

---

# Python Control Library Documentation

*Release dev*

**RMM**

**Dec 22, 2018**



---

## Contents

---

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	Overview of the toolbox . . . . .	3
1.2	Some differences from MATLAB . . . . .	3
1.3	Installation . . . . .	3
1.4	Getting started . . . . .	4
<b>2</b>	<b>Library conventions</b>	<b>5</b>
2.1	LTI system representation . . . . .	5
2.2	Time series data . . . . .	6
2.3	Package configuration . . . . .	8
<b>3</b>	<b>Function reference</b>	<b>9</b>
3.1	System creation . . . . .	9
3.2	System interconnections . . . . .	14
3.3	Frequency domain plotting . . . . .	18
3.4	Time domain simulation . . . . .	20
3.5	Block diagram algebra . . . . .	24
3.6	Control system analysis . . . . .	24
3.7	Matrix computations . . . . .	30
3.8	Control system synthesis . . . . .	32
3.9	Model simplification tools . . . . .	36
3.10	Utility functions and conversions . . . . .	39
<b>4</b>	<b>LTI system classes</b>	<b>49</b>
4.1	control.TransferFunction . . . . .	49
4.2	control.StateSpace . . . . .	52
<b>5</b>	<b>MATLAB compatibility module</b>	<b>57</b>
5.1	Creating linear models . . . . .	57
5.2	Utility functions and conversions . . . . .	61
5.3	System interconnections . . . . .	64
5.4	System gain and dynamics . . . . .	67
5.5	Time-domain analysis . . . . .	69
5.6	Frequency-domain analysis . . . . .	72
5.7	Compensator design . . . . .	76
5.8	State-space (SS) models . . . . .	78
5.9	Model simplification . . . . .	80

5.10	Time delays . . . . .	83
5.11	Matrix equation solvers and linear algebra . . . . .	84
5.12	Additional functions . . . . .	85
5.13	Functions imported from other modules . . . . .	86
<b>Python Module Index</b>		<b>89</b>

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



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 things 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 *openblas* 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 *
```



---

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:

$$\begin{aligned}\frac{dx}{dt} &= Ax + Bu \\ y &= Cx + Du\end{aligned}$$

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

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

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 `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` and `TransferFunction` 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 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(sys)
plot(t, y)
```

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

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

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

```
ft = D * U
```

## 2.3 Package configuration

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

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

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

Functions that can be used to set standard configurations:

---

<code>use_fbs_defaults()</code>	Use <a href="#">Feedback Systems</a> (FBS) compatible settings
<code>use_matlab_defaults()</code>	Use MATLAB compatible configuration settings

---

### 2.3.1 `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

### 2.3.2 `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

## Function reference

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

### 3.1 System creation

<i>ss</i> (A, B, C, D[, dt])	Create a state space system.
<i>tf</i> (num, den[, dt])	Create a transfer function system.
<i>FRD</i> (d, w)	A class for models defined by frequency response data (FRD)
<i>rss</i> ([states, outputs, inputs])	Create a stable <i>continuous</i> random state space object.
<i>drss</i> ([states, outputs, inputs])	Create a stable <i>discrete</i> 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:

$$\begin{aligned}x[k+1] &= A \cdot x[k] + B \cdot u[k] \\ y[k] &= C \cdot x[k] + D \cdot u[k]\end{aligned}$$

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.

#### 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  $2s^2 + 3s + 4$ .

#### Examples

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

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

### 3.1.3 control.FRD

**class** `control.FRD(d, w)`

A class for models defined by frequency response data (FRD)

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

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

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

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

`__init__` (\*args, \*\*kwargs)  
FRD(d, w)

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

<code>__init__</code> (*args, **kwargs)	FRD(d, w)
<code>damp</code> ()	Natural frequency, damping ratio of system poles
<code>dcgain</code> ()	Return the zero-frequency gain
<code>eval</code> (omega)	Evaluate a transfer function at a single angular frequency.
<code>evalfr</code> (omega)	Evaluate a transfer function at a single angular frequency.
<code>feedback</code> ([other, sign])	Feedback interconnection between two FRD objects.
<code>freqresp</code> (omega)	Evaluate a transfer function at a list of angular frequencies.
<code>isctime</code> ([strict])	Check to see if a system is a continuous-time system
<code>isdttime</code> ([strict])	Check to see if a system is a discrete-time system
<code>issiso</code> ()	Check to see if a system is single input, single output

## Attributes

<code>epsw</code>
-------------------

`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  $s = i * \text{omega}$ , where omega is a list of angular frequencies, and is a sorted version of the input omega.

**isctime** (*strict=False*)

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

#### Parameters

- **sys** (*LTI system*) – System to be checked
- **strict** (*bool (default = False)*) – If strict is True, make sure that timebase is not None

**isdttime** (*strict=False*)

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

**Parameters** **strict** (*bool (default = False)*) – If strict is True, make sure that timebase is not None

**issiso** ()

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

### 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

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

**Todo:** also implement for transfer function, zpk, etc.

## 3.2.2 control.connect

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

Index-base interconnection of system

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

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

### Parameters

- **sys** (*StateSpace Transferfunction*) – System to be connected
- **Q** (*2d array*) – Interconnection matrix. First column gives the input to be connected second column gives the output to be fed into this input. Negative values for the second column mean the feedback is negative, 0 means no connection is made
- **inputv** (*1d array*) – list of final external inputs
- **outputv** (*1d array*) – list of final external outputs

**Returns** `sys` – 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 = sp.mat([ [ 1, 2], [2, -1] ]) # basically feedback, output 2 in 1
>>> 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 plant.

