## Contents

1 Introduction
   1.1 Overview of the Toolbox ................................................. 3
   1.2 Some Differences from MATLAB ........................................ 3
   1.3 Getting Started .......................................................... 3

2 Python-Control Functions
   2.1 Creating System Models ............................................... 5
   2.2 Block Diagram Algebra ................................................. 12
   2.3 Control System Analysis .............................................. 14
   2.4 Frequency Domain Plotting .......................................... 21
   2.5 Time Domain Simulation ............................................. 24
   2.6 Control System Synthesis ........................................... 29
   2.7 Model Simplification Tools ......................................... 30
   2.8 Utility Functions ..................................................... 32

3 Python-Control Classes
   3.1 LTI System Class ..................................................... 33
   3.2 State Space Class .................................................... 34
   3.3 Transfer Function Class ............................................ 36
   3.4 FRD Class ............................................................. 38

4 MATLAB Compatibility Module
   4.1 Creating linear models .............................................. 41
   4.2 Data extraction ....................................................... 42
   4.3 Conversions ............................................................ 42
   4.4 System interconnections .......................................... 42
   4.5 System gain and dynamics ....................................... 43
   4.6 Time-domain analysis ............................................ 43
   4.7 Frequency-domain analysis ..................................... 43
   4.8 Model simplification .............................................. 43
   4.9 Compensator design ................................................. 44
   4.10 LQR/LQG design ................................................... 44
   4.11 State-space (SS) models ......................................... 44
   4.12 Frequency response data (FRD) models ......................... 44
   4.13 Time delays .......................................................... 45
   4.14 Model dimensions and characteristics ......................... 45
   4.15 Overloaded arithmetic operations ............................... 45
   4.16 Matrix equation solvers and linear algebra .................. 45
Welcome to the Python Control Systems Library (python-control) User’s Manual. This manual describes the python-control package, including all of the functions defined in the package and examples showing how to use the package.

Contents:
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.

In addition to the documentation here, there is a project wiki that contains some additional information about how to use the package (including some detailed worked examples):

    http://python-control.sourceforge.net

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:

    http://www.scipy.org/NumPy_for_Matlab_Users

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
- Transfer functions are currently only implemented for SISO systems; use state space representations for MIMO systems.

1.3 Getting Started

2. Untar the source code in a temporary directory and run ‘python setup.py install’ to build and install the code.

3. To see if things are working correctly, run ipython -pylab and run the script ‘examples/secord-matlab.py’. This should generate a step response, Bode plot and Nyquist plot for a simple second order system. (For more detailed tests, run nosetests in the main directory.)

4. To see the commands that are available, run the following commands in ipython:

```plaintext
>>> import control
>>> ?control
```

5. If you want to have a MATLAB-like environment for running the control toolbox, use:

```plaintext
>>> from control.matlab import *
>>> ?control.matlab
```
Python-Control Functions

2.1 Creating System Models

Python-control provides a number of methods for creating LTI control systems.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ss()</td>
<td>create state-space (SS) models</td>
</tr>
<tr>
<td>tf()</td>
<td>create transfer function (TF) models</td>
</tr>
</tbody>
</table>

2.1.1 System creation

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

The StateSpace class represents state space instances and functions.

The StateSpace class is used throughout the python-control library to represent systems in state space form. This class is derived from the Lti base class.

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’ class 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.

**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`: `Lti (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`: `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)
```

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

The `TransferFunction` class represents TF instances and functions.

- The `TransferFunction` class is derived from the `Lti` parent class. It is used through the python-control library 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` class 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.
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: 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

Todo
The next paragraph contradicts the comment in the example! Also “input” should come before “output” in the
sentence:

“from the (j+1)st output to the (i+1)st input”

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

The coefficients [2, 3, 4] denote the polynomial $2 \cdot s^2 + 3 \cdot s + 4$. 

2.1. Creating System Models
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)
```

2.1.2 Utility functions and conversions

control.drss(states=1, outputs=1, inputs=1)
Create a stable discrete random state space object.

Parameters
- states: integer
  Number of state variables
- inputs: integer
  Number of system inputs
- outputs: integer
  Number of system outputs

Returns
- sys: StateSpace
  The randomly created linear system

Raises
- ValueError
  if any input is not a positive integer

See also:
- rss

Notes

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

control.istctime(sys, strict=False)
Check to see if a system is a continuous-time system

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

control.isdtime(sys, strict=False)
Check to see if a system is a discrete time system

Parameters
- sys: LTI system
System to be checked

strict: bool (default = False):

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

control.issys(object)

control.pade(T, n=1)

Create a linear system that approximates a delay.

Return the numerator and denominator coefficients of the Pade approximation.

