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

**Features**

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

**Documentation**
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 package (control.matlab) is available that provides many of the common functions corresponding to commands available in the MATLAB Control Systems Toolbox.

1.2 Some Differences from MATLAB

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

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

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

1.3 Installation

The python-control package may be installed using pip, conda or the standard distutils/setuptools mechanisms. To install using pip:
Many parts of python-control will work without slycot, but some functionality is limited or absent, and installation of slycot is recommended.

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

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

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

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

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

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

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

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

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

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

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

## 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 *
```
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 Time series data

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

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

Types:

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

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

\[ T = \begin{bmatrix} t_1, & t_2, & t_3, & \ldots, & t_n \end{bmatrix} \]

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:

\[ U = \begin{bmatrix} u_1(t_1), & u_1(t_2), & u_1(t_3), & \ldots, & u_1(t_n) \\ u_2(t_1), & u_2(t_2), & u_2(t_3), & \ldots, & u_2(t_n) \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ u_i(t_1), & u_i(t_2), & u_i(t_3), & \ldots, & u_i(t_n) \end{bmatrix} \]
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 \((j, 1)\):

\[
X0 = 
\begin{bmatrix}
x_1 \\
x_2 \\
\vdots \\
\vdots \\
x_j 
\end{bmatrix}
\]

As all simulation functions return arrays, plotting is convenient:

```python
import matplotlib.pyplot as plt
import numpy as np

sys = ...  # define your system

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

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

```python
import matplotlib.pyplot as plt
import numpy as np

sys = ...  # define your system

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:

```python
ft = D * U
```

### 2.2 Package configuration

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

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

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

Functions that can be used to set standard configurations:

```python
use_fbs_defaults()  # Use Astrom and Murray compatible settings
use_matlab_defaults()  # Use MATLAB compatible configuration settings
```
2.2.1 control.use_fbs_defaults

control.use_fbs_defaults()

Use Astrom and Murray compatible settings

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

2.2.2 control.use_matlab_defaults

control.use_matlab_defaults()

Use MATLAB compatible configuration settings

• Bode plots plot gain in dB, phase in degrees, frequency in Hertz
Chapter 3

Function reference

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

### 3.1 System creation

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ss(*args)</code></td>
<td>Create a state space system.</td>
</tr>
<tr>
<td><code>tf(*args)</code></td>
<td>Create a transfer function system.</td>
</tr>
<tr>
<td><code>frd(*args)</code></td>
<td>Construct a Frequency Response Data model, or convert a system.</td>
</tr>
<tr>
<td><code>rss([states, outputs, inputs])</code></td>
<td>Create a stable <em>continuous</em> random state space object.</td>
</tr>
<tr>
<td><code>drss([states, outputs, inputs])</code></td>
<td>Create a stable <em>discrete</em> random state space object.</td>
</tr>
</tbody>
</table>

#### 3.1.1 control.ss

*control.ss(*args)*

Create a state space system.

The function accepts either 1, 4 or 5 parameters:

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

- `ss(A, B, C, D)` Create a state space system from the matrices of its state and output equations:
  
  \[
  \dot{x} = A \cdot x + B \cdot u \\
  y = C \cdot x + D \cdot u
  \]

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

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

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

**Parameters**

- `sys`: StateSpace or TransferFunction
  A linear system

  - `A`: array_like or string
    System matrix

  - `B`: array_like or string
    Control matrix

  - `C`: array_like or string
    Output matrix

  - `D`: array_like or string
    Feed forward matrix

- `dt`: If present, specifies the sampling period and a discrete time system is created

**Returns**

- `out`: :class:`StateSpace`
  The new linear system

**Raises**

- ValueError
  if matrix sizes are not self-consistent

**See also:**

- `tf`, `ss2tf`, `tf2ss`

**Examples**

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

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

### 3.1.2 control.tf

`control.tf(*args)`

Create a transfer function system. Can create MIMO systems.

The function accepts either 1 or 2 parameters:
tf(sys) Convert a linear system into transfer function form. Always creates a new system, even if sys is already a TransferFunction object.

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

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

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

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

Parameters sys: LTI (StateSpace or TransferFunction):

A linear system

num: array_like, or list of list of array_like:
Polynomial coefficients of the numerator

den: array_like, or list of list of array_like:
Polynomial coefficients of the denominator

Returns out: :class:`TransferFunction`:

The new linear system

Raises ValueError:

if num and den have invalid or unequal dimensions

TypeError:

if num or den are of incorrect type

See also:

ss, ss2tf, tf2ss

Notes

den[i][j] contains the polynomial coefficients of the denominator 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

```python
>>> # 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

control.frd(*args)

Construct a Frequency Response Data model, or convert a system

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

This function can be called in different ways:

frd(response, freqs) Create an frd model with the given response data, in the form of complex re-

sponse 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 list :
vector with frequencies

sys: LTI (StateSpace or TransferFunction) :
A linear system

Returns sys: FRD :
New frequency response system

See also:

ss, tf

3.1.4 control.rss

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

Create a stable continuous random state space object.

Parameters states: integer :
Number of state variables

inputs: integer :
Number of system inputs

outputs: integer :
Number of system outputs

Returns sys: StateSpace :
The randomly created linear system

Raises ValueError :
if any input is not a positive integer
See also:

\texttt{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

\texttt{control.drss(states=1, outputs=1, inputs=1)}

Create a stable \texttt{discrete} random state space object.

\textbf{Parameters}

\begin{itemize}
\item \texttt{states: integer} : \\
\hspace{1cm} Number of state variables
\item \texttt{inputs: integer} : \\
\hspace{1cm} Number of system inputs
\item \texttt{outputs: integer} : \\
\hspace{1cm} Number of system outputs
\end{itemize}

\textbf{Returns}

\begin{itemize}
\item \texttt{sys: StateSpace} : \\
\hspace{1cm} The randomly created linear system
\item \texttt{ValueError} : \\
\hspace{1cm} if any input is not a positive integer
\end{itemize}

See also:

\texttt{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

\textbf{append(*sys)}

Group models by appending their inputs and outputs

\textbf{connect(sys, Q, inputv, outputv)}

Index-base interconnection of system

\textbf{feedback(sys1[, sys2, sign])}

Feedback interconnection between two I/O systems.

\textbf{negate(sys)}

Return the negative of a system.

\textbf{parallel(sys1, sys2)}

Return the parallel connection \texttt{sys1} + \texttt{sys2}.

\textbf{series(sys1, sys2)}

Return the series connection \texttt{sys2 * sys1} for \texttt{-->} \texttt{sys1} \texttt{-->}.
3.2.1 control.append

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

Group models by appending their inputs and outputs

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

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

LTI systems to combine

**Returns** `sys`: LTI system:

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

**Examples**

```python
>>> 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

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

Index-base interconnection of system

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

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

**Parameters** `sys`: StateSpace Transferfunction:

System to be connected

**Q**: 2d array:

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

**inputv**: 1d array:

list of final external inputs

**outputv**: 1d array:

list of final external outputs

**Returns** `sys`: LTI system:

Connected and trimmed LTI system
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: 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: 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

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

3.2.5 control.parallel

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

Return the parallel connection sys1 + sys2.

Parameters

- sys1: scalar, StateSpace, TransferFunction, or FRD
- sys2: scalar, StateSpace, TransferFunction, or FRD

Returns

- out: scalar, StateSpace, or TransferFunction

Raises

ValueError:

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

See also:

series, feedback

Notes

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

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

Examples

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

3.2.6 control.series

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

Return the series connection sys2 * sys1 for → sys1 → sys2 →.

Parameters

- sys1: scalar, StateSpace, TransferFunction, or FRD
- sys2: scalar, StateSpace, TransferFunction, or FRD

Notes

This function is a wrapper for the __mul__ 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.
Returns out: scalar, StateSpace, or TransferFunction:

Raises ValueError:

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

See also:
parallel, feedback

Notes

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

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

Examples

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

3.3 Frequency domain plotting

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

3.3.1 control.bode_plot