- **sys2**(*scalar*, *StateSpace*, *TransferFunction*, *FRD*) – The feedback plant (often a feedback controller).
- **sign**(*scalar*) – The sign of feedback. *sign* = -1 indicates negative feedback, and *sign* = 1 indicates positive feedback. *sign* is an optional argument; it assumes a value of -1 if not specified.

**Returns out**

**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.

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

### Examples

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

### 3.2.5 control.parallel

`control.parallel(sys1, *sysn)`

Return the parallel connection  $\text{sys1} + \text{sys2} (+ \text{sys3} + \dots)$

#### 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  $(\dots * \text{sys3} *) \text{sys2} * \text{sys1}$

#### 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

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

### 3.3.1 control.bode\_plot

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

Bode plot for a system

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

#### Parameters

- **syslist** (*linsys*) – List of linear input/output systems (single system is OK)
- **omega** (*list*) – List of frequencies in rad/sec to be used for frequency response
- **dB** (*boolean*) – If True, plot result in dB
- **Hz** (*boolean*) – If True, plot frequency in Hz (omega must be provided in rad/sec)
- **deg** (*boolean*) – If True, plot phase in degrees (else radians)
- **Plot** (*boolean*) – If True, plot magnitude and phase
- **omega\_limits** (*tuple, list, .. of two values*) – Limits of the to generate frequency vector. If `Hz=True` the limits are in Hz otherwise in rad/s.
- **omega\_num** (*int*) – number of samples
- **\*\*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

## 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 ( $dt = \text{True}$ ),  $dt$  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='b', labelFreq=0, *args, **kwargs)`

Nyquist plot for a system

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

#### Parameters

- **syslist** (*list of LTI*) – List of linear input/output systems (single system is OK)
- **omega** (*freq\_range*) – Range of frequencies (list or bounds) in rad/sec
- **Plot** (*boolean*) – If True, plot magnitude
- **labelFreq** (*int*) – Label every nth frequency on the plot
- **\*\*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** ( $P_r$ ) – 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` (*syslist*, *omega=None*, *grid=True*)

Nichols plot for a system

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

**Parameters**

- **syslist** (*list of LTI, or LTI*) – List of linear input/output systems (single system is OK)
- **omega** (*array\_like*) – Range of frequencies (list or bounds) in rad/sec
- **grid** (*boolean, optional*) – True if the plot should include a Nichols-chart grid. Default is True.

**Returns**

**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 matplotlib 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

<code>forced_response</code> ( <i>sys</i> [, <i>T</i> , <i>U</i> , <i>X0</i> , <i>transpose</i> ])	Simulate the output of a linear system.
<code>impulse_response</code> ( <i>sys</i> [, <i>T</i> , <i>X0</i> , <i>input</i> , ...])	Impulse response of a linear system
<code>initial_response</code> ( <i>sys</i> [, <i>T</i> , <i>X0</i> , <i>input</i> , ...])	Initial condition response of a linear system
<code>step_response</code> ( <i>sys</i> [, <i>T</i> , <i>X0</i> , <i>input</i> , <i>output</i> , ...])	Step response of a linear system
<code>phase_plot</code> ( <i>odefun</i> [, <i>X</i> , <i>Y</i> , <i>scale</i> , <i>X0</i> , <i>T</i> , ...])	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*)

Simulate the output of a linear system.

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

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

**Parameters**

- **sys** (*LTI (StateSpace, or TransferFunction)*) – LTI system to simulate



- **T** (*array-like*) – Time steps at which the input is defined; values must be evenly spaced.
- **U** (*array-like or number, optional*) – Input array giving input at each time *T* (default = 0).

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

- **X0** (*array-like or number, optional*) – Initial condition (default = 0).
- **transpose** (*bool*) – If `True`, transpose all input and output arrays (for backward compatibility with MATLAB and `scipy.signal.lsim`)

#### Returns

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

See also:

`step_response()`, `initial_response()`, `impulse_response()`

#### Examples

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

See [Time series data](#).

### 3.4.2 control.impulse\_response

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

Impulse response of a linear system

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

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

#### Parameters

- **sys** (*StateSpace, TransferFunction*) – LTI system to simulate
- **T** (*array-like object, optional*) – Time vector (argument is autocomputed if not given)
- **X0** (*array-like object or number, optional*) – Initial condition (default = 0)

Numbers are converted to constant arrays with the correct shape.

- **input** (*int*) – Index of the input that will be used in this simulation.
- **output** (*int*) – Index of the output that will be used in this simulation. Set to `None` to not trim outputs
- **transpose** (*bool*) – If `True`, transpose all input and output arrays (for backward compatibility with MATLAB and `scipy.signal.lsim`)

- **return\_x** (*bool*) – If True, return the state vector (default = False).

#### Returns

- **T** (*array*) – Time values of the output
- **yout** (*array*) – Response of the system
- **xout** (*array*) – Individual response of each x variable

See also:

[`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*)

Initial condition response of a linear system

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

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

#### Parameters

- **sys** (*StateSpace, or TransferFunction*) – LTI system to simulate
- **T** (*array-like object, optional*) – Time vector (argument is autocomputed if not given)
- **X0** (*array-like object or number, optional*) – Initial condition (default = 0)

Numbers are converted to constant arrays with the correct shape.

- **input** (*int*) – Ignored, has no meaning in initial condition calculation. Parameter ensures compatibility with `step_response` and `impulse_response`
- **output** (*int*) – Index of the output that will be used in this simulation. Set to None to not trim outputs
- **transpose** (*bool*) – If True, transpose all input and output arrays (for backward compatibility with MATLAB and `scipy.signal.lsim`)
- **return\_x** (*bool*) – If True, return the state vector (default = False).

