.

4.1 control.TransferFunction

class control.**TransferFunction**(*num*, *den*[, *dt*])

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.

The TransferFunction class defines two constants `s` and `z` that represent the differentiation and delay operators in continuous and discrete time. These can be used to create variables that allow algebraic creation of transfer functions. For example,

```
>>> s = TransferFunction.s
>>> G  = (s + 1)/(s**2 + 2*s + 1)
```

__init__(**args*)

TransferFunction(num, den[, dt])

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 create a discrete time transfer funtion, use TransferFunction(num, den, dt) where 'dt' is the sampling time (or True for unspecified sampling time). To call the copy constructor, call TransferFunction(sys), where sys is a TransferFunction object (continuous or discrete).

Methods

`__init__`(*args)	TransferFunction(num, den[, dt])
`damp`()	Natural frequency, damping ratio of system poles
`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`()	Check to see if a system is single input, single output
`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.

Attributes

`s`
`z`

damp()

Natural frequency, damping ratio of system poles

Returns

- **wn** (*array*) – Natural frequencies for each system pole
- **zeta** (*array*) – Damping ratio for each system pole
- **poles** (*array*) – Array of system poles

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** – The zero-frequency gain

Return type ndarray

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, optional*) – If strict is True, make sure that timebase is not None. Default is False.

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

Parameters **strict** (*bool, optional*) – If strict is True, make sure that timebase is not None. Default is False.

issiso()
Check to see if a system is single input, single output

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"} Method to use for sampling:
  - gbt: generalized bilinear transformation
  - bilinear: Tustin's approximation ("gbt" with alpha=0.5)
  - euler: Euler (or forward difference) method ("gbt" with alpha=0)
  - backward_diff: Backwards difference ("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** – Discrete time system, with sampling rate Ts

Return type StateSpace system

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.

4.2 control.StateSpace

class control.**StateSpace**(*A*, *B*, *C*, *D*[, *dt*])

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). The data format used to store state space matrices is set using the value of *config.defaults['use_numpy_matrix']*. If True (default), the state space elements are stored as *numpy.matrix* objects; otherwise they are *numpy.ndarray* objects. The *use_numpy_matrix()* function can be used to set the storage type.

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*, ***kw*)

StateSpace(A, B, C, D[, dt])

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 create a discrete time system, use StateSpace(A, B, C, D, dt) where 'dt' is the sampling time (or True for unspecified sampling time). To call the copy constructor, call StateSpace(sys), where sys is a StateSpace object.

Methods

`__init__`(*args, **kw)	StateSpace(A, B, C, D[, dt])
`append`(other)	Append a second model to the present model.
`damp`()	Natural frequency, damping ratio of system poles
`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`()	Check to see if a system is single input, single output
`lft`(other[, nu, ny])	Return the Linear Fractional Transformation.
`minreal`([tol])	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.
`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

damp()

Natural frequency, damping ratio of system poles

Returns

- **wn** (*array*) – Natural frequencies for each system pole
- **zeta** (*array*) – Damping ratio for each system pole
- **poles** (*array*) – Array of system poles

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 – 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.

Return type ndarray

`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 freqs, omega.

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

Reports the frequency response of the system,

G(j*omega) = mag*exp(j*phase)

for continuous time. For discrete time systems, the response is evaluated around the unit circle such that

G(exp(j*omega*dt)) = mag*exp(j*phase).

Parameters `omega` (*`array`*) – A list of frequencies in radians/sec at which the system should be evaluated. The list can be either a python list or a numpy array and will be sorted before evaluation.

Returns

- **mag** (*float*) – The magnitude (absolute value, not dB or log10) of the system frequency response.
- **phase** (*float*) – The wrapped phase in radians of the system frequency response.
- **omega** (*array*) – The list of sorted frequencies at which the response was evaluated.

`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, optional`*) – If strict is True, make sure that timebase is not None. Default is False.

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

Parameters `strict` (*`bool, optional`*) – If strict is True, make sure that timebase is not None. Default is False.

`issiso`()
Check to see if a system is single input, single output

lft (*other*, *nu=-1*, *ny=-1*)
Return the Linear Fractional Transformation.

A definition of the LFT operator can be found in Appendix A.7, page 512 in the 2nd Edition, Multivariable Feedback Control by Sigurd Skogestad.

An alternative definition can be found here: https://www.mathworks.com/help/control/ref/lft.html

Parameters

- **other** (*LTI*) – The lower LTI system
- **ny** (*int, optional*) – Dimension of (plant) measurement output.
- **nu** (*int, optional*) – Dimension of (plant) control input.

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** – Discrete time system, with sampling rate Ts

Return type *StateSpace*

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.

4.3 control.FrequencyResponseData

class control.**FrequencyResponseData**(*d*, *w*)
A class for models defined by frequency response data (FRD)

The FrequencyResponseData (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`()	Natural frequency, damping ratio of system poles
`dcgain`()	Return the zero-frequency gain
`eval`(omega)	Evaluate a transfer function at a single angular frequency.
`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.

Continued on next page

Table 5 – continued from previous page

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()	Check to see if a system is single input, single output

Attributes

epsw

damp()
Natural frequency, damping ratio of system poles

Returns

- **wn** (*array*) – Natural frequencies for each system pole
- **zeta** (*array*) – Damping ratio for each system pole
- **poles** (*array*) – Array of system poles

dcgain()
Return the zero-frequency gain

eval(*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.

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, optional*) – If strict is True, make sure that timebase is not None. Default is False.

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

Parameters **strict** (*bool, optional*) – If strict is True, make sure that timebase is not None. Default is False.

issiso()
Check to see if a system is single input, single output

4.4 control.InputOutputSystem

class control.**InputOutputSystem**(*inputs=None*, *outputs=None*, *states=None*, *params={}*, *dt=None*, *name=None*)
A class for representing input/output systems.

The InputOutputSystem class allows (possibly nonlinear) input/output systems to be represented in Python. It is intended as a parent class for a set of subclasses that are used to implement specific structures and operations for different types of input/output dynamical systems.

Parameters

- **inputs** (*int, list of str, or None*) – Description of the system inputs. This can be given as an integer count or as a list of strings that name the individual signals. If an integer count is specified, the names of the signal will be of the form *s[i]* (where *s* is one of *u*, *y*, or *x*). If this parameter is not given or given as *None*, the relevant quantity will be determined when possible based on other information provided to functions using the system.
- **outputs** (*int, list of str, or None*) – Description of the system outputs. Same format as *inputs*.
- **states** (*int, list of str, or None*) – Description of the system states. Same format as *inputs*.
- **dt** (*None, True or float, optional*) – System timebase. None (default) indicates continuous time, True indicates discrete time with undefined sampling time, positive number is discrete time with specified sampling time.
- **params** (*dict, optional*) – Parameter values for the systems. Passed to the evaluation functions for the system as default values, overriding internal defaults.
- **name** (*string, optional*) – System name (used for specifying signals)

ninputs, noutputs, nstates
Number of input, output and state variables

Type int

input_index, output_index, state_index
Dictionary of signal names for the inputs, outputs and states and the index of the corresponding array

Type dict

dt
System timebase. None (default) indicates continuous time, True indicates discrete time with undefined sampling time, positive number is discrete time with specified sampling time.

Type None, True or float

params
Parameter values for the systems. Passed to the evaluation functions for the system as default values, overriding internal defaults.

Type dict, optional

name
System name (used for specifying signals)

Type string, optional

Notes

The *InputOuputSystem* class (and its subclasses) makes use of two special methods for implementing much of the work of the class:

- _rhs(t, x, u): compute the right hand side of the differential or difference equation for the system. This must be specified by the subclass for the system.
- _out(t, x, u): compute the output for the current state of the system. The default is to return the entire system state.

__init__(*inputs=None*, *outputs=None*, *states=None*, *params={}*, *dt=None*, *name=None*)
Create an input/output system.

The InputOutputSystem contructor is used to create an input/output object with the core information required for all input/output systems. Instances of this class are normally created by one of the input/output subclasses: `LinearIOSystem`, `NonlinearIOSystem`, `InterconnectedSystem`.

Parameters

- **inputs** (*int, list of str, or None*) – Description of the system inputs. This can be given as an integer count or as a list of strings that name the individual signals. If an integer count is specified, the names of the signal will be of the form *s[i]* (where *s* is one of *u*, *y*, or *x*). If this parameter is not given or given as *None*, the relevant quantity will be determined when possible based on other information provided to functions using the system.
- **outputs** (*int, list of str, or None*) – Description of the system outputs. Same format as *inputs*.
- **states** (*int, list of str, or None*) – Description of the system states. Same format as *inputs*.
- **dt** (*None, True or float, optional*) – System timebase. None (default) indicates continuous time, True indicates discrete time with undefined sampling time, positive number is discrete time with specified sampling time.
- **params** (*dict, optional*) – Parameter values for the systems. Passed to the evaluation functions for the system as default values, overriding internal defaults.
- **name** (*string, optional*) – System name (used for specifying signals)

Returns Input/output system object

Return type *InputOutputSystem*

Methods

`__init__`([inputs, outputs, states, params, . . .])	Create an input/output system.
`copy`()	Make a copy of an input/output system.
`feedback`([other, sign, params])	Feedback interconnection between two input/output systems

Continued on next page

Table 7 – continued from previous page

`find_input`(name)	Find the index for an input given its name (*None* if not found)
`find_output`(name)	Find the index for an output given its name (*None* if not found)
`find_state`(name)	Find the index for a state given its name (*None* if not found)
`linearize`(x0, u0[, t, params, eps])	Linearize an input/output system at a given state and input.
`set_inputs`(inputs[, prefix])	Set the number/names of the system inputs.
`set_outputs`(outputs[, prefix])	Set the number/names of the system outputs.
`set_states`(states[, prefix])	Set the number/names of the system states.

copy()
Make a copy of an input/output system.

feedback(*other=1*, *sign=-1*, *params={}*)
Feedback interconnection between two input/output systems

Parameters

- **sys1** (`InputOutputSystem`) – The primary process.
- **sys2** (`InputOutputSystem`) – The feedback process (often a feedback controller).
- **sign** (`scalar, optional`) – 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

Return type *InputOutputSystem*

Raises `ValueError` – if the inputs, outputs, or timebases of the systems are incompatible.

find_input(*name*)
Find the index for an input given its name (*None* if not found)

find_output(*name*)
Find the index for an output given its name (*None* if not found)

find_state(*name*)
Find the index for a state given its name (*None* if not found)

linearize(*x0*, *u0*, *t=0*, *params={}*, *eps=1e-06*)
Linearize an input/output system at a given state and input.