```python
control.bode_plot (syslist, omega=0, dB=0, Hz=0, deg=0, Plot=True, omega_limits=0, omega_num=0, *args, **kwargs)
```

Bode plot for a system

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

Parameters

- **syslist**: linsys
  List of linear input/output systems (single system is OK)
- **omega**: freq_range
  Range of frequencies in rad/sec
- **dB**: boolean
  If True, plot result in dB
- **Hz**: boolean
  If True, plot frequency in Hz (omega must be provided in rad/sec)
- **deg**: boolean
If True, plot phase in degrees (else radians)

Plot : boolean
If True, plot magnitude and phase

omega_limits: tuple, list, … of two values :
Limits of the to generate frequency vector. If Hz=True the limits are in Hz otherwise in rad/s.

omega_num: int :
number of samples

*args, **kwargs: :
    Additional options to matplotlib (color, linestyle, etc)

Returns mag : array (list if len(syslist) > 1)
magnitude
phase : array (list if len(syslist) > 1)
phase in radians
omega : array (list if len(syslist) > 1)
frequency in rad/sec

Notes

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

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

Examples

```python
>>> 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

*args, **kwargs:

Additional options to matplotlib (color, linestyle, etc)

**Returns**

**real** : array
real part of the frequency response array

**imag** : array
imaginary part of the frequency response array

**freq** : array
frequencies

**Examples**

```python
>>> 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**

**P, C** : LTI
Linear input/output systems (process and control)

**omega** : array
Range of frequencies (list or bounds) in rad/sec

**Returns**

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.

3.3. Frequency domain plotting
3.4 Time domain simulation

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>forced_response</code></td>
<td>Simulate the output of a linear system.</td>
</tr>
<tr>
<td><code>impulse_response</code></td>
<td>Impulse response of a linear system</td>
</tr>
<tr>
<td><code>initial_response</code></td>
<td>Initial condition response of a linear system</td>
</tr>
<tr>
<td><code>step_response</code></td>
<td>Step response of a linear system</td>
</tr>
<tr>
<td><code>phase_plot</code></td>
<td>Phase plot for 2D dynamical systems</td>
</tr>
</tbody>
</table>

### 3.4.1 `control.forced_response`

```python
control.forced_response(sys[, T, U, X0, transpose])
```

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

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

See Time series data.

### 3.4.2 control.impulse_response

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

Impulse response of a linear system

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

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

**Parameters**

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

**Returns**

- `T`: array
  - Time values of the output
- `yout`: array
  - Response of the system
- `xout`: array
  - Individual response of each x variable

See also:

ForcedReponse, initial_response, step_response

---

3.4. Time domain simulation 21
Examples

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

### 3.4.3 control.initial_response

The `control.initial_response` function is used to calculate the 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

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

### 3.4.5 control.phase_plot

The `control.phase_plot` function is used to create a phase plot for 2D dynamical systems. It produces a vector field or stream line plot for a planar system.

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

**Parameters**

- **func**: callable(x, t, ...)
  Computes the time derivative of y (compatible with `odeint`). The function should be the same for as used for `scipy.integrate`. Namely, it should be a function of the form \( \frac{dx}{dt} = F(x, t) \) that accepts a state \( x \) of dimension 2 and returns a derivative \( \frac{dx}{dt} \) of dimension 2.

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

- **scale**: float, optional
  Scale size of arrows; default = 1

- **X0**: ndarray of initial conditions, optional
  List of initial conditions from which streamlines are plotted. Each initial condition should be a pair of numbers.

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

- **lingrid = N or (N, M)**: integer or 2-tuple of integers, optional
  If \( X0 \) is given and \( X, Y \) are missing, a grid of arrows is produced using the limits of the initial conditions, with \( N \) grid points in each dimension or \( N \) grid points in \( x \) and \( M \) grid points in \( y \).

- **lintime = N**: integer, optional
  Draw \( N \) arrows using equally space time points

- **logtime = (N, lambda)**: (integer, float), optional
  Draw \( N \) arrows using exponential time constant \( \lambda \)

- **timepts = [t1, t2, ...]**: array-like, optional
  A list of time points at which to evaluate the vector field.
Draw arrows at the given list times

**parms**: tuple, optional:

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

See also:

box_grid, Y

### 3.5 Block diagram algebra

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>series(sys1, sys2)</code></td>
<td>Return the series connection ( sys2 \ast sys1 ) for ( \rightarrow sys1 \rightarrow sys2 ).</td>
</tr>
<tr>
<td><code>parallel(sys1, sys2)</code></td>
<td>Return the parallel connection ( sys1 + sys2 ).</td>
</tr>
<tr>
<td><code>feedback(sys1[, sys2, sign])</code></td>
<td>Feedback interconnection between two I/O systems.</td>
</tr>
<tr>
<td><code>negate(sys)</code></td>
<td>Return the negative of a system.</td>
</tr>
</tbody>
</table>

### 3.6 Control system analysis

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

#### 3.6.1 control.dcgain

```python
control.dcgain(sys)
```

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

**Returns** `gain` : ndarray

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

#### 3.6.2 control.evalfr

```python
control.evalfr(sys, x)
```

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

To evaluate at a frequency, enter \( x = \omega j \), where \( \omega \) is the frequency in radians
Parameters

sys: StateSpace or TransferFunction:

Linear system

x: scalar:

Complex number

Returns fresp: ndarray:

See also:

cfrequresp, bode

Notes

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

Examples

```python
>>> 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:

evfr, 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

```python
>>> 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

```python
#>>> 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(*args)

Calculate gain and phase margins and associated crossover frequencies

Parameters

- `sysdata`: LTI system or (mag, phase, omega) sequence
  - `sys` [StateSpace or TransferFunction] Linear SISO system
  - `mag, phase, omega` [sequence of array_like] Input magnitude, phase (in deg.), and frequencies (rad/sec) from bode frequency response data

Returns

- `gm` [float] Gain margin
- `pm` [float] Phase margin (in degrees)
- `Wcg` [float] Gain crossover frequency (corresponding to phase margin)
- `Wcp` [float] Phase crossover frequency (corresponding to gain margin) (in rad/sec)

Margins are of SISO open-loop. If more than one crossover frequency is detected, returns the lowest corresponding margin.

Examples

```python
>>> sys = tf(1, [1, 2, 1, 0])
>>> gm, pm, Wcg, Wcp = 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 one margin is 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:
Gain margin crossover frequency (where phase crosses -180 degrees)

wp: float or array_like:
Phase margin crossover frequency (where gain crosses 0 dB)

ws: float or array_like:
Stability margin frequency (where Nyquist plot is closest to -1)

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

```python
>>> tf = TransferFunction([1], [1, 2, 3, 4])
>>> PhaseCrossoverFrequencies(tf)
(array([ 1.73205081, 0.   ]), array([-0.5 , 0.25]))
```
3.6.7 control.pole

```python
def control.pole(sys):
    Compute system poles.

    Parameters sys: StateSpace or TransferFunction:
        Linear system

    Returns poles: ndarray:
        Array that contains the system’s poles.

    Raises NotImplementedError:
        when called on a TransferFunction object

    See also:
        zero, TransferFunction.pole, StateSpace.pole
```

3.6.8 control.zer0

```python
def control.zero(sys):
    Compute system zeros.

    Parameters sys: StateSpace or TransferFunction:
        Linear system

    Returns zeros: ndarray:
        Array that contains the system’s zeros.