#### Returns

- **T** (*array*) – Time values of the output
- **yout** (*array*) – Response of the system
- **xout** (*array*) – Individual response of each x variable

See also:

[`forced\_response\(\)`](#), [`impulse\_response\(\)`](#), [`step\_response\(\)`](#)

## Examples

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

### 3.4.4 control.step\_response

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

Step response of a linear system

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

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

#### Parameters

- **sys** (*StateSpace*, or *TransferFunction*) – LTI system to simulate
- **T** (*array-like object*, *optional*) – Time vector (argument is autocomputed if not given)
- **X0** (*array-like or number*, *optional*) – Initial condition (default = 0)  
Numbers are converted to constant arrays with the correct shape.
- **input** (*int*) – Index of the input that will be used in this simulation.
- **output** (*int*) – Index of the output that will be used in this simulation. Set to None to not trim outputs
- **transpose** (*bool*) – If True, transpose all input and output arrays (for backward compatibility with MATLAB and `scipy.signal.lsim`)
- **return\_x** (*bool*) – If True, return the state vector (default = False).

#### Returns

- **T** (*array*) – Time values of the output
- **yout** (*array*) – Response of the system
- **xout** (*array*) – Individual response of each x variable

See also:

[`forced\_response\(\)`](#), [`initial\_response\(\)`](#), [`impulse\_response\(\)`](#)

## Examples

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

### 3.4.5 control.phase\_plot

`control.phase_plot` (*odefun*, *X=None*, *Y=None*, *scale=1*, *X0=None*, *T=None*, *lingrid=None*, *lin-time=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 as used for `scipy.integrate`. Namely, it should be a function of the form  $dx/dt = F(x, t)$  that accepts a state  $x$  of dimension 2 and returns a derivative  $dx/dt$  of dimension 2.
- **Y** (*X, ...*) – Two 1-D arrays representing  $x$  and  $y$  coordinates of a grid. These arguments are passed to `meshgrid` and generate the lists of points at which the vector field is plotted. If absent (or `None`), the vector field is not plotted.
- **scale** (*float, optional*) – Scale size of arrows; default = 1
- **X0** (*ndarray of initial conditions, optional*) – List of initial conditions from which streamlines are plotted. Each initial condition should be a pair of numbers.
- **T** (*array-like or number, optional*) – Length of time to run simulations that generate streamlines. If a single number, the same simulation time is used for all initial conditions. Otherwise, should be a list of length `len(X0)` that gives the simulation time for each initial condition. Default value = 50.
- **= 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 spaced 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(), Y()`

#### Examples

## 3.5 Block diagram algebra

<code>series(sys1, *sysn)</code>	Return the series connection (...)
<code>parallel(sys1, *sysn)</code>	Return the parallel connection $sys1 + sys2 (+ sys3 + \dots)$
<code>feedback(sys1[, sys2, sign])</code>	Feedback interconnection between two I/O systems.
<code>negate(sys)</code>	Return the negative of a system.

## 3.6 Control system analysis

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

### 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\_like*) – 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])
>>> PhaseCrossoverFrequencies(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, title='Pole Zero Map')`

Plot a pole/zero map for a linear system.

**Parameters**

- **sys** (*LTI (StateSpace or TransferFunction)*) – Linear system for which poles and zeros are computed.
- **Plot** (*bool*) – If `True` a graph is generated with Matplotlib, otherwise the poles and zeros are only computed and returned.

**Returns**

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

### 3.6.10 control.root\_locus

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

Root locus plot

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

**Parameters**

- **sys** (*LTI object*) – Linear input/output systems (SISO only, for now)
- **kvect** (*list or ndarray, optional*) – List of gains to use in computing diagram
- **xlim** (*tuple or list, optional*) – control of x-axis range, normally with tuple (see `matplotlib.axes`)
- **ylim** (*tuple or list, optional*) – control of y-axis range
- **Plot** (*boolean, optional (default = True)*) – If `True`, plot root locus diagram.
- **PrintGain** (*boolean (default = True)*) – If `True`, report mouse clicks when close to the root-locus branches, calculate gain, damping and print
- **grid** (*boolean (default = False)*) – If `True` plot s-plane grid.

**Returns**

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

## 3.7 Matrix computations

<code>care(A, B, Q[, R, S, E])</code>	<code>(X,L,G) = care(A,B,Q,R=None)</code> solves the continuous-time algebraic Riccati equation
<code>dare(A, B, Q, R[, S, E])</code>	<code>(X,L,G) = dare(A,B,Q,R)</code> solves the discrete-time algebraic Riccati equation
<code>lyap(A, Q[, C, E])</code>	<code>X = lyap(A,Q)</code> solves the continuous-time Lyapunov equation
<code>dlyap(A, Q[, C, E])</code>	<code>dlyap(A,Q)</code> solves the discrete-time Lyapunov equation
<code>ctrb(A, B)</code>	Controllability matrix
<code>obsv(A, C)</code>	Observability matrix
<code>gram(sys, type)</code>	Gramian (controllability or observability)

### 3.7.1 control.care

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

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

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

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

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

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

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

### 3.7.2 control.dare

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

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

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

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

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

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

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

### 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)`

Controllability 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

<code>acker(A, B, poles)</code>	Pole placement using Ackermann method
<code>h2syn(P, nmeas, ncon)</code>	H <sub>2</sub> control synthesis for plant P.
<code>hinfsyn(P, nmeas, ncon)</code>	H <sub>∞</sub> control synthesis for plant P.
<code>lqr(A, B, Q, R[, N])</code>	Linear quadratic regulator design
<code>mixsyn(g[, w1, w2, w3])</code>	Mixed-sensitivity H-infinity synthesis.
<code>place(A, B, p)</code>	Place closed loop eigenvalues $K = \text{place}(A, B, p)$