Return the linearization of an input/output system at a given state and input value as a StateSpace system. See `linearize()` for complete documentation.

set_inputs(*inputs*, *prefix='u'*)
Set the number/names of the system inputs.

Parameters

- **inputs** (`int, list of str, or None`) – Description of the system inputs. This can be given as an integer count or as a list of strings that name the individual signals. If an integer count is specified, the names of the signal will be of the form *u[i]* (where the prefix *u* can be changed using the optional prefix parameter).
- **prefix** (`string, optional`) – If *inputs* is an integer, create the names of the states using the given prefix (default = 'u'). The names of the input will be of the form *prefix[i]*.

set_outputs (*outputs*, *prefix='y'*)
Set the number/names of the system outputs.

Parameters

- **outputs** (*int, list of str, or None*) – Description of the system outputs. This can be given as an integer count or as a list of strings that name the individual signals. If an integer count is specified, the names of the signal will be of the form *u[i]* (where the prefix *u* can be changed using the optional prefix parameter).
- **prefix** (*string, optional*) – If *outputs* is an integer, create the names of the states using the given prefix (default = 'y'). The names of the input will be of the form *prefix[i]*.

set_states (*states*, *prefix='x'*)
Set the number/names of the system states.

Parameters

- **states** (*int, list of str, or None*) – Description of the system states. This can be given as an integer count or as a list of strings that name the individual signals. If an integer count is specified, the names of the signal will be of the form *u[i]* (where the prefix *u* can be changed using the optional prefix parameter).
- **prefix** (*string, optional*) – If *states* is an integer, create the names of the states using the given prefix (default = 'x'). The names of the input will be of the form *prefix[i]*.

4.5 Input/Output system subclasses

Input/output systems are accessed primarily via a set of subclasses that allow for linear, nonlinear, and interconnected elements:

LinearIOSystem(linsys[, inputs, outputs, . . .])	Input/output representation of a linear (state space) system.
NonlinearIOSystem(updfcn[, outfcn, inputs, . . .])	Nonlinear I/O system.
InterconnectedSystem(syslist[, connections, . . .])	Interconnection of a set of input/output systems.

4.5.1 control.LinearIOSystem

class control.**LinearIOSystem** (*linsys*, *inputs=None*, *outputs=None*, *states=None*, *name=None*)
Input/output representation of a linear (state space) system.

This class is used to implementat a system that is a linear state space system (defined by the StateSpace system object).

__init__ (*linsys*, *inputs=None*, *outputs=None*, *states=None*, *name=None*)
Create an I/O system from a state space linear system.

Converts a StateSpace system into an InputOutputSystem with the same inputs, outputs, and states. The new system can be a continuous or discrete time system

Parameters

- **linsys** (StateSpace) – LTI StateSpace system to be converted
- **inputs** (*int, list of str or None, optional*) – Description of the system inputs. This can be given as an integer count or as a list of strings that name the

individual signals. If an integer count is specified, the names of the signal will be of the form *s[i]* (where *s* is one of *u*, *y*, or *x*). If this parameter is not given or given as *None*, the relevant quantity will be determined when possible based on other information provided to functions using the system.

- **outputs** (*int, list of str or None, optional*) – Description of the system outputs. Same format as *inputs*.
- **states** (*int, list of str, or None, optional*) – Description of the system states. Same format as *inputs*.
- **dt** (*None, True or float, optional*) – System timebase. None (default) indicates continuous time, True indicates discrete time with undefined sampling time, positive number is discrete time with specified sampling time.
- **params** (*dict, optional*) – Parameter values for the systems. Passed to the evaluation functions for the system as default values, overriding internal defaults.
- **name** (*string, optional*) – System name (used for specifying signals)

Returns **iosys** – Linear system represented as an input/output system

Return type *LinearIOSystem*

Methods

`__init__`(linsys[, inputs, outputs, states, name])	Create an I/O system from a state space linear system.
`append`(other)	Append a second model to the present model.
`copy`()	Make a copy of an input/output system.
`damp`()	Natural frequency, damping ratio of system poles
`dcgain`()	Return the zero-frequency gain
`evalfr`(omega)	Evaluate a SS system's transfer function at a single frequency.
`feedback`([other, sign, params])	Feedback interconnection between two input/output systems
`find_input`(name)	Find the index for an input given its name (*None* if not found)
`find_output`(name)	Find the index for an output given its name (*None* if not found)
`find_state`(name)	Find the index for a state given its name (*None* if not found)
`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`()	Check to see if a system is single input, single output
`lft`(other[, nu, ny])	Return the Linear Fractional Transformation.
`linearize`(x0, u0[, t, params, eps])	Linearize an input/output system at a given state and input.
`minreal`([tol])	Calculate a minimal realization, removes unobservable and uncontrollable states
`pole`()	Compute the poles of a state space system.

Continued on next page

Table 9 – continued from previous page

`returnScipySignalLTI`()	Return a list of a list of scipy.signal.lti objects.
`sample`(Ts[, method, alpha])	Convert a continuous time system to discrete time
`set_inputs`(inputs[, prefix])	Set the number/names of the system inputs.
`set_outputs`(outputs[, prefix])	Set the number/names of the system outputs.
`set_states`(states[, prefix])	Set the number/names of the system states.
`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

copy()
Make a copy of an input/output system.

damp()
Natural frequency, damping ratio of system poles

Returns

- **wn** (*array*) – Natural frequencies for each system pole
- **zeta** (*array*) – Damping ratio for each system pole
- **poles** (*array*) – Array of system poles

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 – 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.

Return type ndarray

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*, *params={}*)
Feedback interconnection between two input/output systems

Parameters

- **sys1** (`InputOutputSystem`) – The primary process.
- **sys2** (`InputOutputSystem`) – The feedback process (often a feedback controller).
- **sign** (*scalar, optional*) – 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

Return type *InputOutputSystem*

Raises `ValueError` – if the inputs, outputs, or timebases of the systems are incompatible.

find_input(*name*)
Find the index for an input given its name (*None* if not found)

find_output (*name*)
Find the index for an output given its name (*None* if not found)

find_state (*name*)
Find the index for a state given its name (*None* if not found)

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

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

Reports the frequency response of the system,

G(j*omega) = mag*exp(j*phase)

for continuous time. For discrete time systems, the response is evaluated around the unit circle such that

G(exp(j*omega*dt)) = mag*exp(j*phase).

Parameters **omega** (*array*) – A list of frequencies in radians/sec at which the system should be evaluated. The list can be either a python list or a numpy array and will be sorted before evaluation.

Returns

- **mag** (*float*) – The magnitude (absolute value, not dB or log10) of the system frequency response.
- **phase** (*float*) – The wrapped phase in radians of the system frequency response.
- **omega** (*array*) – The list of sorted frequencies at which the response was evaluated.

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, optional*) – If strict is True, make sure that timebase is not None. Default is False.

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

Parameters **strict** (*bool, optional*) – If strict is True, make sure that timebase is not None. Default is False.

issiso ()
Check to see if a system is single input, single output

lft (*other, nu=-1, ny=-1*)
Return the Linear Fractional Transformation.

A definition of the LFT operator can be found in Appendix A.7, page 512 in the 2nd Edition, Multivariable Feedback Control by Sigurd Skogestad.

An alternative definition can be found here: https://www.mathworks.com/help/control/ref/lft.html

Parameters

- **other** (*LTI*) – The lower LTI system
- **ny** (*int, optional*) – Dimension of (plant) measurement output.
- **nu** (*int, optional*) – Dimension of (plant) control input.

linearize (*x0, u0, t=0, params={}, eps=1e-06*)
Linearize an input/output system at a given state and input.

Return the linearization of an input/output system at a given state and input value as a StateSpace system. See `linearize()` for complete documentation.

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** – Discrete time system, with sampling rate Ts

Return type StateSpace

Notes

Uses the command 'cont2discrete' from scipy.signal

Examples

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

set_inputs(*inputs*, *prefix='u'*)

Set the number/names of the system inputs.

Parameters

- **inputs** (*int, list of str, or None*) – Description of the system inputs. This can be given as an integer count or as a list of strings that name the individual signals. If an integer count is specified, the names of the signal will be of the form *u[i]* (where the prefix *u* can be changed using the optional prefix parameter).
- **prefix** (*string, optional*) – If *inputs* is an integer, create the names of the states using the given prefix (default = 'u'). The names of the input will be of the form *prefix[i]*.

set_outputs(*outputs*, *prefix='y'*)

Set the number/names of the system outputs.

Parameters

- **outputs** (*int, list of str, or None*) – Description of the system outputs. This can be given as an integer count or as a list of strings that name the individual signals. If an integer count is specified, the names of the signal will be of the form *u[i]* (where the prefix *u* can be changed using the optional prefix parameter).
- **prefix** (*string, optional*) – If *outputs* is an integer, create the names of the states using the given prefix (default = 'y'). The names of the input will be of the form *prefix[i]*.

set_states(*states*, *prefix='x'*)

Set the number/names of the system states.