    Raises NotImplementedError:
        when called on a MIMO system

    See also:
        pole, StateSpace.zero, TransferFunction.zero
```

3.6.9 control.pzmap

```python
def 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)

Root locus plot

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

Parameters

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

Returns

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

3.7 Matrix computations

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

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

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

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

ctrb(A, B)  
Controllability matrix

obsv(A, C)  
Observability matrix

gram(sys, type)
Gramian (controllability or observability)

3.7.1 control.care

control.care(A, B, Q, R=None, S=None, E=None)
(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 \( A - B G \).

\[ (X,L,G) = \text{care}(A,B,Q,R,S,E) \]

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 \( A - B G \), \( E \).

### 3.7.2 control.dare

\[ \text{dare}(A, B, Q, R, S=None, E=None) \]

\[ (X,L,G) = \text{dare}(A,B,Q,R) \]

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 \( A - B G \), \( E \).

### 3.7.3 control.lyap

\[ \text{lyap}(A, Q, C=None, E=None) \]

\[ X = \text{lyap}(A,Q) \]

solves the continuous-time Lyapunov equation

\[ AX + X A^T + Q = 0 \]

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

\[ AX + X Q + C = 0 \]

where \( A \) and \( Q \) are square matrices.

\[ X = \text{lyap}(A,Q,\text{None},E) \]

solves the generalized continuous-time Lyapunov equation

\[ AX E^T + E X A^T + Q = 0 \]

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

### 3.7.4 control.dlyap

\[ \text{dlyap}(A, Q, \text{C=None}, \text{E=None}) \]

\[ dlyap(A,Q) \]

solves the discrete-time Lyapunov equation

\[ A X A^T - X + Q = 0 \]
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 A, B: array_like or string :
Dynamics and input matrix of the system

Returns C: matrix :
Controllability matrix

Examples

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

3.7.6 control obsv

control.obsv(A, C)
Observability matrix

Parameters A, C: array_like or string :
Dynamics and output matrix of the system

Returns O: matrix :
Observability matrix

Examples

```python
>>> 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: array:
Gramian of system

Raises ValueError:
• if system is not instance of StateSpace class
• if type is not ‘c’, ‘o’, ‘cf’ or ‘of’
• if system is unstable (sys.A has eigenvalues not in left half plane)

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

Examples

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

3.8 Control system synthesis

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>acker(A, B, poles)</td>
<td>Pole placement using Ackermann method</td>
</tr>
<tr>
<td>h2syn(P, nmeas, ncon)</td>
<td>(H_2) control synthesis for plant P.</td>
</tr>
<tr>
<td>hinfysyn(P, nmeas, ncon)</td>
<td>(H_{\infty}) control synthesis for plant P.</td>
</tr>
<tr>
<td>lqr(*args, **keywords)</td>
<td>Linear quadratic regulator design</td>
</tr>
<tr>
<td>place(A, B, p)</td>
<td>Place closed loop eigenvalues</td>
</tr>
</tbody>
</table>

3.8.1 control.acker

control.acker(A, B, poles)
Pole placement using Ackermann method

Call: \(K = \text{acker}(A, B, \text{poles})\)

Parameters A, B: 2-d arrays
State and input matrix of the system

poles: 1-d list:
Desired eigenvalue locations

Returns K: matrix:
Gains such that \(A - B K\) has given eigenvalues
3.8.2 control.h2syn

```python
control.h2syn(P, nmeas, ncon)
```
H_2 control synthesis for plant P.

Parameters
- **P**: partitioned lti plant (State-space sys)
  - `nmeas`: number of measurements (input to controller)
  - `ncon`: number of control inputs (output from controller)

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

Raises
- `ImportError`
  - if slycot routine sb10hd is not loaded

See also:
- `StateSpace`

Examples

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

3.8.3 control.hinfsyn

```python
control.hinfsyn(P, nmeas, ncon)
```
H_{inf} control synthesis for plant P.

Parameters
- **P**: partitioned lti plant
  - `nmeas`: number of measurements (input to controller)
  - `ncon`: number of control inputs (output from controller)

Returns
- **K**: controller to stabilize P (State-space sys)
- **CL**: closed loop system (State-space sys)
- **gam**: infinity norm of closed loop system
- **rcond**: 4-vector, reciprocal condition estimates of:
  - 1: control transformation matrix
  - 2: measurement transformation matrix
  - 3: X-Ricatti equation
  - 4: Y-Ricatti equation

Todo: document significance of rcond

Raises
- `ImportError`
  - if slycot routine sb10ad is not loaded

See also:
- `StateSpace`
Examples

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

### 3.8.4 control.lqr

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

Linear quadratic regulator design

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

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

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

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

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

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

**Returns**  
- `K`: 2-d array  
  State feedback gains
- `S`: 2-d array  
  Solution to Riccati equation
- `E`: 1-d array  
  Eigenvalues of the closed loop system

**Examples**

```python
>>> K, S, E = lqr(sys, Q, R, [N])
>>> K, S, E = lqr(A, B, Q, R, [N])
```
3.8.5 control.place

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

Place closed loop eigenvalues

Parameters

- **A**: 2-d array
  - Dynamics matrix
- **B**: 2-d array
  - Input matrix
- **p**: 1-d list
  - Desired eigenvalue locations

Returns

- **K**: 2-d array
  - Gains such that $A - BK$ has given eigenvalues

Examples

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

3.9 Model simplification tools

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

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:** StateSpace or TransferFunction:

Cleaned model

### 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.


**Parameters**

- **sys:** StateSpace:
  
  Original system to reduce

- **orders:** integer or array of integer:
  
  Desired order of reduced order model (if a vector, returns a vector of systems)

- **method:** string:
  
  Method of removing states, either 'truncate' or 'matchdc'.

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

**Returns rsys:** StateSpace:

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

**Raises** ValueError:

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

**ImportError:**

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

**ValueError:**

- if there are more unstable modes than any value in orders

**Examples**

```python
>>> 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 : Matrix

A list of Hankel singular values

**See also:**

gram

**Notes**

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

**Examples**

```python
>>> 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: StateSpace :

A reduced order model

**Raises**

ValueError :

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

**Examples**

```python
>>> 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 $YY$.

**Parameters**
- **YY**: array
  $nout \times nin$ dimensional impulse-response data
- **m**: integer
  Number of rows in Hankel matrix
- **n**: integer
  Number of columns in Hankel matrix
- **nin**: integer
  Number of input variables
- **nout**: integer
  Number of output variables
- **r**: integer
  Order of model

**Returns**
- **sys**: StateSpace
  A reduced order model $sys=ss(Ar,Br,Cr,Dr)$

**Examples**

```python
def era(YY, m, n, nin, nout, r):
    # ERA code goes here
```

3.9.6 control.markov

control.markov(Y, U, M)
Calculate the first $M$ Markov parameters $[D CB CAB \ldots ]$ from input $U$, output $Y$.