### 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<sub>2</sub> 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<sub>{inf}</sub> 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** (`2-d array`) – State feedback gains
- **S** (`2-d array`) – Solution to Riccati equation
- **E** (`1-d array`) – Eigenvalues of the closed loop system

### Examples

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

### 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  $k_1$ -by- $n_y$  LTI*) –
- **w2** (*weighting on  $k*s$ ; None, or scalar or  $k_2$ -by- $n_u$  LTI*) –
- **w3** (*weighting on  $t = g*k*(1+g*k)^{-1}$ ; None, or scalar or  $k_3$ -by- $n_y$  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  $-[z] = [p_{11} \ p_{12}] [w], [y] [p_{21} \ g] [u]$*
- *then cl is the system from  $w \rightarrow z$  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 `hinfsv` for details
- *If a weighting w is scalar, it will be replaced by  $I*w$ , where I is*
- *$n_y$ -by- $n_y$  for w1 and w3, and  $n_u$ -by- $n_u$  for w2.*

See also:

`hinfsv()`, `augw()`

### 3.8.6 control.place

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

Place closed loop eigenvalues  $K = \text{place}(A, B, p)$

#### 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  $\text{rank}(B)$  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

<code>minreal(sys[, tol, verbose])</code>	Eliminates uncontrollable or unobservable states in state-space models or cancelling pole-zero pairs in transfer functions.
<code>balred(sys, orders[, method, alpha])</code>	Balanced reduced order model of <code>sys</code> of a given order.
<code>hsvd(sys)</code>	Calculate the Hankel singular values.
<code>modred(sys, ELIM[, method])</code>	Model reduction of <code>sys</code> by eliminating the states in <i>ELIM</i> using a given method.
<code>era(YY, m, n, nin, nout, r)</code>	Calculate an ERA model of order $r$ based on the impulse-response data <i>YY</i> .
<code>markov(Y, U, M)</code>	Calculate the first $M$ Markov parameters $[D \text{ } CB \text{ } CAB \dots]$ 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,  $\alpha \leq 0$  defines the stability boundary for the real part of *A*'s eigenvalues and for discrete-time systems,  $0 \leq \alpha \leq 1$  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** Matrix

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` – 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  $\text{sys}=\text{ss}(\text{Ar},\text{Br},\text{Cr},\text{Dr})$

**Return type** *StateSpace*

### Examples

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

## 3.9.6 control.markov

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

Calculate the first  $M$  Markov parameters  $[D \text{ CB CAB} \dots]$  from input  $U$ , output  $Y$ .

#### Parameters

- **Y**(*array\_like*) – Output data
- **U**(*array\_like*) – Input data
- **M**(*integer*) – Number of Markov parameters to output

**Returns** **H** – First  $M$  Markov parameters

**Return type** matrix

### Notes

Currently only works for SISO

### Examples

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

## 3.10 Utility functions and conversions

<i>augw</i> (g[, w1, w2, w3])	Augment plant for mixed sensitivity problem.
<i>canonical_form</i> (xsys[, form])	Convert a system into canonical form
<i>damp</i> (sys[, dprint])	Compute natural frequency, damping ratio, and poles of a system
<i>db2mag</i> (db)	Convert a gain in decibels (dB) to a magnitude
<i>isctime</i> (sys[, strict])	Check to see if a system is a continuous-time system
<i>isdtime</i> (sys[, strict])	Check to see if a system is a discrete time system

Continued on next page

Table 12 – continued from previous page

<code>issiso(sys[, strict])</code>	Check to see if a system is single input, single output
<code>issys(obj)</code>	Return True if an object is a system, otherwise False
<code>mag2db(mag)</code>	Convert a magnitude to decibels (dB)
<code>observable_form(xsys)</code>	Convert a system into observable canonical form
<code>pade(T[, n, numdeg])</code>	Create a linear system that approximates a delay.
<code>reachable_form(xsys)</code>	Convert a system into reachable canonical form
<code>sample_system(sysc, Ts[, method, alpha])</code>	Convert a continuous time system to discrete time
<code>ss2tf(sys)</code>	Transform a state space system to a transfer function.
<code>ssdata(sys)</code>	Return state space data objects for a system
<code>tf2ss(sys)</code>	Transform a transfer function to a state space system.
<code>tfdata(sys)</code>	Return transfer function data objects for a system
<code>timebase(sys[, strict])</code>	Return the timebase for an LTI system
<code>timebaseEqual(sys1, sys2)</code>	Check to see if two systems have the same timebase
<code>unwrap(angle[, period])</code>	Unwrap a phase angle to give a continuous curve
<code>use_fbs_defaults()</code>	Use <a href="#">Feedback Systems</a> (FBS) compatible settings
<code>use_matlab_defaults()</code>	Use MATLAB compatible configuration settings

### 3.10.1 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()`

### 3.10.2 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,  $z = T * x$

### 3.10.3 control.damp

`control.damp(sys, dprint=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
- **dprint** – 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, –  $wn = \text{abs}(\text{poles})$   $Z = -\text{real}(\text{poles})/\text{poles}$ .*
- *If the system is discrete, the discrete poles are mapped to their*
- *equivalent location in the s-plane via –  $s = \log10(\text{poles})/dt$*
- *and –  $wn = \text{abs}(s)$   $Z = -\text{real}(s)/wn$ .*

See also:

`pole()`

### 3.10.4 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*

### 3.10.5 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

### 3.10.6 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

### 3.10.7 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

### 3.10.8 control.issys

`control.issys(obj)`

Return True if an object is a system, otherwise False

### 3.10.9 control.mag2db

`control.mag2db(mag)`

Convert a magnitude to decibels (dB)

If *A* is magnitude,

$$\text{db} = 20 * \log_{10}(A)$$

**Parameters** `mag` (*float* or *ndarray*) – input magnitude or array of magnitudes

**Returns** **db** – corresponding values in decibels

**Return type** `float` or `ndarray`

### 3.10.10 `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:  $z = T * x$

### 3.10.11 `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  $\geq 0$ , specifies degree of numerator. If  $< 0$ , numerator degree is  $n + \text{numdeg}$ .