Parameters

- **states** (*int, list of str, or None*) – Description of the system states. This can be given as an integer count or as a list of strings that name the individual signals. If an integer count is specified, the names of the signal will be of the form *u[i]* (where the prefix *u* can be changed using the optional prefix parameter).
- **prefix** (*string, optional*) – If *states* is an integer, create the names of the states using the given prefix (default = 'x'). The names of the input will be of the form *prefix[i]*.

zero()

Compute the zeros of a state space system.

4.5.2 control.NonlinearIOSystem

class control.**NonlinearIOSystem**(*updfcn*, *outfcn=None*, *inputs=None*, *outputs=None*, *states=None*, *params={}*, *dt=None*, *name=None*)

Nonlinear I/O system.

This class is used to implement a system that is a nonlinear state space system (defined by and update function and an output function).

__init__(*updfcn*, *outfcn=None*, *inputs=None*, *outputs=None*, *states=None*, *params={}*, *dt=None*, *name=None*)

Create a nonlinear I/O system given update and output functions.

Creates an *InputOutputSystem* for a nonlinear system by specifying a state update function and an output function. The new system can be a continuous or discrete time system (Note: discrete-time systems not yet supported by most function.)

Parameters

- **updfcn** (*callable*) – Function returning the state update function

  updfcn(t, x, u[, param]) -> array

  where *x* is a 1-D array with shape (nstates,), *u* is a 1-D array with shape (ninputs,), *t* is a float representing the currrent time, and *param* is an optional dict containing the values of parameters used by the function.

- **outfcn** (*callable*) – Function returning the output at the given state

  outfcn(t, x, u[, param]) -> array

  where the arguments are the same as for *upfcn*.

- **inputs** (*int, list of str or None, optional*) – Description of the system inputs. This can be given as an integer count or as a list of strings that name the individual signals. If an integer count is specified, the names of the signal will be of the form *s[i]* (where *s* is one of *u*, *y*, or *x*). If this parameter is not given or given as *None*, the relevant quantity will be determined when possible based on other information provided to functions using the system.

- **outputs** (*int, list of str or None, optional*) – Description of the system outputs. Same format as *inputs*.

- **states** (*int, list of str, or None, optional*) – Description of the system states. Same format as *inputs*.

- **params** (*dict, optional*) – Parameter values for the systems. Passed to the evaluation functions for the system as default values, overriding internal defaults.

- **dt** (*timebase, optional*) – The timebase for the system, used to specify whether the system is operating in continuous or discrete time. It can have the following values:
  - dt = None No timebase specified
  - dt = 0 Continuous time system
  - dt > 0 Discrete time system with sampling time dt
  - dt = True Discrete time with unspecified sampling time

- **name** (*string, optional*) – System name (used for specifying signals).

Returns **iosys** – Nonlinear system represented as an input/output system.

Return type *NonlinearIOSystem*

Methods

__init__(updfcn[, outfcn, inputs, outputs, . . .])	Create a nonlinear I/O system given update and output functions.
copy()	Make a copy of an input/output system.
feedback([other, sign, params])	Feedback interconnection between two input/output systems

Continued on next page

Table 10 – continued from previous page

find_input(name)	Find the index for an input given its name (*None* if not found)
find_output(name)	Find the index for an output given its name (*None* if not found)
find_state(name)	Find the index for a state given its name (*None* if not found)
linearize(x0, u0[, t, params, eps])	Linearize an input/output system at a given state and input.
set_inputs(inputs[, prefix])	Set the number/names of the system inputs.
set_outputs(outputs[, prefix])	Set the number/names of the system outputs.
set_states(states[, prefix])	Set the number/names of the system states.

copy()
Make a copy of an input/output system.

feedback(*other=1*, *sign=-1*, *params={}*)
Feedback interconnection between two input/output systems

Parameters

- **sys1** (InputOutputSystem) – The primary process.
- **sys2** (InputOutputSystem) – The feedback process (often a feedback controller).
- **sign** (*scalar, optional*) – 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

Return type *InputOutputSystem*

Raises ValueError – if the inputs, outputs, or timebases of the systems are incompatible.

find_input(*name*)
Find the index for an input given its name (*None* if not found)

find_output(*name*)
Find the index for an output given its name (*None* if not found)

find_state(*name*)
Find the index for a state given its name (*None* if not found)

linearize(*x0*, *u0*, *t=0*, *params={}*, *eps=1e-06*)
Linearize an input/output system at a given state and input.

Return the linearization of an input/output system at a given state and input value as a StateSpace system. See linearize() for complete documentation.

set_inputs(*inputs*, *prefix='u'*)
Set the number/names of the system inputs.

Parameters

- **inputs** (*int, list of str, or None*) – Description of the system inputs. This can be given as an integer count or as a list of strings that name the individual signals. If an integer count is specified, the names of the signal will be of the form *u[i]* (where the prefix *u* can be changed using the optional prefix parameter).
- **prefix** (*string, optional*) – If *inputs* is an integer, create the names of the states using the given prefix (default = 'u'). The names of the input will be of the form *prefix[i]*.

set_outputs (*outputs*, *prefix='y'*)
Set the number/names of the system outputs.

Parameters

- **outputs** (*int, list of str, or None*) – Description of the system outputs. This can be given as an integer count or as a list of strings that name the individual signals. If an integer count is specified, the names of the signal will be of the form *u[i]* (where the prefix *u* can be changed using the optional prefix parameter).
- **prefix** (*string, optional*) – If *outputs* is an integer, create the names of the states using the given prefix (default = 'y'). The names of the input will be of the form *prefix[i]*.

set_states (*states*, *prefix='x'*)
Set the number/names of the system states.

Parameters

- **states** (*int, list of str, or None*) – Description of the system states. This can be given as an integer count or as a list of strings that name the individual signals. If an integer count is specified, the names of the signal will be of the form *u[i]* (where the prefix *u* can be changed using the optional prefix parameter).
- **prefix** (*string, optional*) – If *states* is an integer, create the names of the states using the given prefix (default = 'x'). The names of the input will be of the form *prefix[i]*.

4.5.3 control.InterconnectedSystem

class control.**InterconnectedSystem** (*syslist*, *connections=[]*, *inplist=[]*, *outlist=[]*, *inputs=None*, *outputs=None*, *states=None*, *params={}*, *dt=None*, *name=None*)
Interconnection of a set of input/output systems.

This class is used to implement a system that is an interconnection of input/output systems. The sys consists of a collection of subsystems whose inputs and outputs are connected via a connection map. The overall system inputs and outputs are subsets of the subsystem inputs and outputs.

__init__ (*syslist*, *connections=[]*, *inplist=[]*, *outlist=[]*, *inputs=None*, *outputs=None*, *states=None*, *params={}*, *dt=None*, *name=None*)
Create an I/O system from a list of systems + connection info.

The InterconnectedSystem class is used to represent an input/output system that consists of an interconnection between a set of subystems. The outputs of each subsystem can be summed together to to provide inputs to other subsystems. The overall system inputs and outputs can be any subset of subsystem inputs and outputs.

Parameters

- **syslist** (*array_like of InputOutputSystems*) – The list of input/output systems to be connected
- **connections** (*tuple of connection specifications, optional*) – Description of the internal connections between the subsystems. Each element of the tuple describes an input to one of the subsystems. The entries are are of the form:

  (input-spec, output-spec1, output-spec2, . . .)

  The input-spec should be a tuple of the form *(subsys_i, inp_j)* where *subsys_i* is the index into *syslist* and *inp_j* is the index into the input vector for the subsystem. If *subsys_i* has a single input, then the subsystem index *subsys_i* can be listed as the input-spec. If systems and signals are given names, then the form 'sys.sig' or ('sys', 'sig') are also recognized.

Each output-spec should be a tuple of the form *(subsys_i, out_j, gain)*. The input will be constructed by summing the listed outputs after multiplying by the gain term. If the gain term is omitted, it is assumed to be 1. If the system has a single output, then the subsystem index *subsys_i* can be listed as the input-spec. If systems and signals are given names, then the form 'sys.sig', ('sys', 'sig') or ('sys', 'sig', gain) are also recognized, and the special form '-sys.sig' can be used to specify a signal with gain -1.

If omitted, the connection map (matrix) can be specified using the `set_connect_map()` method.

- **inplist** (*tuple of input specifications, optional*) – List of specifications for how the inputs for the overall system are mapped to the subsystem inputs. The input specification is the same as the form defined in the connection specification. Each system input is added to the input for the listed subsystem.

  If omitted, the input map can be specified using the *set_input_map* method.

- **outlist** (*tuple of output specifications, optional*) – List of specifications for how the outputs for the subsystems are mapped to overall system outputs. The output specification is the same as the form defined in the connection specification (including the optional gain term). Numbered outputs must be chosen from the list of subsystem outputs, but named outputs can also be contained in the list of subsystem inputs.

  If omitted, the output map can be specified using the *set_output_map* method.

- **params** (*dict, optional*) – Parameter values for the systems. Passed to the evaluation functions for the system as default values, overriding internal defaults.
- **dt** (*timebase, optional*) – The timebase for the system, used to specify whether the system is operating in continuous or discrete time. It can have the following values:
  - dt = None No timebase specified
  - dt = 0 Continuous time system
  - dt > 0 Discrete time system with sampling time dt
  - dt = True Discrete time with unspecified sampling time
- **name** (*string, optional*) – System name (used for specifying signals).