**Parameters**
- **Y**: array_like
  Output data
- **U**: array_like
  Input data
- **M**: integer
  Number of Markov parameters to output

**Returns**
- **H**: matrix
  First $M$ Markov parameters
Notes

Currently only works for SISO

Examples

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

3.10 Utility functions and conversions

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

3.10.1 control.unwrap

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

Unwrap a phase angle to give a continuous curve

**Parameters**

- **angle**: array_like
  
  Array of angles to be unwrapped

- **period**: float, optional
  
  Period (defaults to 2*pi)

**Returns**

- **angle_out**: array_like
  
  Output array, with jumps of period/2 eliminated
Examples

```python
>>> 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]
```

3.10.2 control.db2mag

`control.db2mag(db)`

Convert a gain in decibels (dB) to a magnitude

If `A` is magnitude,

\[ \text{db} = 20 \times \log_{10}(A) \]

**Parameters**

- `db`: float or ndarray
  input value or array of values, given in decibels

**Returns**

- `mag`: float or ndarray
  corresponding magnitudes

3.10.3 control.mag2db

`control.mag2db(mag)`

Convert a magnitude to decibels (dB)

If `A` is magnitude,

\[ \text{db} = 20 \times \log_{10}(A) \]

**Parameters**

- `mag`: float or ndarray
  input magnitude or array of magnitudes

**Returns**

- `db`: float or ndarray
  corresponding values in decibels

3.10.4 control.damp

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

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

The function takes 1 or 2 parameters

**Parameters**

- `sys`: LTI (StateSpace or TransferFunction):
  A linear system object

  `doprint`:
  if true, print table with values

**Returns**

- `wn`: array:
  Natural frequencies of the poles

3.10. Utility functions and conversions
damping: array:
   Damping values
poles: array:
   Pole locations
See also:
   pole

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)

3.10.8 control.issys

control.issys(obj)
   Return True if an object is a system, otherwise False

3.10.9 control.pade

control.pade(T, n=1, numdeg=None)
   Create a linear system that approximates a delay.
   Return the numerator and denominator coefficients of the Pade approximation.
   Parameters T: number
      time delay
n : positive integer
degree of denominator of approximation

numdeg: integer, or None (the default):
If None, numerator degree equals denominator degree If >= 0, specifies degree of nu-
merator If < 0, numerator degree is n+numdeg

Returns num, den : array
Polynomial coefficients of the delay model, in descending powers of s.

Notes

Based on:
1. Algorithm 11.3.1 in Golub and van Loan, “Matrix Computation” 3rd. Ed. pp. 572-574

3.10.10 control.sample_system

control.sample_system(sysc, Ts, method='zoh', alpha=None)
Convert a continuous time system to discrete time
Creates a discrete time system from a continuous time system by sampling. Multiple methods of conversion are supported.

Parameters sysc : linsys
Continuous time system to be converted
Ts : real
Sampling period
method : string
Method to use for conversion: ‘matched’, ‘tustin’, ‘zoh’ (default)

Returns sysd : linsys
Discrete time system, with sampling rate Ts

Notes

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

Examples

```python
>>> sysc = TransferFunction([1], [1, 2, 1])
>>> sysd = sample_system(sysc, 1, method='matched')
```
3.10.11 control.canonical_form

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

Parameters xsys : StateSpace object
System to be transformed, with state ‘x’
form : String
Canonical form for transformation. Chosen from:
• ‘reachable’ - reachable canonical form
• ‘observable’ - observable canonical form
• ‘modal’ - modal canonical form [not implemented]

Returns zsys : StateSpace object
System in desired canonical form, with state ‘z’
T : matrix
Coordinate transformation matrix, z = T * x

3.10.12 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

3.10.13 controlreachable_form

controlreachable_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
3.10.14 control.ss2tf

control.ss2tf(*args)
Transform a state space system to a transfer function.

The function accepts either 1 or 4 parameters:

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

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

Parameters sys: StateSpace :
A linear system
A: array_like or string :
System matrix
B: array_like or string :
Control matrix
C: array_like or string :
Output matrix
D: array_like or string :
Feedthrough matrix

Returns out: TransferFunction :
New linear system in transfer function form

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

TypeError :
if sys is not a StateSpace object

See also:
tf, ss, tf2ss

Examples

```python
>>> 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)`: list of matrices

  State space data for the system

3.10.16 control.tf2ss

`control.tf2ss(*args)`

Transform a transfer function to a state space system.

The function accepts either 1 or 2 parameters:

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

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

  For details see: `tf()`

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

  A linear system

- `num`: array_like, or list of list of array_like

  Polynomial coefficients of the numerator

- `den`: array_like, or list of list of array_like

  Polynomial coefficients of the denominator

**Returns**
- `out`: StateSpace

  New linear system in state space form

**Raises**
- ValueError

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

- TypeError

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

**See also:**
- `ss, tf, ss2tf`

**Examples**

```python
>>> 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)
```
```python
>>> 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)`: numerator and denominator arrays
    - Transfer function coefficients (SISO only)

### 3.10.18 control.timebase

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

Return the timebase for an LTI system

```python
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

```python
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.
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.

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>TransferFunction(*args)</code></td>
<td>A class for representing transfer functions</td>
</tr>
<tr>
<td><code>StateSpace(*args)</code></td>
<td>A class for representing state-space models</td>
</tr>
<tr>
<td><code>FRD(*args, **kwargs)</code></td>
<td>A class for models defined by Frequency Response Data (FRD)</td>
</tr>
</tbody>
</table>

### 4.1 control.TransferFunction

```python
class control.TransferFunction(*args)
A class for representing transfer functions
```

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

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

```python
>>> 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.

```python
__init__(*args)
Construct a transfer function.
```

The default constructor is TransferFunction(num, den), where num and den are lists of lists of arrays containing polynomial coefficients. To create a discrete time transfer function, use TransferFunction(num, den, dt). To call the copy constructor, call TransferFunction(sys), where sys is a TransferFunction object (continuous or discrete).
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 : ndarray
        The zero-frequency gain

evalfr(omega)
    Evaluate a transfer function at a single angular frequency.
    self.evalfr(omega) returns the value of the transfer function matrix with input value \( s = i \times \omega \).

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

dfreqresp(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 \( s = i \times \omega \), where omega is a list of angular frequencies, and is a sorted version of the input omega.

horner(s)
    Evaluate the systems’s transfer function for a complex variable
    Returns a matrix of values evaluated at complex variable s.

isctime(strict=False)
    Check to see if a system is a continuous-time system
    
    Parameters sys : LTI system
        System to be checked
        
        strict: bool (default = False) :
            If strict is True, make sure that timebase is not None

isdtime(strict=False)
    Check to see if a system is a discrete-time system
    
    Parameters strict: bool (default = False) :
        If strict is True, make sure that timebase is not None

minreal(tol=None)
    Remove cancelling pole/zero pairs from a transfer function

pole()
    Compute the poles of a transfer function.

returnScipySignalLTI()
    Return a list of a list of scipy.signal.lti objects.
    
    For instance,
    
    >>> out = tfobject.returnScipySignalLTI()
    >>> out[3][5]

    is a signal.scipy.lti object corresponding to the transfer function from the 6th input to the 4th output.
sample (Ts, method='zoh', alpha=None)
Convert a continuous-time system to discrete time

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

Parameters

Ts : float
Sampling period

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

alpha : float within [0, 1]
The generalized bilinear transformation weighting parameter, which should only be specified with method="gbt", and is ignored otherwise

Returns

sysd : StateSpace system
Discrete time system, with sampling rate Ts

Notes

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

Examples

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

zero ()
Compute the zeros of a transfer function.

4.2 control.StateSpace

class control.StateSpace (*args)
A class for representing state-space models

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

\[
dx/dt = Ax + Bu \quad y = Cx + Du
\]

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

The main data members are the A, B, C, and D matrices. The class also keeps track of the number of states (i.e., the size of A).
Discrete-time state space system are implemented by using the ‘dt’ instance variable and setting it to the sampling period. If ‘dt’ is not None, then it must match whenever two state space systems are combined. Setting dt = 0 specifies a continuous system, while leaving dt = None means the system timebase is not specified. If ‘dt’ is set to True, the system will be treated as a discrete time system with unspecified sampling time.

```python
__init__ (*args)
    Construct a state space object.

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

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

dcgain ()
    Return the zero-frequency gain

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

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

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

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

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

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

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

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

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

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

    Returns a matrix of values evaluated at complex variable s.