**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

### 3.10.12 `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:  $z = T * x$

### 3.10.13 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')
```

### 3.10.14 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)
```

### 3.10.15 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

### 3.10.16 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)
```

## 3.10.17 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

## 3.10.18 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.

## 3.10.19 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.

### 3.10.20 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  $\text{period}/2$  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]
```



## LTI system classes

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

<code>TransferFunction(num, den[, dt])</code>	A class for representing transfer functions
<code>StateSpace(A, B, C, D[, dt])</code>	A class for representing state-space models
<code>FRD(d, w)</code>	A class for models defined by frequency response data (FRD)

### 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.

**\_\_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 function, 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

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

### **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 continuous-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  $s = i * \text{omega}$ .

### **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  $s = i * \text{omega}$ , where  $\text{omega}$  is a list of angular frequencies, and is a sorted version of the input  $\text{omega}$ .

**horner** (*s*)

Evaluate the systems's transfer function for a complex variable

Returns a matrix of values evaluated at complex variable *s*.

**isctime** (*strict=False*)

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

#### Parameters

- **sys** (*LTI system*) – System to be checked
- **strict** (*bool (default = False)*) – If strict is True, make sure that timebase is not None

**isdttime** (*strict=False*)

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

**Parameters** **strict** (*bool (default = False)*) – If strict is True, make sure that timebase is not None

**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" }*) – 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  $T_s$

**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:

$$\mathbf{dx}/dt = \mathbf{A} \mathbf{x} + \mathbf{B} \mathbf{u} \quad \mathbf{y} = \mathbf{C} \mathbf{x} + \mathbf{D} \mathbf{u}$$

where  $\mathbf{u}$  is the input,  $\mathbf{y}$  is the output, and  $\mathbf{x}$  is the state.

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

Discrete-time state space systems 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*)

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

Construct a state space object.

The default constructor is `StateSpace(A, B, C, D)`, where  $\mathbf{A}$ ,  $\mathbf{B}$ ,  $\mathbf{C}$ ,  $\mathbf{D}$  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



<code>__init__(*args)</code>	StateSpace(A, B, C, D[, dt])
<code>append(other)</code>	Append a second model to the present model.
<code>damp()</code>	Natural frequency, damping ratio of system poles
<code>dcgain()</code>	Return the zero-frequency gain
<code>evalfr(omega)</code>	Evaluate a SS system's transfer function at a single frequency.
<code>feedback([other, sign])</code>	Feedback interconnection between two LTI systems.
<code>freqresp(omega)</code>	Evaluate the system's transfer func.
<code>horner(s)</code>	Evaluate the systems's transfer function for a complex variable
<code>isctime([strict])</code>	Check to see if a system is a continuous-time system
<code>isdttime([strict])</code>	Check to see if a system is a discrete-time system
<code>issiso()</code>	Check to see if a system is single input, single output
<code>minreal([tol])</code>	Calculate a minimal realization, removes unobservable and uncontrollable states
<code>pole()</code>	Compute the poles of a state space system.
<code>returnScipySignalLTI()</code>	Return a list of a list of scipy.signal.lti objects.
<code>sample(Ts[, method, alpha])</code>	Convert a continuous time system to discrete time
<code>zero()</code>	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,

$$G(j*\omega) = \text{mag} * \exp(j*\text{phase})$$

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

$$G(\exp(j*\omega*dt)) = \text{mag} * \exp(j*\text{phase}).$$

**omega:** 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** (*The magnitude (absolute value, not dB or log10) of the system*) – frequency response.
- **phase** (*The wrapped phase in radians of the system frequency*) – response.
- **omega** (*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 (default = False)*) – If strict is True, make sure that timebase is not None

#### **isdttime** (*strict=False*)

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

**Parameters** **strict** (*bool (default = False)*) – If strict is True, make sure that timebase is not None

#### **issiso** ()

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

#### **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` system

### 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.



---

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

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

**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  $2s^2 + 3s + 4$ .

**Examples**

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

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

## 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:

$$\begin{aligned}x[k+1] &= A \cdot x[k] + B \cdot u[k] \\ y[k] &= C \cdot x[k] + D \cdot u[k]\end{aligned}$$

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\_like* 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

<code>mag2db(mag)</code>	Convert a magnitude to decibels (dB)
<code>db2mag(db)</code>	Convert a gain in decibels (dB) to a magnitude
<code>c2d(sysc, Ts[, method])</code>	Return a discrete-time system
<code>ss2tf(sys)</code>	Transform a state space system to a transfer function.
<code>tf2ss(sys)</code>	Transform a transfer function to a state space system.
<code>tfdata(sys)</code>	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,

$$db = 20 * \log_{10}(A)$$

**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,

$$db = 20 * \log_{10}(A)$$

**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

<code>series(sys1, *sysn)</code>	Return the series connection (...)
<code>parallel(sys1, *sysn)</code>	Return the parallel connection $\text{sys1} + \text{sys2} (+ \text{sys3} + \dots)$
<code>feedback(sys1[, sys2, sign])</code>	Feedback interconnection between two I/O systems.
<code>negate(sys)</code>	Return the negative of a system.
<code>connect(sys, Q, inputv, outputv)</code>	Index-base interconnection of system
<code>append(sys1, sys2, ..., sysn)</code>	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  $\text{sys1} + \text{sys2} (+ \text{sys3} + \dots)$

**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 plant.
- **sys2** (scalar, *StateSpace*, *TransferFunction*, *FRD*) – The feedback plant (often a feedback controller).
- **sign** (scalar) – The sign of feedback. *sign* = -1 indicates negative feedback, and *sign* = 1 indicates positive feedback. *sign* is an optional argument; it assumes a value of -1 if not specified.