Methods

`__init__`(syslist[, connections, inplist, . . .])	Create an I/O system from a list of systems + connection info.
`copy`()	Make a copy of an input/output system.
`feedback`([other, sign, params])	Feedback interconnection between two input/output systems
`find_input`(name)	Find the index for an input given its name (*None* if not found)
`find_output`(name)	Find the index for an output given its name (*None* if not found)
`find_state`(name)	Find the index for a state given its name (*None* if not found)
`linearize`(x0, u0[, t, params, eps])	Linearize an input/output system at a given state and input.

Continued on next page

Table 11 – continued from previous page

`set_connect_map`(connect_map)	Set the connection map for an interconnected I/O system.
`set_input_map`(input_map)	Set the input map for an interconnected I/O system.
`set_inputs`(inputs[, prefix])	Set the number/names of the system inputs.
`set_output_map`(output_map)	Set the output map for an interconnected I/O system.
`set_outputs`(outputs[, prefix])	Set the number/names of the system outputs.
`set_states`(states[, prefix])	Set the number/names of the system states.

copy()
Make a copy of an input/output system.

feedback(*other=1*, *sign=-1*, *params={}*)
Feedback interconnection between two input/output systems

Parameters

- **sys1** (`InputOutputSystem`) – The primary process.
- **sys2** (`InputOutputSystem`) – The feedback process (often a feedback controller).
- **sign** (*scalar, optional*) – 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

Return type *InputOutputSystem*

Raises `ValueError` – if the inputs, outputs, or timebases of the systems are incompatible.

find_input(*name*)
Find the index for an input given its name (*None* if not found)

find_output(*name*)
Find the index for an output given its name (*None* if not found)

find_state(*name*)
Find the index for a state given its name (*None* if not found)

linearize(*x0*, *u0*, *t=0*, *params={}*, *eps=1e-06*)
Linearize an input/output system at a given state and input.

Return the linearization of an input/output system at a given state and input value as a StateSpace system. See `linearize()` for complete documentation.

set_connect_map(*connect_map*)
Set the connection map for an interconnected I/O system.

Parameters **connect_map** (*2D array*) – Specify the matrix that will be used to multiply the vector of subsystem outputs to obtain the vector of subsystem inputs.

set_input_map(*input_map*)
Set the input map for an interconnected I/O system.

Parameters **input_map** (*2D array*) – Specify the matrix that will be used to multiply the vector of system inputs to obtain the vector of subsystem inputs. These values are added to the inputs specified in the connection map.

set_inputs(*inputs*, *prefix='u'*)
Set the number/names of the system inputs.

Parameters

- **inputs** (*int, list of str, or None*) – Description of the system inputs. This can be given as an integer count or as a list of strings that name the individual signals. If an integer count is specified, the names of the signal will be of the form *u[i]* (where the prefix *u* can be changed using the optional prefix parameter).
- **prefix** (*string, optional*) – If *inputs* is an integer, create the names of the states using the given prefix (default = 'u'). The names of the input will be of the form *prefix[i]*.

set_output_map (*output_map*)

Set the output map for an interconnected I/O system.

Parameters **output_map** (*2D array*) – Specify the matrix that will be used to multiply the vector of subsystem outputs to obtain the vector of system outputs.

set_outputs (*outputs*, *prefix='y'*)

Set the number/names of the system outputs.

Parameters

- **outputs** (*int, list of str, or None*) – Description of the system outputs. This can be given as an integer count or as a list of strings that name the individual signals. If an integer count is specified, the names of the signal will be of the form *u[i]* (where the prefix *u* can be changed using the optional prefix parameter).
- **prefix** (*string, optional*) – If *outputs* is an integer, create the names of the states using the given prefix (default = 'y'). The names of the input will be of the form *prefix[i]*.

set_states (*states*, *prefix='x'*)

Set the number/names of the system states.

Parameters

- **states** (*int, list of str, or None*) – Description of the system states. This can be given as an integer count or as a list of strings that name the individual signals. If an integer count is specified, the names of the signal will be of the form *u[i]* (where the prefix *u* can be changed using the optional prefix parameter).
- **prefix** (*string, optional*) – If *states* is an integer, create the names of the states using the given prefix (default = 'x'). The names of the input will be of the form *prefix[i]*.

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).

5.1 Creating linear models

`tf`(num, den[, dt])	Create a transfer function system.
`ss`(A, B, C, D[, dt])	Create a state space system.
`frd`(d, w)	Construct a frequency response data model
`rss`([states, outputs, inputs])	Create a stable *continuous* random state space object.
`drss`([states, outputs, inputs])	Create a stable *discrete* random state space object.

5.1.1 control.matlab.tf

`control.matlab.`**`tf`**`(`*num*, *den*[, *dt*]`)`

Create a transfer function system. Can create MIMO systems.

The function accepts either 1, 2, or 3 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.

tf('s') or tf('z') Create a transfer function representing the differential operator ('s') or delay operator ('z').

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** – The new linear system

Return type TransferFunction

Raises

- ValueError – if *num* and *den* have invalid or unequal dimensions
- TypeError – if *num* or *den* are of incorrect type

See also:

TransferFunction(), *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 $2s^2 + 3s + 4$.

The special forms tf('s') and tf('z') can be used to create transfer functions for differentiation and unit delays.

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)
```

```
>>> # Create a variable 's' to allow algebra operations for SISO systems
>>> s = tf('s')
>>> G  = (s + 1)/(s**2 + 2*s + 1)
```

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

5.1.2 control.matlab.ss

`control.matlab.ss` (*A*, *B*, *C*, *D*[, *dt*])

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** – The new linear system

Return type `StateSpace`

Raises `ValueError` – if matrix sizes are not self-consistent

See also:

`StateSpace()`, *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)
```

5.1.3 control.matlab.frd

control.matlab.**frd**(*d*, *w*)

Construct a frequency response data model

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** – New frequency response system

Return type FRD

See also:

FRD(), *ss()*, *tf()*

5.1.4 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** – The randomly created linear system

Return type *StateSpace*

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.

5.1.5 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** – The randomly created linear system

Return type *StateSpace*

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.

5.2 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(sys)	Transform a state space system to a transfer function.
tf2ss(sys)	Transform a transfer function to a state space system.
tfdata(sys)	Return transfer function data objects for a system

5.2.1 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** – corresponding values in decibels

Return type float or ndarray

5.2.2 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** – corresponding magnitudes

Return type float or ndarray

5.2.3 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

5.2.4 control.matlab.ss2tf

`control.matlab.ss2tf(sys)`

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** – New linear system in transfer function form

Return type *TransferFunction*

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)
```

5.2.5 control.matlab.tf2ss

control.matlab.**tf2ss**(*sys*)

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** – New linear system in state space form

Return type *StateSpace*

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)
```

5.2.6 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)** – Transfer function coefficients (SISO only)

Return type numerator and denominator arrays

5.3 System interconnections

series(sys1, *sysn)	Return the series connection (…
parallel(sys1, *sysn)	Return the parallel connection sys1 + sys2 (+ sys3 + …)
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-based interconnection of an LTI system.
append(sys1, sys2, …, sysn)	Group models by appending their inputs and outputs

5.3.1 control.matlab.series

control.matlab.**series**(*sys1*, **sysn*)

Return the series connection (… * sys3 *) sys2 * sys1

Parameters

- **sys1** (*scalar, StateSpace, TransferFunction, or FRD*) –
- **sysn** (*other scalars, StateSpaces, TransferFunctions, or FRDs*) –

Returns **out**

Return type 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
```

```
>>> sys5 = series(sys1, sys2, sys3, sys4) # More systems
```

5.3.2 control.matlab.parallel

`control.matlab.parallel(sys1, *sysn)`

Return the parallel connection sys1 + sys2 (+ sys3 + . . .)

Parameters

- **sys1** (*scalar, StateSpace, TransferFunction, or FRD*) –
- ***sysn** (*other scalars, StateSpaces, TransferFunctions, or FRDs*) –

Returns out

Return type 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
```

```
>>> sys5 = parallel(sys1, sys2, sys3, sys4) # More systems
```

5.3.3 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 process.
- **sys2** (*scalar, StateSpace, TransferFunction, FRD*) – The feedback process (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

Return type *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.

5.3.4 control.matlab.negate

`control.matlab.negate(sys)`

Return the negative of a system.

Parameters **sys** (*StateSpace, TransferFunction or FRD*) –

Returns out

Return type *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.

Examples

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

5.3.5 control.matlab.connect

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

Index-based interconnection of an LTI 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: Inputs and outputs are indexed starting at 1 and negative values correspond to a negative feedback interconnection.