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

        Parameters sys : LTI system
            System to be checked

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

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

        Parameters strict: bool (default = False) :
            If strict is True, make sure that timebase is not None
**minreal** *(tol=0.0)*

Calculate a minimal realization, removes unobservable and uncontrollable states

**pole()**

Compute the poles of a state space system.

**returnScipySignalLTI()**

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

For instance,

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

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

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

Convert a continuous time system to discrete time

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

**Parameters**

- Ts : float
  Sampling period
- method : {'gbt', 'bilinear', 'euler', 'backward_diff', 'zoh'}
  Which method to use:
  - gbt: generalized bilinear transformation
  - bilinear: Tustin's approximation ("gbt" with alpha=0.5)
  - euler: Euler (or forward differencing) method ("gbt" with alpha=0)
  - backward_diff: Backwards differencing ("gbt" with alpha=1.0)
  - zoh: zero-order hold (default)
- alpha : float within [0, 1]
  The generalized bilinear transformation weighting parameter, which should only be specified with method="gbt", and is ignored otherwise

**Returns**

- sysd : StateSpace system
  Discrete time system, with sampling rate Ts

**Notes**

Uses the command ‘cont2discrete’ from scipy.signal

**Examples**

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

**zero()**

Compute the zeros of a state space system.
4.3 control.FRDM

class control.FRDM(*args, **kwargs)
   A class for models defined by Frequency Response Data (FRD)

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

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

means that the frequency response from the 6th input to the 3rd output at the frequencies defined in omega is set

   to the array above, i.e. the rows represent the outputs and the columns represent the inputs.

   `__init__(*args, **kwargs)`
   Construct an FRD object

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

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

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

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

dcgain()  
   Return the zero-frequency gain

   `evalfr(omega)`
   Evaluate a transfer function at a single angular frequency.

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

   Note that a “normal” FRD only returns values for which there is an entry in the omega vector. An interpo-
   lating FRD can return intermediate values.

   feedback(other=1, sign=-1)
   Feedback interconnection between two FRD objects.

   freqresp(omega)
   Evaluate a transfer function at a list of angular frequencies.

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

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

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

   Parameters sys : LTI system

   System to be checked

   strict : bool (default = False) :

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

   isdtime(strict=False)
   Check to see if a system is a discrete-time system
Parameters \texttt{strict}: \texttt{bool} (default = \texttt{False}) : 

If \texttt{strict} is \texttt{True}, make sure that \texttt{timebase} is not \texttt{None}
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

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>tf(*args)</code></td>
<td>Create a transfer function system. Can create MIMO systems.</td>
</tr>
<tr>
<td><code>ss(*args)</code></td>
<td>Create a state space system.</td>
</tr>
<tr>
<td><code>frd(*args)</code></td>
<td>Construct a Frequency Response Data model, or convert a system</td>
</tr>
<tr>
<td><code>rss([states, outputs, inputs])</code></td>
<td>Create a stable continuous random state space object.</td>
</tr>
<tr>
<td><code>drss([states, outputs, inputs])</code></td>
<td>Create a stable discrete random state space object.</td>
</tr>
</tbody>
</table>

5.1.1 control.matlab.tf

control.matlab.tf(*args)

Create a transfer function system. Can create MIMO systems.

The function accepts either 1 or 2 parameters:

- `tf(sys)` Convert a linear system into transfer function form. Always creates a new system, even if sys is already a TransferFunction object.
- `tf(num, den)` Create a transfer function system from its numerator and denominator polynomial coefficients.

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

To create a MIMO system, `num` and `den` need to be 2D nested lists of array_like objects. (A 3 dimensional data structure in total.) (For details see note below.)
**tf(num, den, dt)** Create a discrete time transfer function system; dt can either be a positive number indicating the sampling time or ‘True’ if no specific timebase is given.

**Parameters**
- **sys**: LTI (StateSpace or TransferFunction)
  - A linear system
- **num**: array_like, or list of list of array_like
  - Polynomial coefficients of the numerator
- **den**: array_like, or list of list of array_like
  - Polynomial coefficients of the denominator

**Returns**
- **out**: :class:`TransferFunction`
  - The new linear system

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

**See also:**
- `ss`, `ss2tf`, `tf2ss`

**Notes**

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

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

**Examples**

```python
>>> # 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(*args)`

Create a state space system.

The function accepts either 1, 4 or 5 parameters:
ss(sys) Convert a linear system into space system form. Always creates a new system, even if sys is already a StateSpace object.

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

\[
\dot{x} = Ax + Bu \\
y = Cx + Du
\]

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

\[
x[k+1] = Ax[k] + Bu[k] \\
y[k] = Cx[k] + Du[k]
\]

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 :
A system is created

Returns out: :class:`StateSpace` :
The new linear system

Raises ValueError :
if matrix sizes are not self-consistent

See also: tf, ss2tf, tf2ss

Examples

```python
>>> # 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(*args)

Construct a Frequency Response Data model, or convert a system

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

This function can be called in different ways:

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

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

Parameters
- **response**: array_like, or list : complex vector with the system response
- **freq**: array_like or list : vector with frequencies
- **sys**: LTI (StateSpace or TransferFunction) : A linear system

Returns
- **sys**: FRD : New frequency response system

See also:
- *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**: StateSpace : The randomly created linear system

Raises
- **ValueError** : if any input is not a positive integer

See also:
- *drss*
Notes

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

5.1.5 control.matlab.drss

control.matlab.\texttt{drss}(\texttt{states}=$1$, \texttt{outputs}=$1$, \texttt{inputs}=$1$)

Create a stable discrete random state space object.

\begin{itemize}
\item \textbf{Parameters} \texttt{states}: integer :
  Number of state variables
\item \texttt{inputs}: integer :
  Number of system inputs
\item \texttt{outputs}: integer :
  Number of system outputs
\end{itemize}

\begin{itemize}
\item \textbf{Returns} \texttt{sys}: StateSpace :
  The randomly created linear system
\item \textbf{Raises} \texttt{ValueError} :
  if any input is not a positive integer
\end{itemize}

\begin{itemize}
\item \textbf{See also:} \texttt{rss}
\end{itemize}

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{mag2db}(mag)</td>
<td>Convert a magnitude to decibels (dB)</td>
</tr>
<tr>
<td>\texttt{db2mag}(db)</td>
<td>Convert a gain in decibels (dB) to a magnitude</td>
</tr>
<tr>
<td>\texttt{c2d}(sysc, Ts[, method])</td>
<td>Return a discrete-time system</td>
</tr>
<tr>
<td>\texttt{ss2tf}(*args)</td>
<td>Transform a state space system to a transfer function.</td>
</tr>
<tr>
<td>\texttt{tf2ss}(*args)</td>
<td>Transform a transfer function to a state space system.</td>
</tr>
<tr>
<td>\texttt{tfdata}(sys)</td>
<td>Return transfer function data objects for a system</td>
</tr>
</tbody>
</table>

5.2.1 control.matlab.mag2db

control.matlab.\texttt{mag2db}(\texttt{mag})

Convert a magnitude to decibels (dB)

If $A$ is magnitude,

$$
\text{db} = 20 \times \log_{10}(A)
$$
Parameters **mag**: float or ndarray
input magnitude or array of magnitudes

Returns **db**: float or ndarray
corresponding values in decibels

### 5.2.2 control.matlab.db2mag

```python
control.matlab.db2mag(db)
```
Convert a gain in decibels (dB) to a magnitude

If A is magnitude,

\[
db = 20 \times \log_{10}(A)
\]

Parameters **db**: float or ndarray
input value or array of values, given in decibels

Returns **mag**: float or ndarray
corresponding magnitudes

### 5.2.3 control.matlab.c2d

```python
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

```python
control.matlab.ss2tf(*args)
```
Transform a state space system to a transfer function.

The function accepts either 1 or 4 parameters:

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

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

For details see: **ss()**

Parameters **sys**: StateSpace:
A linear system

A: array_like or string :
    System matrix
B: array_like or string :
    Control matrix
C: array_like or string :
    Output matrix
D: array_like or string :
    Feedthrough matrix

Returns out: TransferFunction :
    New linear system in transfer function form

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

TypeError :
    if sys is not a StateSpace object

See also:
tf, ss, tf2ss

Examples

```python
>>> 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(*args)
    Transform a transfer function to a state space system.