**Returns out**

**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.

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

## Examples

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

### 5.3.5 control.matlab.connect

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

Index-base interconnection of system

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

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

#### Parameters

- **sys** (StateSpace Transferfunction) – System to be connected
- **Q** (2d array) – Interconnection matrix. First column gives the input to be connected second column gives the output to be fed into this input. Negative values for the second column mean the feedback is negative, 0 means no connection is made
- **inputv** (1d array) – list of final external inputs
- **outputv** (1d array) – list of final external outputs

**Returns** *sys* – 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 = sp.mat([ [ 1, 2], [2, -1] ]) # basically feedback, output 2 in 1
>>> 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)
```

---

**Todo:** also implement for transfer function, zpk, etc.

---

## 5.4 System gain and dynamics

<code>dcgain(*args)</code>	Compute the gain of the system in steady state.
<code>pole(sys)</code>	Compute system poles.
<code>zero(sys)</code>	Compute system zeros.
<code>damp(sys[, doprint])</code>	Compute natural frequency, damping ratio, and poles of a system
<code>pzmap(sys[, Plot, title])</code>	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:  $y = \text{gain} \cdot u$

**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:

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

## 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, –  $\text{wn} = \text{abs}(\text{poles})$   $Z = -\text{real}(\text{poles})/\text{poles}$ .*
- *If the system is discrete, the discrete poles are mapped to their*
- *equivalent location in the s-plane via –  $s = \log10(\text{poles})/\text{dt}$*
- *and –  $\text{wn} = \text{abs}(s)$   $Z = -\text{real}(s)/\text{wn}$ .*

See also:

`pole()`

### 5.4.5 control.matlab.pzmap

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

Plot a pole/zero map for a linear system.

**Parameters**

- **sys** (*LTI (StateSpace or TransferFunction)*) – Linear system for which poles and zeros are computed.
- **Plot** (*bool*) – If `True` a graph is generated with Matplotlib, otherwise the poles and zeros are only computed and returned.

**Returns**

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

## 5.5 Time-domain analysis

<code>step(sys[, T, X0, input, output, return_x])</code>	Step response of a linear system
<code>impulse(sys[, T, X0, input, output, return_x])</code>	Impulse response of a linear system
<code>initial(sys[, T, X0, input, output, return_x])</code>	Initial condition response of a linear system
<code>lsim(sys[, U, T, X0])</code>	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

<code>bode(syslist[, omega, dB, Hz, deg, ...])</code>	Bode plot of the frequency response
<code>nyquist(syslist[, omega, Plot, color, labelFreq])</code>	Nyquist plot for a system
<code>nichols(syslist[, omega, grid])</code>	Nichols plot for a system
<code>margin(sysdata)</code>	Calculate gain and phase margins and associated crossover frequencies
<code>freqresp(sys, omega)</code>	Frequency response of an LTI system at multiple angular frequencies.
<code>evalfr(sys, x)</code>	Evaluate the transfer function of an LTI system for a single complex number <i>x</i> .

### 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='b'*, *labelFreq=0*, *\*args*, *\*\*kwargs*)  
Nyquist plot for a system

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

### Parameters

- **syslist** (*list of LTI*) – List of linear input/output systems (single system is OK)
- **omega** (*freq\_range*) – Range of frequencies (list or bounds) in rad/sec
- **Plot** (*boolean*) – If True, plot magnitude
- **labelFreq** (*int*) – Label every nth frequency on the plot
- **\*\*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` (*syslist*, *omega=None*, *grid=True*)

Nichols plot for a system

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

#### Parameters

- **syslist** (*list of LTI, or LTI*) – List of linear input/output systems (single system is OK)
- **omega** (*array\_like*) – Range of frequencies (list or bounds) in rad/sec
- **grid** (*boolean, optional*) – True if the plot should include a Nichols-chart grid. Default is True.

#### Returns

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

<code>rlocus(sys[, kvect, xlim, ylim, plotstr, ...])</code>	Root locus plot
<code>place(A, B, p)</code>	Place closed loop eigenvalues $K = \text{place}(A, B, p)$
<code>lqr(A, B, Q, R[, N])</code>	Linear quadratic regulator design

### 5.7.1 control.matlab.rlocus

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

Root locus plot

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

#### Parameters

- **sys** (*LTI object*) – Linear input/output systems (SISO only, for now)
- **kvect** (*list or ndarray, optional*) – List of gains to use in computing diagram



- **xlim** (*tuple or list, optional*) – control of x-axis range, normally with tuple (see `matplotlib.axes`)
- **ylim** (*tuple or list, optional*) – control of y-axis range
- **Plot** (*boolean, optional (default = True)*) – If True, plot root locus diagram.
- **PrintGain** (*boolean (default = True)*) – If True, report mouse clicks when close to the root-locus branches, calculate gain, damping and print
- **grid** (*boolean (default = False)*) – If True plot s-plane grid.

**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.place

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

Place closed loop eigenvalues  $K = \text{place}(A, B, p)$

**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  $\text{rank}(B)$  times.*

## Examples

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

See also:

`place_varga()`, `acker()`