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. Inputs and outputs are indexed starting at 1.
- **inputv** (*1D array*) – list of final external inputs
- **outputv** (*1D array*) – list of final external outputs

Returns **sys** – Connected and trimmed LTI system

Return type 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 = [[1, 2], [2, -1]]  # negative feedback interconnection
>>> sysc = connect(sys, Q, [2], [1, 2])
```

5.3.6 control.matlab.append

control.matlab.**append**(*sys1*, *sys2*, ..., *sysn*)

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 **sys2, .. sysn** (*sys1,*) – LTI systems to combine

Returns **sys** – Combined LTI system, with input/output vectors consisting of all input/output vectors appended

Return type LTI system

Examples

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

5.4 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, grid, title])	Plot a pole/zero map for a linear system.

5.4.1 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

- **B, C, D** (*A,*) – A linear system in state space form.
- **P, k** (*Z,*) – A linear system in zero, pole, gain form.
- **den** (*num,*) – A linear system in transfer function form.
- **sys** (*LTI (StateSpace or TransferFunction)*) – A linear system object.

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

Return type ndarray

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:

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

5.4.2 control.matlab.pole

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

Compute system poles.

Parameters **sys** (StateSpace or TransferFunction) – Linear system

Returns **poles** – Array that contains the system's poles.

Return type ndarray

Raises `NotImplementedError` – when called on a TransferFunction object

See also:

zero(), `TransferFunction.pole()`, `StateSpace.pole()`

5.4.3 control.matlab.zero

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

Compute system zeros.

Parameters **sys** (`StateSpace` *or* `TransferFunction`) – Linear system

Returns **zeros** – Array that contains the system's zeros.

Return type ndarray

Raises `NotImplementedError` – when called on a MIMO system

See also:

pole(), `StateSpace.zero()`, `TransferFunction.zero()`

5.4.4 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
- *Algorithm*
- ———
- *If the system is continuous,* – wn = abs(poles) Z = -real(poles)/poles.
- *If the system is discrete, the discrete poles are mapped to their*
- *equivalent location in the s-plane via* – s = log10(poles)/dt
- *and* – wn = abs(s) Z = -real(s)/wn.

See also:

pole()

5.4.5 control.matlab.pzmap

`control.matlab.pzmap(sys, Plot=True, grid=False, 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.
- **grid** (*boolean (default = False)*) – If True plot omega-damping grid.

Returns

- **pole** (*array*) – The systems poles
- **zeros** (*array*) – The system's zeros.

5.5 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.

5.5.1 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)
```

5.5.2 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)
```

5.5.3 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)
```

5.5.4 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)
```

5.6 Frequency-domain analysis

bode(syslist[, omega, dB, Hz, deg, . . .])	Bode plot of the frequency response
nyquist(syslist[, omega, Plot, color, labelFreq])	Nyquist plot for a system
nichols(sys_list[, omega, grid])	Nichols plot for a system
margin(sysdata)	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.

5.6.1 control.matlab.bode

control.matlab.**bode**(*syslist*[, *omega*, *dB*, *Hz*, *deg*, *...*])

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')
  ```

5.6.2 control.matlab.nyquist

control.matlab.**nyquist**(*syslist*, *omega=None*, *Plot=True*, *color=None*, *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
- **color** (*string*) – Used to specify the color of the plot
- **labelFreq** (*int*) – Label every nth frequency on the plot
- ****kwargs** (**args,*) – 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)
```

5.6.3 control.matlab.nichols

control.matlab.**nichols**(*sys_list*, *omega=None*, *grid=None*)

Nichols plot for a system

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

Parameters

- **sys_list** (*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

Return type None

5.6.4 control.matlab.margin

`control.matlab.margin(sysdata)`

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)
- **wg** (*float*) – Frequency for gain margin (at phase crossover, phase = -180 degrees)
- **wp** (*float*) – Frequency for phase margin (at gain crossover, gain = 1)
- *Margins are calculated for a SISO open-loop system.*
- *If there is more than one gain crossover, the one at the smallest*
- *margin (deviation from gain = 1), in absolute sense, is*
- *returned. Likewise the smallest phase margin (in absolute sense)*
- *is returned.*

Examples

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

5.6.5 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.

5.6.6 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

Return type 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

5.7 Compensator design

rlocus(sys[, kvect, xlim, ylim, plotstr, . . .])	Root locus plot
sisotool(sys[, kvect, xlim_rlocus, . . .])	Sisotool style collection of plots inspired by MATLAB's sisotool.
place(A, B, p)	Place closed loop eigenvalues K = place(A, B, p)
lqr(A, B, Q, R[, N])	Linear quadratic regulator design

5.7.1 control.matlab.rlocus

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

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*) – Set limits of x axis, normally with tuple (see matplotlib.axes).
- **ylim** (*tuple or list, optional*) – Set limits of y axis, normally with tuple (see matplotlib.axes).
- **Plot** (*boolean, optional*) – If True (default), plot root locus diagram.
- **PrintGain** (*bool*) – If True (default), report mouse clicks when close to the root locus branches, calculate gain, damping and print.
- **grid** (*bool*) – If True plot omega-damping grid. Default is False.

Returns

- **rlist** (*ndarray*) – Computed root locations, given as a 2D array
- **klist** (*ndarray or list*) – Gains used. Same as klist keyword argument if provided.

5.7.2 control.matlab.sisotool

control.matlab.**sisotool**(*sys*, *kvect=None*, *xlim_rlocus=None*, *ylim_rlocus=None*, *plotstr_rlocus='C0'*, *rlocus_grid=False*, *omega=None*, *dB=None*, *Hz=None*, *deg=None*, *omega_limits=None*, *omega_num=None*, *margins_bode=True*, *tvect=None*)

Sisotool style collection of plots inspired by MATLAB's sisotool. The left two plots contain the bode magnitude and phase diagrams. The top right plot is a clickable root locus plot, clicking on the root locus will change the gain of the system. The bottom left plot shows a closed loop time response.

Parameters

- **sys** (*LTI object*) – Linear input/output systems (SISO only)
- **kvect** (*list or ndarray, optional*) – List of gains to use for plotting root locus
- **xlim_rlocus** (*tuple or list, optional*) – control of x-axis range, normally with tuple (see matplotlib.axes)
- **ylim_rlocus** (*tuple or list, optional*) – control of y-axis range
- **plotstr_rlocus** (*Additional options to matplotlib*) – plotting style for the root locus plot(color, linestyle, etc)
- **rlocus_grid** (*boolean (default = False)*) – If True plot s-plane grid.
- **omega** (*freq_range*) – Range of frequencies in rad/sec for the bode plot
- **dB** (*boolean*) – If True, plot result in dB for the bode plot
- **Hz** (*boolean*) – If True, plot frequency in Hz for the bode plot (omega must be provided in rad/sec)
- **deg** (*boolean*) – If True, plot phase in degrees for the bode plot (else radians)
- **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
- **margins_bode** (*boolean*) – If True, plot gain and phase margin in the bode plot
- **tvect** (*list or ndarray, optional*) – List of timesteps to use for closed loop step response

Examples

```
>>> sys = tf([1000], [1,25,100,0])
>>> sisotool(sys)
```

5.7.3 control.matlab.place

control.matlab.**place**(*A*, *B*, *p*)
Place closed loop eigenvalues K = place(A, B, p)

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*) – Gain such that A - B K has eigenvalues given in p
- *Algorithm*
- ———
- *This is a wrapper function for scipy.signal.place_poles, which*
- *implements the Tits and Yang algorithm [1]. It will handle SISO,*
- *MISO, and MIMO systems. If you want more control over the algorithm,*

- *use scipy.signal.place_poles directly.*
- *[1] A.L. Tits and Y. Yang, "Globally convergent algorithms for robust*
- *pole assignment by state feedback, IEEE Transactions on Automatic*
- *Control, Vol. 41, pp. 1432-1452, 1996.*
- *Limitations*
- ———
- *The algorithm will not place poles at the same location more*
- *than rank(B) times.*

Examples

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

See also:

`place_varga()`, `acker()`

5.7.4 control.matlab.lqr

`control.matlab.`**`lqr`**`(`*A*, *B*, *Q*, *R*[, *N*]`)`

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

- **B** (*A,*) – Dynamics and input matrices
- **sys** (*LTI (StateSpace or TransferFunction)*) – Linear I/O system
- **R** (*Q,*) – State and input weight matrices
- **N** (*2-d array, optional*) – Cross weight matrix

Returns

- **K** (*2D array*) – State feedback gains
- **S** (*2D array*) – Solution to Riccati equation
- **E** (*1D 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])
```

5.8 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.
ctrb(A, B)	Controllabilty matrix
obsv(A, C)	Observability matrix
gram(sys, type)	Gramian (controllability or observability)

5.8.1 control.matlab.ctrb

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

Controllabilty matrix

Parameters **B** (*A,*) – Dynamics and input matrix of the system

Returns **C** – Controllability matrix

Return type matrix

Examples

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

5.8.2 control.matlab.obsv

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

Observability matrix

Parameters **C** (*A,*) – Dynamics and output matrix of the system

Returns **O** – Observability matrix

Return type matrix

Examples

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

5.8.3 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** – Gramian of system

Return type array

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
```

5.9 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*.

5.9.1 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** – Cleaned model

Return type *StateSpace* or *TransferFunction*

5.9.2 control.matlab.hsvd

control.matlab.**hsvd**(*sys*)

Calculate the Hankel singular values.

Parameters **sys** (StateSpace) – A state space system

Returns **H** – A list of Hankel singular values

Return type array

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)
```

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

Return type *StateSpace*

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')
```

5.9.4 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** – A reduced order model

Return type *StateSpace*

Raises `ValueError` – Raised under the following conditions:

- 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')
```

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

Return type *StateSpace*

Examples

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

5.9.6 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** (*int*) – Number of Markov parameters to output

Returns **H** – First m Markov parameters

Return type ndarray

Notes

Currently only works for SISO

Examples

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

5.10 Time delays

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

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

Return type array

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

5.11 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
care(A, B, Q[, R, S, E, stabilizing])	(X,L,G) = care(A,B,Q,R=None) solves the continuous-time algebraic Riccati equation
dare(A, B, Q, R[, S, E, stabilizing])	(X,L,G) = dare(A,B,Q,R) solves the discrete-time algebraic Riccati equation

5.11.1 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.

5.11.2 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.

5.11.3 control.matlab.care

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

(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.

5.11.4 control.matlab.dare

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

(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.