    The function accepts either 1 or 2 parameters:

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

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

    For details see: tf()

Parameters sys: LTI (StateSpace or TransferFunction) :
A linear system

**num**: array_like, or list of list of array_like:
Polynomial coefficients of the numerator

**den**: array_like, or list of list of array_like:
Polynomial coefficients of the denominator

**Returns out**: StateSpace:
New linear system in state space form

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

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

**See also**: `ss`, `tf`, `ss2tf`

**Examples**

```python
>>> 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)`: numerator and denominator arrays:
  Transfer function coefficients (SISO only)

### 5.3 System interconnections

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>series(sys1, sys2)</code></td>
<td>Return the series connection $sys2 * sys1$ for $sys1 \rightarrow sys2 \rightarrow$</td>
</tr>
<tr>
<td><code>parallel(sys1, sys2)</code></td>
<td>Return the parallel connection $sys1 + sys2$.</td>
</tr>
<tr>
<td><code>feedback(sys1[, sys2, sign])</code></td>
<td>Feedback interconnection between two I/O systems.</td>
</tr>
</tbody>
</table>

Continued on next page
5.3.1 control.matlab.series

control.matlab.series(sys1, sys2)

Return the series connection sys2 * sys1 for \(\rightarrow sys1 \rightarrow sys2 \rightarrow\).

Parameters
- sys1: scalar, StateSpace, TransferFunction, or FRD
- sys2: scalar, StateSpace, TransferFunction, or FRD

Returns
- out: scalar, StateSpace, or TransferFunction

Raises ValueError:
- if sys2.inputs does not equal sys1.outputs if sys1.dt is not compatible with sys2.dt

See also:
- parallel, feedback

Notes

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

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

Examples

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

5.3.2 control.matlab.parallel

control.matlab.parallel(sys1, sys2)

Return the parallel connection sys1 + sys2.

Parameters
- sys1: scalar, StateSpace, TransferFunction, or FRD
- sys2: scalar, StateSpace, TransferFunction, or FRD

Returns
- out: scalar, StateSpace, or TransferFunction

Raises ValueError:
- if sys1 and sys2 do not have the same numbers of inputs and outputs

See also:
- series, feedback
Notes

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

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

Examples

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

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`: 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: 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

```python
>>> 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: LTI system

Connected and trimmed LTI system
Examples

```python
>>> 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(*sys*)

Group models by appending their inputs and outputs

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

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

LTI systems to combine

**Returns** sys: LTI system :

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

Examples

```python
>>> 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

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>dcgain</em>(<em>args</em>)</td>
<td>Compute the gain of the system in steady state.</td>
</tr>
<tr>
<td><em>pole</em>(sys)</td>
<td>Compute system poles.</td>
</tr>
<tr>
<td><em>zero</em>(sys)</td>
<td>Compute system zeros.</td>
</tr>
<tr>
<td><em>damp</em>(sys[, doprint])</td>
<td>Compute natural frequency, damping ratio, and poles of a system</td>
</tr>
<tr>
<td><em>pzmap</em>(sys[, Plot, title])</td>
<td>Plot a pole/zero map for a linear system.</td>
</tr>
</tbody>
</table>

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

Returns  gain: ndarray:
  The gain of each output versus each input: \( y = gain \cdot u \)

Notes

This function is only useful for systems with invertible system matrix A.
All systems are first converted to state space form. The function then computes:

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

### 5.4.2 control.matlab.pole

control.matlab.pole(sys)
Compute system poles.

Parameters  sys: StateSpace or TransferFunction:
  Linear system

Returns  poles: ndarray:
  Array that contains the system’s poles.

Raises  NotImplementedError:
  when called on a TransferFunction object

See also:
zero, TransferFunction.pole, StateSpace.pole

### 5.4.3 control.matlab.zero

control.matlab.zero(sys)
Compute system zeros.

Parameters  sys: StateSpace or TransferFunction:
  Linear system

Returns  zeros: ndarray:
  Array that contains the system’s zeros.

Raises  NotImplementedError:
when called on a MIMO system

See also:

pole, StateSpace.zero, TransferFunction.zero

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

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
### 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**

```python
>>> 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 (MIMO), 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

```python
>>> 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**

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

## 5.6 Frequency-domain analysis

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

### 5.6.1 control.matlab.bode

**control.matlab.bode(*args, **keywords)**
Bode plot of the frequency response
Plots a bode gain and phase diagram

**Parameters**
- **sys**: LTI, or list of LTI

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

- **omega**: freq_range :

  Range of frequencies in rad/s

- **dB** : boolean

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

deg : boolean
  If True, return phase in degrees (else radians)

Plot : boolean
  If True, plot magnitude and phase

Examples

```python
>>> 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

*args, **kwargs : 
  Additional options to matplotlib (color, linestyle, etc)

Returns real : array
  real part of the frequency response array

imag : array

5.6. Frequency-domain analysis 75
imaginary part of the frequency response array

freq : array
  frequencies

Examples

```python
>>> 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
  None :

5.6.4 control.matlab.margin

control.matlab.margin(*args)
Calculate gain and phase margins and associated crossover frequencies

Parameters
  sysdata: LTI system or (mag, phase, omega) sequence :
  sys  [StateSpace or TransferFunction] Linear SISO system
  mag, phase, omega [sequence of array_like] Input magnitude, phase (in deg.), and frequencies (rad/sec) from bode frequency response data

Returns
  gm : float
    Gain margin
  pm [float] Phase margin (in degrees)
  Wcg [float] Gain crossover frequency (corresponding to phase margin)
  Wcp [float] Phase crossover frequency (corresponding to gain margin) (in rad/sec)

Margins are of SISO open-loop. If more than one crossover frequency is detected, returns the lowest corresponding margin.
Examples

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

```python
>>> 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

```python
#>>> 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. Frequency-domain analysis 77
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: ndarray:
See also:
freqresp, bode

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

Examples

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

Todo: Add example with MIMO system

5.7 Compensator design

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

5.7.1 control.matlab.rlocus

control.matlab.rlocus(sys[, kvect=None, xlim=None, ylim=None, plotstr='-', Plot=True, Print-Gain=True])
Root locus plot
Calculate the root locus by finding the roots of 1+k*TF(s) where TF is self.num(s)/self.den(s) and each k is an element of kvect.

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

**xlim**: tuple or list, optional
control of x-axis range, normally with tuple (see matplotlib.axes)

**ylim**: tuple or list, optional
control of y-axis range

**Plot**: boolean, optional (default = True)
If True, plot magnitude and phase

**PrintGain**: boolean (default = True)
If True, report mouse clicks when close to the root-locus branches, calculate gain, damping and print

**Returns**

**rilist**: 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

**Parameters**

**A**: 2-d array
Dynamics matrix

**B**: 2-d array
Input matrix

**p**: 1-d list
Desired eigenvalue locations

**Returns**

**K**: 2-d array
Gains such that \(A - BK\) has given eigenvalues

### Examples

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

### 5.7.3 control.matlab.lqr

control.matlab.lqr\(*args, **keywords\)
Linear quadratic regulator design
The \texttt{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:

- \texttt{lqr(sys, Q, R)}
- \texttt{lqr(sys, Q, R, N)}
- \texttt{lqr(A, B, Q, R)}
- \texttt{lqr(A, B, Q, R, N)}

where \texttt{sys} is an \textit{LTI} object, and \texttt{A, B, Q, R,} and \texttt{N} are 2d arrays or matrices of appropriate dimension.