### 5.7.3 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** (*2-d array*) – State feedback gains
- **S** (*2-d array*) – Solution to Riccati equation
- **E** (*1-d array*) – Eigenvalues of the closed loop system

## Examples

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

## 5.8 State-space (SS) models

<code>rss([states, outputs, inputs])</code>	Create a stable <i>continuous</i> random state space object.
<code>drss([states, outputs, inputs])</code>	Create a stable <i>discrete</i> random state space object.
<code>ctrb(A, B)</code>	Controllability matrix
<code>obsv(A, C)</code>	Observability matrix
<code>gram(sys, type)</code>	Gramian (controllability or observability)

### 5.8.1 control.matlab.ctrb

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

**Parameters** **B** (*A, B*) – 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, C*) – 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

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

### 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** Matrix

**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,  $\alpha \leq 0$  defines the stability boundary for the real part of `A`'s eigenvalues and for discrete-time systems,  $0 \leq \alpha \leq 1$  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* – 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(Y, 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

- $\mathbf{Y}$  (*array\_like*) – Output data
- $\mathbf{U}$  (*array\_like*) – Input data
- $\mathbf{M}$  (*integer*) – Number of Markov parameters to output

**Returns**  $\mathbf{H}$  – First  $M$  Markov parameters

**Return type** matrix

#### 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

- $\mathbf{T}$  (*number*) – time delay
- $\mathbf{n}$  (*positive integer*) – degree of denominator of approximation
- **numdeg** (*integer, or None (the default)*) – If `None`, numerator degree equals denominator degree If  $\geq 0$ , specifies degree of numerator If  $< 0$ , numerator degree is  $n + \text{numdeg}$

**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

<code>lyap(A, Q[, C, E])</code>	$X = \text{lyap}(A, Q)$ solves the continuous-time Lyapunov equation
<code>dlyap(A, Q[, C, E])</code>	$\text{dlyap}(A, Q)$ solves the discrete-time Lyapunov equation
<code>care(A, B, Q[, R, S, E])</code>	$(X, L, G) = \text{care}(A, B, Q, R=None)$ solves the continuous-time algebraic Riccati equation
<code>dare(A, B, Q, R[, S, E])</code>	$(X, L, G) = \text{dare}(A, B, Q, R)$ solves the discrete-time algebraic Riccati equation

### 5.11.1 control.matlab.lyap

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

$X = \text{lyap}(A, Q)$  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.

$X = \text{lyap}(A, Q, C)$  solves the Sylvester equation

$$AX + XQ + C = 0$$

where  $A$  and  $Q$  are square matrices.

$X = \text{lyap}(A, Q, None, E)$  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)`

$\text{dlyap}(A, Q)$  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.

$\text{dlyap}(A, Q, C)$  solves the Sylvester equation

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

where  $A$  and  $Q$  are square matrices.

$\text{dlyap}(A, Q, None, E)$  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)`

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

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

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  $A - B G$ .

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

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

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

### 5.11.4 control.matlab.dare

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

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

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

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

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

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

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

## 5.12 Additional functions

<code>gangof4(P, C[, omega])</code>	Plot the “Gang of 4” transfer functions for a system
<code>unwrap(angle[, period])</code>	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** ( $P, \cdot$ ) – 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  $\text{period}/2$  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

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

- `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/>



### C

`control`, [9](#)

`control.matlab`, [57](#)



## Symbols

`__init__()` (*control.FRD method*), 12  
`__init__()` (*control.StateSpace method*), 52  
`__init__()` (*control.TransferFunction method*), 49

## A

`acker()` (*in module control*), 32  
`append()` (*control.StateSpace method*), 53  
`append()` (*in module control*), 14  
`append()` (*in module control.matlab*), 67  
`augw()` (*in module control*), 40

## B

`balred()` (*in module control*), 37  
`balred()` (*in module control.matlab*), 81  
`bode()` (*in module control.matlab*), 72  
`bode_plot()` (*in module control*), 18

## C

`c2d()` (*in module control.matlab*), 61  
`canonical_form()` (*in module control*), 40  
`care()` (*in module control*), 30  
`care()` (*in module control.matlab*), 85  
`connect()` (*in module control*), 15  
`connect()` (*in module control.matlab*), 66  
`control` (*module*), 9  
`control.matlab` (*module*), 57  
`ctrb()` (*in module control*), 31  
`ctrb()` (*in module control.matlab*), 79

## D

`damp()` (*control.FRD method*), 12  
`damp()` (*control.StateSpace method*), 53  
`damp()` (*control.TransferFunction method*), 50  
`damp()` (*in module control*), 41  
`damp()` (*in module control.matlab*), 68  
`dare()` (*in module control*), 30  
`dare()` (*in module control.matlab*), 85  
`db2mag()` (*in module control*), 41

`db2mag()` (*in module control.matlab*), 61  
`dcgain()` (*control.FRD method*), 12  
`dcgain()` (*control.StateSpace method*), 53  