5.12 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

5.12.1 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

- **C** (*P,*) – Linear input/output systems (process and control)
- **omega** (*array*) – Range of frequencies (list or bounds) in rad/sec

Returns

Return type None

5.12.2 control.matlab.unwrap

control.matlab.**unwrap**(*angle*, *period=6.283185307179586*)

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** – Output array, with jumps of period/2 eliminated

Return type array_like

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]
```

5.13 Functions imported from other modules

linspace(start, stop[, num, endpoint, ...])	Return evenly spaced numbers over a specified interval.
logspace(start, stop[, num, endpoint, base, ...])	Return numbers spaced evenly on a log scale.
ss2zpk(A, B, C, D[, input])	State-space representation to zero-pole-gain representation.
tf2zpk(b, a)	Return zero, pole, gain (z, p, k) representation from a numerator, denominator representation of a linear filter.
zpk2ss(z, p, k)	Zero-pole-gain representation to state-space representation
zpk2tf(z, p, k)	Return polynomial transfer function representation from zeros and poles

CHAPTER 6

Differentially flat systems

The `control.flatsys` package contains a set of classes and functions that can be used to compute trajectories for differentially flat systems.

A differentially flat system is defined by creating an object using the `FlatSystem` class, which has member functions for mapping the system state and input into and out of flat coordinates. The `point_to_point()` function can be used to create a trajectory between two endpoints, written in terms of a set of basis functions defined using the `BasisFamily` class. The resulting trajectory is return as a `SystemTrajectory` object and can be evaluated using the `eval()` member function.

6.1 Overview of differential flatness

A nonlinear differential equation of the form

$$\dot{x} = f(x, u), \qquad x \in R^n, u \in R^m$$

is *differentially flat* if there exists a function α such that

$$z = \alpha(x, u, \dot{u} \ldots, u^{(p)})$$

and we can write the solutions of the nonlinear system as functions of z and a finite number of derivatives

$$\begin{aligned} x &= \beta(z, \dot{z}, \ldots, z^{(q)}) \\ u &= \gamma(z, \dot{z}, \ldots, z^{(q)}). \end{aligned}$$

For a differentially flat system, all of the feasible trajectories for the system can be written as functions of a flat output $z(\cdot)$ and its derivatives. The number of flat outputs is always equal to the number of system inputs.

Differentially flat systems are useful in situations where explicit trajectory generation is required. Since the behavior of a flat system is determined by the flat outputs, we can plan trajectories in output space, and then map these to appropriate inputs. Suppose we wish to generate a feasible trajectory for the the nonlinear system

$$\dot{x} = f(x, u), \qquad x(0) = x_0, \, x(T) = x_f.$$

If the system is differentially flat then

$$x(0) = \beta\big(z(0), \dot z(0), \dots, z^{(q)}(0)\big) = x_0,$$
$$x(T) = \gamma\big(z(T), \dot z(T), \dots, z^{(q)}(T)\big) = x_f,$$

and we see that the initial and final condition in the full state space depends on just the output z and its derivatives at the initial and final times. Thus any trajectory for z that satisfies these boundary conditions will be a feasible trajectory for the system, using equation~eqref{eq:trajgen:flat2state} to determine the full state space and input trajectories.

In particular, given initial and final conditions on z and its derivatives that satisfy the initial and final conditions any curve $z(\cdot)$ satisfying those conditions will correspond to a feasible trajectory of the system. We can parameterize the flat output trajectory using a set of smooth basis functions $\psi_i(t)$:

$$z(t) = \sum_{i=1}^N \alpha_i \psi_i(t), \qquad \alpha_i \in R$$

We seek a set of coefficients α_i, $i = 1, \dots, N$ such that $z(t)$ satisfies the boundary conditions for $x(0)$ and $x(T)$. The derivatives of the flat output can be computed in terms of the derivatives of the basis functions:

$$\dot z(t) = \sum_{i=1}^N \alpha_i \dot \psi_i(t)$$
$$\vdots$$
$$\dot z^{(q)}(t) = \sum_{i=1}^N \alpha_i \psi^{(q)}_i(t).$$

We can thus write the conditions on the flat outputs and their derivatives as

$$\begin{bmatrix} \psi_1(0) & \psi_2(0) & \dots & \psi_N(0) \\ \dot \psi_1(0) & \dot \psi_2(0) & \dots & \dot \psi_N(0) \\ \vdots & \vdots & & \vdots \\ \psi^{(q)}_1(0) & \psi^{(q)}_2(0) & \dots & \psi^{(q)}_N(0) \\ \psi_1(T) & \psi_2(T) & \dots & \psi_N(T) \\ \dot \psi_1(T) & \dot \psi_2(T) & \dots & \dot \psi_N(T) \\ \vdots & \vdots & & \vdots \\ \psi^{(q)}_1(T) & \psi^{(q)}_2(T) & \dots & \psi^{(q)}_N(T) \end{bmatrix} \begin{bmatrix} \alpha_1 \\ \vdots \\ \alpha_N \end{bmatrix} = \begin{bmatrix} z(0) \\ \dot z(0) \\ \vdots \\ z^{(q)}(0) \\ z(T) \\ \dot z(T) \\ \vdots \\ z^{(q)}(T) \end{bmatrix}$$

This equation is a *linear* equation of the form

$$M \alpha = \begin{bmatrix} \bar z(0) \\ \bar z(T) \end{bmatrix}$$

where $\bar z$ is called the *flat flag* for the system. Assuming that M has a sufficient number of columns and that it is full column rank, we can solve for a (possibly non-unique) α that solves the trajectory generation problem.

6.2 Module usage

To create a trajectory for a differentially flat system, a `FlatSystem` object must be created. This is done by specifying the *forward* and *reverse* mappings between the system state/input and the differentially flat outputs and their derivatives ("flat flag").

The `forward()` method computes the flat flag given a state and input:

zflag = sys.forward(x, u)

The `reverse()` method computes the state and input given the flat flag:

x, u = sys.reverse(zflag)

The flag $\bar{z}$ is implemented as a list of flat outputs z_i and their derivatives up to order q_i:

zflag[i][j] = $z_i^{(j)}$

The number of flat outputs must match the number of system inputs.

For a linear system, a flat system representation can be generated using the `LinearFlatSystem` class:

flatsys = control.flatsys.LinearFlatSystem(linsys)

For more general systems, the *FlatSystem* object must be created manually

flatsys = control.flatsys.FlatSystem(nstate, ninputs, forward, reverse)

In addition to the flat system descriptionn, a set of basis functions $\phi_i(t)$ must be chosen. The *FlatBasis* class is used to represent the basis functions. A polynomial basis function of the form 1, t, t^2 $:, ...can be computed using the$ class, which is initialized by passing the desired order of the polynomial basis set:

polybasis = control.flatsys.PolyBasis(N)

Once the system and basis function have been defined, the `point_to_point()` function can be used to compute a trajectory between initial and final states and inputs:

traj = control.flatsys.point_to_point(x0, u0, xf, uf, Tf, basis=polybasis)

The returned object has class `SystemTrajectory` and can be used to compute the state and input trajectory between the initial and final condition:

xd, ud = traj.eval(T)

where *T* is a list of times on which the trajectory should be evaluated (e.g., *T = numpy.linspace(0, Tf, M)*.

6.3 Example

To illustrate how we can use a two degree-of-freedom design to improve the performance of the system, consider the problem of steering a car to change lanes on a road. We use the non-normalized form of the dynamics, which are derived *Feedback Systems* by Astrom and Murray, Example 3.11.

```
import control.flatsys as fs

# Function to take states, inputs and return the flat flag
def vehicle_flat_forward(x, u, params={}):
    # Get the parameter values
    b = params.get('wheelbase', 3.)

    # Create a list of arrays to store the flat output and its derivatives
    zflag = [np.zeros(3), np.zeros(3)]

    # Flat output is the x, y position of the rear wheels
    zflag[0][0] = x[0]
    zflag[1][0] = x[1]

    # First derivatives of the flat output
    zflag[0][1] = u[0] * np.cos(x[2])  # dx/dt
    zflag[1][1] = u[0] * np.sin(x[2])  # dy/dt
```

(continues on next page)

(continued from previous page)

```
    # First derivative of the angle
    thdot = (u[0]/b) * np.tan(u[1])

    # Second derivatives of the flat output (setting vdot = 0)
    zflag[0][2] = -u[0] * thdot * np.sin(x[2])
    zflag[1][2] =  u[0] * thdot * np.cos(x[2])

    return zflag

# Function to take the flat flag and return states, inputs
def vehicle_flat_reverse(zflag, params={}):
    # Get the parameter values
    b = params.get('wheelbase', 3.)

    # Create a vector to store the state and inputs
    x = np.zeros(3)
    u = np.zeros(2)

    # Given the flat variables, solve for the state
    x[0] = zflag[0][0]  # x position
    x[1] = zflag[1][0]  # y position
    x[2] = np.arctan2(zflag[1][1], zflag[0][1])  # tan(theta) = ydot/xdot

    # And next solve for the inputs
    u[0] = zflag[0][1] * np.cos(x[2]) + zflag[1][1] * np.sin(x[2])
    u[1] = np.arctan2(
        (zflag[1][2] * np.cos(x[2]) - zflag[0][2] * np.sin(x[2])), u[0]/b)

    return x, u

vehicle_flat = fs.FlatSystem(
    3, 2, forward=vehicle_flat_forward, reverse=vehicle_flat_reverse)
```

To find a trajectory from an initial state x_0 to a final state x_f in time T_f we solve a point-to-point trajectory generation problem. We also set the initial and final inputs, whi ch sets the vehicle velocity v and steering wheel angle δ at the endpoints.