**Parameters**

- **A, B**: 2-d array:
  Dynamics and input matrices

- **sys**: \textit{LTI} (\textit{StateSpace} or \textit{TransferFunction}):
  Linear I/O system

- **Q, R**: 2-d array:
  State and input weight matrices

- **N**: 2-d array, optional:
  Cross weight matrix

**Returns**

- **K**: 2-d array:
  State feedback gains

- **S**: 2-d array:
  Solution to Riccati equation

- **E**: 1-d array:
  Eigenvalues of the closed loop system

**Examples**

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

## 5.8 State-space (SS) models

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{rss}([states, outputs, inputs])</td>
<td>Create a stable \textbf{continuous} random state space object.</td>
</tr>
<tr>
<td>\texttt{drss}([states, outputs, inputs])</td>
<td>Create a stable \textbf{discrete} random state space object.</td>
</tr>
<tr>
<td>\texttt{ctrb}(A, B)</td>
<td>Controllability matrix</td>
</tr>
<tr>
<td>\texttt{obsv}(A, C)</td>
<td>Observability matrix</td>
</tr>
<tr>
<td>\texttt{gram}(sys, type)</td>
<td>Gramian (controllability or observability)</td>
</tr>
</tbody>
</table>
5.8.1 control.matlab ctrb

control.matlab.\texttt{ctrb}(A, B)

Controllability matrix

\textbf{Parameters} \( A, B \): array_like or string :
  Dynamics and input matrix of the system

\textbf{Returns} \( C \): matrix :
  Controllability matrix

\textbf{Examples}

\begin{verbatim}
>>> C = ctrb(A, B)
\end{verbatim}

5.8.2 control.matlab obsv

control.matlab.\texttt{obsv}(A, C)

Observability matrix

\textbf{Parameters} \( A, C \): array_like or string :
  Dynamics and output matrix of the system

\textbf{Returns} \( O \): matrix :
  Observability matrix

\textbf{Examples}

\begin{verbatim}
>>> O = obsv(A, C)
\end{verbatim}

5.8.3 control.matlab gram

control.matlab.\texttt{gram}(sys, type)

Gramian (controllability or observability)

\textbf{Parameters} \( sys \): StateSpace :
  State-space system to compute Gramian for

\textbf{type}: String :
  Type of desired computation. \texttt{type} is either \texttt{c} (controllability) or \texttt{o} (observability).
  To compute the Cholesky factors of gramians use \texttt{cf} (controllability) or \texttt{of} (observability)

\textbf{Returns} \( \text{gram} \): array :
  Gramian of system

\textbf{Raises} \texttt{ValueError} :
  \begin{itemize}
    \item 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)

```
import Error:

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>minreal(sys[, tol, verbose])</code></td>
<td>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.</td>
</tr>
<tr>
<td><code>hsvd(sys)</code></td>
<td>Calculate the Hankel singular values.</td>
</tr>
<tr>
<td><code>balred(sys, orders[, method, alpha])</code></td>
<td>Balanced reduced order model of sys of a given order.</td>
</tr>
<tr>
<td><code>modred(sys, ELIM[, method])</code></td>
<td>Model reduction of sys by eliminating the states in ELIM using a given method.</td>
</tr>
<tr>
<td><code>era(YY, m, n, nin, nout, r)</code></td>
<td>Calculate an ERA model of order r based on the impulse-response data YY.</td>
</tr>
<tr>
<td><code>markov(Y, U, M)</code></td>
<td>Calculate the first M Markov parameters [D CB CAB ...] from input U, output Y.</td>
</tr>
</tbody>
</table>

#### 5.9.1 control.matlab.minreal

`control.matlab.minreal(sys, tol=0, verbose=True)`

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

Parameters sys: StateSpace or TransferFunction:

Original system
tol: real:
Tolerance
verbose: bool:
Print results if True

Returns rsys: StateSpace or TransferFunction:
Cleaned model
5.9.2 control.matlab.hsvd

control.matlab.hsvd(sys)
Calculate the Hankel singular values.

Parameters sys : StateSpace
A state space system

Returns H : Matrix
A list of Hankel singular values

See also:
gram

Notes

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

Examples

>>> H = hsvd(sys)

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.


Parameters sys : StateSpace :
Original system to reduce

orders: integer or array of integer :
Desired order of reduced order model (if a vector, returns a vector of systems)

method: string :
Method of removing states, either 'truncate' or 'matchdc'.

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

Returns `rsys: StateSpace`:
A reduced order model or a list of reduced order models if orders is a list

Raises `ValueError`:
- if `method` is not 'truncate' or 'matchdc'

`ImportError`:
if slycot routine ab09ad, ab09md, or ab09nd is not found

`ValueError`:
if there are more unstable modes than any value in orders

Examples

```python
>>> 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: StateSpace`:
A reduced order model