`dcgain()` (*control.TransferFunction method*), 50  
`dcgain()` (*in module control*), 25  
`dcgain()` (*in module control.matlab*), 67  
`dlyap()` (*in module control*), 31  
`dlyap()` (*in module control.matlab*), 84  
`drss()` (*in module control*), 14  
`drss()` (*in module control.matlab*), 60

## E

`era()` (*in module control*), 38  
`era()` (*in module control.matlab*), 82  
`eval()` (*control.FRD method*), 12  
`evalfr()` (*control.FRD method*), 13  
`evalfr()` (*control.StateSpace method*), 53  
`evalfr()` (*control.TransferFunction method*), 50  
`evalfr()` (*in module control*), 25  
`evalfr()` (*in module control.matlab*), 76

## F

`feedback()` (*control.FRD method*), 13  
`feedback()` (*control.StateSpace method*), 53  
`feedback()` (*control.TransferFunction method*), 50  
`feedback()` (*in module control*), 15  
`feedback()` (*in module control.matlab*), 65  
`forced_response()` (*in module control*), 20  
`FRD` (*class in control*), 11  
`frd()` (*in module control.matlab*), 59  
`freqresp()` (*control.FRD method*), 13  
`freqresp()` (*control.StateSpace method*), 53  
`freqresp()` (*control.TransferFunction method*), 50  
`freqresp()` (*in module control*), 26  
`freqresp()` (*in module control.matlab*), 75

## G

`gangof4()` (*in module control.matlab*), 85  
`gangof4_plot()` (*in module control*), 19

`gram()` (in module *control*), 32  
`gram()` (in module *control.matlab*), 79

## H

`h2syn()` (in module *control*), 33  
`hinfsyn()` (in module *control*), 33  
`horner()` (*control.StateSpace* method), 54  
`horner()` (*control.TransferFunction* method), 51  
`hsvd()` (in module *control*), 37  
`hsvd()` (in module *control.matlab*), 80

## I

`impulse()` (in module *control.matlab*), 70  
`impulse_response()` (in module *control*), 21  
`initial()` (in module *control.matlab*), 71  
`initial_response()` (in module *control*), 22  
`isctime()` (*control.FRD* method), 13  
`isctime()` (*control.StateSpace* method), 54  
`isctime()` (*control.TransferFunction* method), 51  
`isctime()` (in module *control*), 42  
`isdtime()` (*control.FRD* method), 13  
`isdtime()` (*control.StateSpace* method), 54  
`isdtime()` (*control.TransferFunction* method), 51  
`isdtime()` (in module *control*), 42  
`issiso()` (*control.FRD* method), 13  
`issiso()` (*control.StateSpace* method), 54  
`issiso()` (*control.TransferFunction* method), 51  
`issiso()` (in module *control*), 42  
`issys()` (in module *control*), 42

## L

`lqr()` (in module *control*), 34  
`lqr()` (in module *control.matlab*), 78  
`lsim()` (in module *control.matlab*), 71  
`lyap()` (in module *control*), 30  
`lyap()` (in module *control.matlab*), 84

## M

`mag2db()` (in module *control*), 42  
`mag2db()` (in module *control.matlab*), 61  
`margin()` (in module *control*), 27  
`margin()` (in module *control.matlab*), 74  
`markov()` (in module *control*), 39  
`markov()` (in module *control.matlab*), 83  
`minreal()` (*control.StateSpace* method), 54  
`minreal()` (*control.TransferFunction* method), 51  
`minreal()` (in module *control*), 36  
`minreal()` (in module *control.matlab*), 80  
`mixsyn()` (in module *control*), 35  
`modred()` (in module *control*), 38  
`modred()` (in module *control.matlab*), 82

## N

`negate()` (in module *control*), 16

`negate()` (in module *control.matlab*), 66  
`nichols()` (in module *control.matlab*), 74  
`nichols_plot()` (in module *control*), 20  
`nyquist()` (in module *control.matlab*), 73  
`nyquist_plot()` (in module *control*), 19

## O

`observable_form()` (in module *control*), 43  
`obsv()` (in module *control*), 31  
`obsv()` (in module *control.matlab*), 79

## P

`pade()` (in module *control*), 43  
`pade()` (in module *control.matlab*), 83  
`parallel()` (in module *control*), 17  
`parallel()` (in module *control.matlab*), 64  
`phase_crossover_frequencies()` (in module *control*), 28  
`phase_plot()` (in module *control*), 23  
`place()` (in module *control*), 35  
`place()` (in module *control.matlab*), 77  
`pole()` (*control.StateSpace* method), 54  
`pole()` (*control.TransferFunction* method), 51  
`pole()` (in module *control*), 28  
`pole()` (in module *control.matlab*), 68  
`pzmap()` (in module *control*), 29  
`pzmap()` (in module *control.matlab*), 69

## R

`reachable_form()` (in module *control*), 43  
`returnScipySignalLTI()` (*control.StateSpace* method), 54  
`returnScipySignalLTI()` (*control.TransferFunction* method), 51  
`rlocus()` (in module *control.matlab*), 76  
`root_locus()` (in module *control*), 29  
`rss()` (in module *control*), 13  
`rss()` (in module *control.matlab*), 60

## S

`sample()` (*control.StateSpace* method), 54  
`sample()` (*control.TransferFunction* method), 51  
`sample_system()` (in module *control*), 44  
`series()` (in module *control*), 17  
`series()` (in module *control.matlab*), 64  
`ss()` (in module *control*), 9  
`ss()` (in module *control.matlab*), 58  
`ss2tf()` (in module *control*), 44  
`ss2tf()` (in module *control.matlab*), 62  
`ssdata()` (in module *control*), 45  
`stability_margins()` (in module *control*), 27  
`StateSpace` (class in *control*), 52  
`step()` (in module *control.matlab*), 69



`step_response()` (*in module control*), 23

## T

`tf()` (*in module control*), 10

`tf()` (*in module control.matlab*), 57

`tf2ss()` (*in module control*), 45

`tf2ss()` (*in module control.matlab*), 63

`tfdata()` (*in module control*), 46

`tfdata()` (*in module control.matlab*), 63

`timebase()` (*in module control*), 46

`timebaseEqual()` (*in module control*), 46

`TransferFunction` (*class in control*), 49

## U

`unwrap()` (*in module control*), 47

`unwrap()` (*in module control.matlab*), 86

`use_fbs_defaults()` (*in module control*), 8

`use_matlab_defaults()` (*in module control*), 8

## Z

`zero()` (*control.StateSpace method*), 55

`zero()` (*control.TransferFunction method*), 52

`zero()` (*in module control*), 28

`zero()` (*in module control.matlab*), 68