```
# Define the endpoints of the trajectory
x0 = [0., -2., 0.]; u0 = [10., 0.]
xf = [100., 2., 0.]; uf = [10., 0.]
Tf = 10

# Define a set of basis functions to use for the trajectories
poly = fs.PolyFamily(6)

# Find a trajectory between the initial condition and the final condition
traj = fs.point_to_point(vehicle_flat, x0, u0, xf, uf, Tf, basis=poly)

# Create the trajectory
t = np.linspace(0, Tf, 100)
x, u = traj.eval(t)
```

6.4 Module classes and functions

6.4.1 Flat systems classes

BasisFamily(N)	Base class for implementing basis functions for flat systems.
FlatSystem(forward, reverse[, updfcn, . . .])	Base class for representing a differentially flat system.
LinearFlatSystem(linsys[, inputs, outputs, . . .])	
PolyFamily(N)	Polynomial basis functions.
SystemTrajectory(sys, basis[, coeffs, flaglen])	Class representing a system trajectory.

control.flatsys.BasisFamily

class control.flatsys.**BasisFamily**(*N*)

Base class for implementing basis functions for flat systems.

A BasisFamily object is used to construct trajectories for a flat system. The class must implement a single function that computes the jth derivative of the ith basis function at a time t:

$z_i^{(q)}(t)$ = basis.eval_deriv(self, i, j, t)

__init__(*N*)

Create a basis family of order N.

Methods

__init__(N)	Create a basis family of order N.

control.flatsys.FlatSystem

class control.flatsys.**FlatSystem**(*forward*, *reverse*, *updfcn=None*, *outfcn=None*, *inputs=None*, *outputs=None*, *states=None*, *params={}*, *dt=None*, *name=None*)

Base class for representing a differentially flat system.

The FlatSystem class is used as a base class to describe differentially flat systems for trajectory generation. The class must implement two functions:

zflag = flatsys.foward(x, u) This function computes the flag (derivatives) of the flat output. The inputs to this function are the state 'x' and inputs 'u' (both 1D arrays). The output should be a 2D array with the first dimension equal to the number of system inputs and the second dimension of the length required to represent the full system dynamics (typically the number of states)

x, u = flatsys.reverse(zflag) This function system state and inputs give the the flag (derivatives) of the flat output. The input to this function is an 2D array whose first dimension is equal to the number of system inputs and whose second dimension is of length required to represent the full system dynamics (typically the number of states). The output is the state *x* and inputs *u* (both 1D arrays).

A flat system is also an input/output system supporting simulation, composition, and linearization. If the update and output methods are given, they are used in place of the flat coordinates.

__init__(*forward*, *reverse*, *updfcn=None*, *outfcn=None*, *inputs=None*, *outputs=None*, *states=None*, *params={}*, *dt=None*, *name=None*)

Create a differentially flat input/output system.

The FlatIOSystem constructor is used to create an input/output system object that also represents a differentially flat system. The output of the system does not need to be the differentially flat output.

Parameters

- **forward** (*callable*) – A function to compute the flat flag given the states and input.
- **reverse** (*callable*) – A function to compute the states and input given the flat flag.
- **updfcn** (*callable, optional*) – Function returning the state update function

  updfcn(t, x, u[, param]) -> array

  where *x* is a 1-D array with shape (nstates,), *u* is a 1-D array with shape (ninputs,), *t* is a float representing the currrent time, and *param* is an optional dict containing the values of parameters used by the function. If not specified, the state space update will be computed using the flat system coordinates.
- **outfcn** (*callable*) – Function returning the output at the given state

  outfcn(t, x, u[, param]) -> array

  where the arguments are the same as for *upfcn*. If not specified, the output will be the flat outputs.
- **inputs** (*int, list of str, or None*) – Description of the system inputs. This can be given as an integer count or as a list of strings that name the individual signals. If an integer count is specified, the names of the signal will be of the form *s[i]* (where *s* is one of *u*, *y*, or *x*). If this parameter is not given or given as *None*, the relevant quantity will be determined when possible based on other information provided to functions using the system.
- **outputs** (*int, list of str, or None*) – Description of the system outputs. Same format as *inputs*.
- **states** (*int, list of str, or None*) – Description of the system states. Same format as *inputs*.
- **dt** (*None, True or float, optional*) – System timebase. None (default) indicates continuous time, True indicates discrete time with undefined sampling time, positive number is discrete time with specified sampling time.
- **params** (*dict, optional*) – Parameter values for the systems. Passed to the evaluation functions for the system as default values, overriding internal defaults.
- **name** (*string, optional*) – System name (used for specifying signals)

Returns Input/output system object

Return type *InputOutputSystem*

Methods

`__init__`(forward, reverse[, updfcn, outfcn, …])	Create a differentially flat input/output system.
`copy`()	Make a copy of an input/output system.
`feedback`([other, sign, params])	Feedback interconnection between two input/output systems
`find_input`(name)	Find the index for an input given its name (*None* if not found)

Continued on next page

Table 3 – continued from previous page

find_output(name)	Find the index for an output given its name (*None* if not found)
find_state(name)	Find the index for a state given its name (*None* if not found)
forward(x, u[, params])	Compute the flat flag given the states and input.
linearize(x0, u0[, t, params, eps])	Linearize an input/output system at a given state and input.
reverse(zflag[, params])	Compute the states and input given the flat flag.
set_inputs(inputs[, prefix])	Set the number/names of the system inputs.
set_outputs(outputs[, prefix])	Set the number/names of the system outputs.
set_states(states[, prefix])	Set the number/names of the system states.

copy()

Make a copy of an input/output system.

feedback(*other=1*, *sign=-1*, *params={}*)

Feedback interconnection between two input/output systems

Parameters

- **sys1** (InputOutputSystem) – The primary process.
- **sys2** (InputOutputSystem) – The feedback process (often a feedback controller).
- **sign** (*scalar, optional*) – 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

Return type *InputOutputSystem*

Raises ValueError – if the inputs, outputs, or timebases of the systems are incompatible.

find_input(*name*)

Find the index for an input given its name (*None* if not found)

find_output(*name*)

Find the index for an output given its name (*None* if not found)

find_state(*name*)

Find the index for a state given its name (*None* if not found)

forward(*x*, *u*, *params={}*)

Compute the flat flag given the states and input.

Given the states and inputs for a system, compute the flat outputs and their derivatives (the flat "flag") for the system.

Parameters

- **x** (*list or array*) – The state of the system.
- **u** (*list or array*) – The input to the system.
- **params** (*dict, optional*) – Parameter values for the system. Passed to the evaluation functions for the system as default values, overriding internal defaults.

Returns **zflag** – For each flat output z_i, zflag[i] should be an ndarray of length q_i that contains the flat output and its first q_i derivatives.

Return type list of 1D arrays

linearize(*x0*, *u0*, *t=0*, *params={}*, *eps=1e-06*)
Linearize an input/output system at a given state and input.

Return the linearization of an input/output system at a given state and input value as a StateSpace system. See `linearize()` for complete documentation.

reverse(*zflag*, *params={}*)
Compute the states and input given the flat flag.

Parameters

- **zflag** (*list of arrays*) – For each flat output z_i, zflag[i] should be an ndarray of length q_i that contains the flat output and its first q_i derivatives.
- **params** (*dict, optional*) – Parameter values for the system. Passed to the evaluation functions for the system as default values, overriding internal defaults.

Returns

- **x** (*1D array*) – The state of the system corresponding to the flat flag.
- **u** (*1D array*) – The input to the system corresponding to the flat flag.

set_inputs(*inputs*, *prefix='u'*)
Set the number/names of the system inputs.

Parameters

- **inputs** (*int, list of str, or None*) – Description of the system inputs. This can be given as an integer count or as a list of strings that name the individual signals. If an integer count is specified, the names of the signal will be of the form *u[i]* (where the prefix *u* can be changed using the optional prefix parameter).
- **prefix** (*string, optional*) – If *inputs* is an integer, create the names of the states using the given prefix (default = 'u'). The names of the input will be of the form *prefix[i]*.

set_outputs(*outputs*, *prefix='y'*)
Set the number/names of the system outputs.

Parameters

- **outputs** (*int, list of str, or None*) – Description of the system outputs. This can be given as an integer count or as a list of strings that name the individual signals. If an integer count is specified, the names of the signal will be of the form *u[i]* (where the prefix *u* can be changed using the optional prefix parameter).
- **prefix** (*string, optional*) – If *outputs* is an integer, create the names of the states using the given prefix (default = 'y'). The names of the input will be of the form *prefix[i]*.

set_states(*states*, *prefix='x'*)
Set the number/names of the system states.

Parameters

- **states** (*int, list of str, or None*) – Description of the system states. This can be given as an integer count or as a list of strings that name the individual signals. If an integer count is specified, the names of the signal will be of the form *u[i]* (where the prefix *u* can be changed using the optional prefix parameter).
- **prefix** (*string, optional*) – If *states* is an integer, create the names of the states using the given prefix (default = 'x'). The names of the input will be of the form *prefix[i]*.

control.flatsys.LinearFlatSystem

class control.flatsys.**LinearFlatSystem**(*linsys*, *inputs=None*, *outputs=None*, *states=None*, *name=None*)

__init__(*linsys*, *inputs=None*, *outputs=None*, *states=None*, *name=None*)
Define a flat system from a SISO LTI system.

Given a reachable, single-input/single-output, linear time-invariant system, create a differentially flat system representation.

Parameters

- **linsys** (`StateSpace`) – LTI StateSpace system to be converted
- **inputs** (*int, list of str or None, optional*) – Description of the system inputs. This can be given as an integer count or as a list of strings that name the individual signals. If an integer count is specified, the names of the signal will be of the form *s[i]* (where *s* is one of *u*, *y*, or *x*). If this parameter is not given or given as *None*, the relevant quantity will be determined when possible based on other information provided to functions using the system.
- **outputs** (*int, list of str or None, optional*) – Description of the system outputs. Same format as *inputs*.
- **states** (*int, list of str, or None, optional*) – Description of the system states. Same format as *inputs*.