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

Examples

```python
>>> 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  
$nout \times nin$ dimensional impulse-response data  

$m$: integer  
Number of rows in Hankel matrix  

$n$: integer  
Number of columns in Hankel matrix  

$nin$: integer  
Number of input variables  

$nout$: integer  
Number of output variables  

$r$: integer  
Order of model  

Returns $sys$: StateSpace  
A reduced order model $sys=sp(Ar,Br,Dr)$  

Examples

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

5.9.6 control.matlab.markov

control.matlab.markov($Y, U, M$)  
Calculate the first $M$ Markov parameters $[D \ CB \ CAB …]$ from input $U$, output $Y$.  

Parameters $Y$: array_like  
Output data  

$U$: array_like  
Input data  

$M$: integer  
Number of Markov parameters to output  

Returns $H$: matrix  
First M Markov parameters  

Notes

Currently only works for SISO
Examples

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

### 5.10 Time delays

#### 5.10.1 control.matlab.pade

`control.matlab.pade(T[, n, numdeg])`

Create a linear system that approximates a delay.

Return the numerator and denominator coefficients of the Pade approximation.

**Parameters**
- `T`: number
  - time delay
- `n`: positive integer
  - degree of denominator of approximation
- `numdeg`: integer, or None (the default)
  - If None, numerator degree equals denominator degree
  - If >= 0, specifies degree of numerator
  - If < 0, numerator degree is n+numdeg

**Returns**
- `num`, `den`: array
  - Polynomial coefficients of the delay model, in descending powers of s.

**Notes**

Based on:

1. Algorithm 11.3.1 in Golub and van Loan, “Matrix Computation” 3rd Ed. pp. 572-574

### 5.11 Matrix equation solvers and linear algebra

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>lyap(A, Q[, C, E])</code></td>
<td>X = lyap(A,Q) solves the continuous-time Lyapunov equation</td>
</tr>
<tr>
<td><code>dlyap(A, Q[, C, E])</code></td>
<td>dlyap(A,Q) solves the discrete-time Lyapunov equation</td>
</tr>
<tr>
<td><code>care(A, B, Q[, R, S, E])</code></td>
<td>(X,L,G) = care(A,B,Q,R=None) solves the continuous-time algebraic Riccati</td>
</tr>
<tr>
<td><code>dare(A, B, Q, R[, S, E])</code></td>
<td>(X,L,G) = dare(A,B,Q,R) solves the discrete-time algebraic Riccati</td>
</tr>
</tbody>
</table>
5.11.1 control.matlab.lyap

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

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

\[ AX + XA^T + Q = 0 \]

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

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

\[ AX + XQ + C = 0 \]

where A and Q are square matrices.

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

\[ AXE^T + EXA^T + Q = 0 \]

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

5.11.2 control.matlab.dlyap

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

\[ AXA^T - X + Q = 0 \]

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

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

\[ AXQ^T - X + C = 0 \]

where A and Q are square matrices.

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

\[ AXA^T - EXE^T + Q = 0 \]

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

5.11.3 control.matlab.care

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

\[ A^TX + XA - XBR^{-1}B^TX + Q = 0 \]

where A and Q 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 \( G = B^T X \) 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

\[ A^TXE + E^TXA - (E^T XB + S)R^{-1}(B^TXE + S^T) = 0 \]

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. Matrix equation solvers and linear algebra
5.11.4 control.matlab.dare

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

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

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

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

the eigenvalues of \( A - B G \).

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

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

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

the eigenvalues of \( A - B G \), E.

5.12 Additional functions

---

### gangof4(P, C[, omega])

Plot the “Gang of 4” transfer functions for a system

Generates a 2x2 plot showing the “Gang of 4” sensitivity functions \([T, PS; CS, S]\)

**Parameters**

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

**Returns**

None

### unwrap(angle[, period])

Unwrap a phase angle to give a continuous curve

**Parameters**

- angle : array_like
  
  Array of angles to be unwrapped
- period : float, optional
  
  Period (defaults to 2*\(\pi\))

**Returns**

- angle_out : array_like
  
  Output array, with jumps of period/2 eliminated

---

88 Chapter 5. MATLAB compatibility module
Examples

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

5.13 Functions imported from other modules

- `linspace`
- `logspace`
- `ss2zpk`
- `tf2zpk`
- `zpk2ss`
- `zpk2tf`

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.FR method), 54
__init__() (control.StateSpace method), 52
__init__() (control.TransformerFunction method), 49

A

t (in module control), 33
t (in module control.matlab), 68

t (in module control), 14

t (in module control.matlab), 68

B

Balred() (in module control), 37
Balred() (in module control.matlab), 83
bode() (in module control.matlab), 74
bode_plot() (in module control), 17

c2d() (in module control.matlab), 62
canonical_form() (in module control), 44
care() (in module control), 30
care() (in module control.matlab), 87
connect() (in module control), 14
connect() (in module control.matlab), 67
control (module), 9
control.matlab (module), 57
ctrb() (in module control), 32
ctrb() (in module control.matlab), 81

D
damp() (in module control), 41
damp() (in module control.matlab), 70
dare() (in module control), 31
dare() (in module control.matlab), 88
db2mag() (in module control), 41
db2mag() (in module control.matlab), 62
dcgain() (control.FR method), 54
dcgain() (control.StateSpace method), 52
dcgain() (control.TransformerFunction method), 49
dcgain() (control.FRD method), 25
dcgain() (control.control.matlab), 68
dlyap() (module control), 31
dlyap() (module control.matlab), 87
drs() (module control), 13
drs() (module control.matlab), 61

E
era() (in module control), 39
era() (in module control.matlab), 84
evalfr() (control.FR method), 54
evalfr() (control.StateSpace method), 52
evalfr() (control.TransformerFunction method), 50
evalfr() (module control), 25
evalfr() (module control.matlab), 78

F
feedback() (control.FR method), 54
feedback() (control.StateSpace method), 52
feedback() (control.TransformerFunction method), 50
feedback() (in module control), 15
feedback() (in module control.matlab), 66
forced_response() (in module control), 20
FRD (class in control), 54
frd() (in module control), 12
frd() (in module control.matlab), 60
freqresp() (control.FR method), 54
freqresp() (control.StateSpace method), 52
freqresp() (control.TransformerFunction method), 50
freqresp() (module control), 26
freqresp() (module control.matlab), 77

G
gangof4() (in module control.matlab), 88
gangof4_plot() (in module control), 19
gram() (module control), 32
gram() (module control.matlab), 81

H
h2syn() (in module control), 34
Index

hinf() (in module control), 34
hinf() (control.StateSpace method), 52
hinf() (control.TransferFunction method), 50
hsvd() (in module control), 37
hsvd() (in module control.matlab), 83

I
impulse() (in module control.matlab), 72
impulse() (in module control.matlab), 21
initial() (in module control.matlab), 72
initial() (in module control.matlab), 22
iscasym() (control.FRD method), 54
iscasym() (control.StateSpace method), 52
iscasym() (control.TransferFunction method), 50
iscasym() (in module control), 42
isdtime() (control.FRD method), 54
isdtime() (control.StateSpace method), 52
isdtime() (control.TransferFunction method), 50
isdtime() (in module control), 42
issiso() (in module control), 42
issys() (in module control), 42

L
lqr() (in module control), 35
lqr() (in module control.matlab), 79
lsim() (in module control.matlab), 73
lyap() (in module control), 31
lyap() (in module control.matlab), 87

M
mag2db() (in module control), 41
mag2db() (in module control.matlab), 61
margin() (in module control), 27
margin() (in module control.matlab), 76
markov() (in module control), 39
markov() (in module control.matlab), 85
minreal() (control.StateSpace method), 52
minreal() (control.TransferFunction method), 50
minreal() (in module control), 36
minreal() (in module control.matlab), 82
modred() (in module control.matlab), 38
modred() (in module control.matlab), 84

N
negate() (in module control), 15
negate() (in module control.matlab), 67
nichols() (in module control.matlab), 76
nichols_plot() (in module control), 19
nyquist() (in module control.matlab), 75
nyquist_plot() (in module control), 18

O
observable_form() (in module control), 44
obsv() (in module control), 32
obsv() (in module control.matlab), 81

P
pade() (in module control), 42
pade() (in module control.matlab), 86
parallel() (in module control), 16
parallel() (in module control.matlab), 65
phase_crossover_frequencies() (in module control), 28
phase_plot() (in module control), 24
place() (in module control), 36
place() (in module control.matlab), 79
pole() (control.StateSpace method), 79
pole() (control.TransferFunction method), 50
pole() (in module control), 29
pole() (in module control.matlab), 69
pzmap() (in module control), 29
pzmap() (in module control.matlab), 70

R
reachable_form() (in module control), 44
returnScipySignalLTI() (control.StateSpace method), 53
returnScipySignalLTI() (control.TransferFunction method), 50
rlocus() (in module control.matlab), 78
root_locus() (in module control), 30
rss() (in module control), 12
rss() (in module control.matlab), 60

S
sample() (control.StateSpace method), 53
sample() (control.TransferFunction method), 50
sample_system() (in module control), 43
series() (in module control), 16
series() (in module control.matlab), 65
ss() (in module control), 9
ss() (in module control.matlab), 58
ss2tf() (in module control), 45
ss2tf() (in module control.matlab), 62
ssdata() (in module control), 46
stability_margins() (in module control), 27
StateSpace (class in control), 51
step() (in module control.matlab), 71
step_response() (in module control), 23

T
tf() (in module control), 10
tf() (in module control.matlab), 57
tf2ss() (in module control), 46
tf2ss() (in module control.matlab), 63
tfdata() (in module control), 47
tfdata() (in module control.matlab), 64
timebase() (in module control), 47

94
timebaseEqual() (in module control), 47
TransferFunction (class in control), 49

U
unwrap() (in module control), 40
unwrap() (in module control.matlab), 88
use_fbs_defaults() (in module control), 7
use_matlab_defaults() (in module control), 7

Z
zero() (control.StateSpace method), 53
zero() (control.TransferFunction method), 51
zero() (in module control), 29
zero() (in module control.matlab), 69