- **dt** (*None, True or float, optional*) – System timebase. None (default) indicates continuous time, True indicates discrete time with undefined sampling time, positive number is discrete time with specified sampling time.
- **params** (*dict, optional*) – Parameter values for the systems. Passed to the evaluation functions for the system as default values, overriding internal defaults.
- **name** (*string, optional*) – System name (used for specifying signals)

Returns **iosys** – Linear system represented as an flat input/output system

Return type *LinearFlatSystem*

Methods

`__init__`(linsys[, inputs, outputs, states, name])	Define a flat system from a SISO LTI system.
`append`(other)	Append a second model to the present model.
`copy`()	Make a copy of an input/output system.
`damp`()	Natural frequency, damping ratio of system poles
`dcgain`()	Return the zero-frequency gain
`evalfr`(omega)	Evaluate a SS system's transfer function at a single frequency.
`feedback`([other, sign, params])	Feedback interconnection between two input/output systems
`find_input`(name)	Find the index for an input given its name (*None* if not found)
`find_output`(name)	Find the index for an output given its name (*None* if not found)

Continued on next page

Table 4 – continued from previous page

`find_state`(name)	Find the index for a state given its name (*None* if not found)
`forward`(x, u)	Compute the flat flag given the states and input.
`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`()	Check to see if a system is single input, single output
`lft`(other[, nu, ny])	Return the Linear Fractional Transformation.
`linearize`(x0, u0[, t, params, eps])	Linearize an input/output system at a given state and input.
`minreal`([tol])	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.
`reverse`(zflag)	Compute the states and input given the flat flag.
`sample`(Ts[, method, alpha])	Convert a continuous time system to discrete time
`set_inputs`(inputs[, prefix])	Set the number/names of the system inputs.
`set_outputs`(outputs[, prefix])	Set the number/names of the system outputs.
`set_states`(states[, prefix])	Set the number/names of the system states.
`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

copy()
Make a copy of an input/output system.

damp()
Natural frequency, damping ratio of system poles

Returns

- **wn** (*array*) – Natural frequencies for each system pole
- **zeta** (*array*) – Damping ratio for each system pole
- **poles** (*array*) – Array of system poles

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 – 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.

Return type ndarray

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*, *params={}*)
Feedback interconnection between two input/output systems

Parameters

- **sys1** (InputOutputSystem) – The primary process.
- **sys2** (InputOutputSystem) – The feedback process (often a feedback controller).
- **sign** (*scalar, optional*) – 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

Return type *InputOutputSystem*

Raises ValueError – if the inputs, outputs, or timebases of the systems are incompatible.

find_input (*name*)
Find the index for an input given its name (*None* if not found)

find_output (*name*)
Find the index for an output given its name (*None* if not found)

find_state (*name*)
Find the index for a state given its name (*None* if not found)

forward (*x*, *u*)
Compute the flat flag given the states and input.

See *control.flatsys.FlatSystem.forward()* for more info.

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

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

Reports the frequency response of the system,

G(j*omega) = mag*exp(j*phase)

for continuous time. For discrete time systems, the response is evaluated around the unit circle such that

G(exp(j*omega*dt)) = mag*exp(j*phase).

Parameters **omega** (*array*) – A list of frequencies in radians/sec at which the system should be evaluated. The list can be either a python list or a numpy array and will be sorted before evaluation.

Returns

- **mag** (*float*) – The magnitude (absolute value, not dB or log10) of the system frequency response.
- **phase** (*float*) – The wrapped phase in radians of the system frequency response.
- **omega** (*array*) – The list of sorted frequencies at which the response was evaluated.

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, optional*) – If strict is True, make sure that timebase is not None. Default is False.

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

Parameters strict (*bool, optional*) – If strict is True, make sure that timebase is not None. Default is False.

issiso ()
Check to see if a system is single input, single output

lft (*other, nu=-1, ny=-1*)
Return the Linear Fractional Transformation.

A definition of the LFT operator can be found in Appendix A.7, page 512 in the 2nd Edition, Multivariable Feedback Control by Sigurd Skogestad.

An alternative definition can be found here: https://www.mathworks.com/help/control/ref/lft.html

Parameters

- **other** (*LTI*) – The lower LTI system
- **ny** (*int, optional*) – Dimension of (plant) measurement output.
- **nu** (*int, optional*) – Dimension of (plant) control input.

linearize (*x0, u0, t=0, params={}, eps=1e-06*)
Linearize an input/output system at a given state and input.

Return the linearization of an input/output system at a given state and input value as a StateSpace system. See `linearize()` for complete documentation.

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.

reverse (*zflag*)
Compute the states and input given the flat flag.

See `control.flatsys.FlatSystem.reverse()` for more info.

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** – Discrete time system, with sampling rate Ts

Return type *StateSpace*

Notes

Uses the command 'cont2discrete' from scipy.signal

Examples

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

set_inputs (*inputs*, *prefix='u'*)
Set the number/names of the system inputs.

Parameters

- **inputs** (*int, list of str, or None*) – Description of the system inputs. This can be given as an integer count or as a list of strings that name the individual signals. If an integer count is specified, the names of the signal will be of the form *u[i]* (where the prefix *u* can be changed using the optional prefix parameter).
- **prefix** (*string, optional*) – If *inputs* is an integer, create the names of the states using the given prefix (default = 'u'). The names of the input will be of the form *prefix[i]*.

set_outputs (*outputs*, *prefix='y'*)
Set the number/names of the system outputs.

Parameters

- **outputs** (*int, list of str, or None*) – Description of the system outputs. This can be given as an integer count or as a list of strings that name the individual signals. If an integer count is specified, the names of the signal will be of the form *u[i]* (where the prefix *u* can be changed using the optional prefix parameter).
- **prefix** (*string, optional*) – If *outputs* is an integer, create the names of the states using the given prefix (default = 'y'). The names of the input will be of the form *prefix[i]*.

set_states (*states*, *prefix='x'*)
Set the number/names of the system states.

Parameters

- **states** (*int, list of str, or None*) – Description of the system states. This can be given as an integer count or as a list of strings that name the individual signals. If an integer count is specified, the names of the signal will be of the form *u[i]* (where the prefix *u* can be changed using the optional prefix parameter).
- **prefix** (*string, optional*) – If *states* is an integer, create the names of the states using the given prefix (default = 'x'). The names of the input will be of the form *prefix[i]*.

zero()

Compute the zeros of a state space system.

control.flatsys.PolyFamily

class control.flatsys.**PolyFamily**(*N*)

Polynomial basis functions.

This class represents the family of polynomials of the form

$$\phi_i(t) = t^i$$

__init__(*N*)

Create a polynomial basis of order N.

Methods

__init__(N)	Create a polynomial basis of order N.
eval_deriv(i, k, t)	Evaluate the kth derivative of the ith basis function at time t.

eval_deriv(*i, k, t*)

Evaluate the kth derivative of the ith basis function at time t.

control.flatsys.SystemTrajectory

class control.flatsys.**SystemTrajectory**(*sys, basis, coeffs=[], flaglen=[]*)

Class representing a system trajectory.

The *SystemTrajectory* class is used to represent the trajectory of a (differentially flat) system. Used by the `point_to_point()` function to return a trajectory.

__init__(*sys, basis, coeffs=[], flaglen=[]*)

Initilize a system trajectory object.

Parameters

- **sys** (`FlatSystem`) – Flat system object associated with this trajectory.
- **basis** (`BasisFamily`) – Family of basis vectors to use to represent the trajectory.
- **coeffs** (*list of 1D arrays, optional*) – For each flat output, define the coefficients of the basis functions used to represent the trajectory. Defaults to an empty list.

- **flaglen** (*list of ints, optional*) – For each flat output, the number of derivatives of the flat output used to define the trajectory. Defaults to an empty list.

Methods

__init__(sys, basis[, coeffs, flaglen])	Initilize a system trajectory object.
eval(tlist)	Return the state and input for a trajectory at a list of times.

eval (*tlist*)

Return the state and input for a trajectory at a list of times.

Evaluate the trajectory at a list of time points, returning the state and input vectors for the trajectory:

x, u = traj.eval(tlist)

Parameters **tlist** (*1D array*) – List of times to evaluate the trajectory.

Returns

- **x** (*2D array*) – For each state, the values of the state at the given times.
- **u** (*2D array*) – For each input, the values of the input at the given times.

6.4.2 Flat systems functions

point_to_point(sys, x0, u0, xf, uf, Tf[, . . .])	Compute trajectory between an initial and final conditions.

control.flatsys.point_to_point

control.flatsys.**point_to_point** (*sys*, *x0*, *u0*, *xf*, *uf*, *Tf*, *T0=0*, *basis=None*, *cost=None*)

Compute trajectory between an initial and final conditions.

Compute a feasible trajectory for a differentially flat system between an initial condition and a final condition.

Parameters

- **flatsys** (*FlatSystem object*) – Description of the differentially flat system. This object must define a function flatsys.forward() that takes the system state and produceds the flag of flat outputs and a system flatsys.reverse() that takes the flag of the flat output and prodes the state and input.
- **u0, xf, uf** (*x0,*) – Define the desired initial and final conditions for the system. If any of the values are given as None, they are replaced by a vector of zeros of the appropriate dimension.
- **Tf** (*float*) – The final time for the trajectory (corresponding to xf)
- **T0** (*float (optional)*) – The initial time for the trajectory (corresponding to x0). If not specified, its value is taken to be zero.
- **basis** (*BasisFamily object (optional)*) – The basis functions to use for generating the trajectory. If not specified, the PolyFamily basis family will be used, with the minimal number of elements required to find a feasible trajectory (twice the number of system states)

Returns traj – The system trajectory is returned as an object that implements the eval() function, we can be used to compute the value of the state and input and a given time t.

Return type SystemTrajectory object

- 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

C

Index

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

Z