Parameters  

- T : number
time delay
- n : integer
order of approximation

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

Notes


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’ (default), ‘tustin’, ‘zoh’

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

Notes

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

Examples

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

control.ssdata(sys)
Return state space data objects for a system

**Parameters**

- `sys`: `Lti` (StateSpace, or TransferFunction)
  LTI system whose data will be returned
Returns  

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

State space data for the system

control.tf2ss (*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)
```

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)

```python
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.

```python
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.

### 2.2 Block Diagram Algebra

```python
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.

```python
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.
```

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.
```

control.series(sys1, sys2)
Return the series connection sys2 * sys1 for → sys1 → sys2 →.

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

Returns out: scalar, StateSpace, or TransferFunction:

Raises ValueError :
     if sys2.inputs does not equal sys1.outputs if sys1.dt is not compatible with sys2.dt
See also:
parallel, feedback

Notes

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

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

Examples

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

2.3 Control System Analysis

```python
control.acker(A, B, poles)
Pole placement using Ackermann method
Call: K = acker(A, B, poles)
Parameters A, B : 2-d arrays
State and input matrix of the system
poles: 1-d list : Desired eigenvalue locations
Returns K: matrix :
Gains such that A - B K has given eigenvalues
```

```python
control ctrb(A, B)
Controllabilty matrix
Parameters A, B: array_like or string :
Dynamics and input matrix of the system
Returns C: matrix :
Controllability matrix
```

Examples

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

```python
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

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

14 Chapter 2. Python-Control Functions
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 \( A - B \, G \).

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

\[
(X,L,G) = \text{dare}(A,B,Q,R,S=None, E=None)
\]
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 must be symmetric. 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 \( A - B \, G \).

\[
dlyap(A,Q)\]
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

\[
AXQT - X + C = 0
\]

where A and Q are square matrices.

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

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

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

\[
dcgain(*args)
\]
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: matrix:
The gain of each output versus each input: $y = gain \cdot u$

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

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

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

Todo
Add example with MIMO system

control.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).

Returns gram: array:
Gramian of system
Raises `ValueError`:

- if system is not instance of `StateSpace` class
- if `type` is not ‘c’ or ‘o’
- if system is unstable (sys.A has eigenvalues not in left half plane)

`ImportError`:

if slycot routin sb03md cannot be found

Examples

```python
>>> Wc = gram(sys,'c')
>>> Wo = gram(sys,'o')
```

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

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

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

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

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

\[ AX + XQ + C = 0 \]

where A and Q are square matrices.

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

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

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

control.freqresp(sys, omega)

Frequency response of an LTI system at multiple angular frequencies.

Parameters

- `sys`: `StateSpace` or `TransferFunction`
  - Linear system

- `omega`: `array_like`
  - List of frequencies

Returns

- `mag`: `ndarray`
  - `phase`: `ndarray`
  - `omega`: list, tuple, or `ndarray`

See also:

evalfr, bode

Notes

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

2.3. Control System Analysis
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, 1i, 10i.
```

control.margin(*args)

Calculate gain and phase margins and associated crossover frequencies

Function `margin` takes either 1 or 3 parameters.

**Parameters**

- `sys` : StateSpace or TransferFunction
  - Linear SISO system
- `mag`, `phase`, `w` : array_like
  - Input magnitude, phase (in deg.), and frequencies (rad/sec) from bode frequency response data

**Returns**

- `gm`, `pm`, `Wcg`, `Wcp` : float
  - Gain margin gm, phase margin pm (in deg), gain crossover frequency (corresponding to phase margin) and phase crossover frequency (corresponding to gain margin), in rad/sec of SISO open-loop. If more than one crossover frequency is detected, returns the lowest corresponding margin.

Examples

```python
>>> sys = ss("1. -2; 3. -4", "5.; 7", "6. 8", "9."
>>> gm, pm, wg, wp = margin(sys)
margin: no magnitude crossings found
```

Todo

better example system!

```python
#>>> gm, pm, wg, wp = margin(mag, phase, w)
```

control.markov(Y, U, M)

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

**Parameters**

- `Y` : array_like : Output data
U: array_like:
   Input data
M: integer:
   Number of Markov parameters to output
Returns H: matrix:
   First M Markov parameters

Notes

Currently only works for SISO

Examples

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

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)
```

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]))
```

control.pole(sys)
Compute system poles.

Parameters sys: StateSpace or TransferFunction:
Linear system

2.3. Control System Analysis
Returns poles: ndarray:
    Array that contains the system’s poles.

Raises NotImplementedError:
    when called on a TransferFunction object

See also:
    zero

Notes

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

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

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

Parameters sys: LTI object
    Linear input/output systems (SISO only, for now)

    kvect: list or ndarray, optional
        List of gains to use in computing diagram

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

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

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

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

Returns rlist: ndarray
    Computed root locations, given as a 2d array

    klist: ndarray or list
        Gains used. Same as klist keyword argument if provided.

control.stabilityMargins(sysdata, deg=True, returnall=False, epsw=1e-10)

Calculate gain, phase and stability margins and associated crossover frequencies.

Parameters sysdata: linsys or (mag, phase, omega) sequence :
    sys [linsys] Linear SISO system

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

    deg=True: boolean :
        If true, all input and output phases in degrees, else in radians
returnall=False: boolean :
   If true, return all margins found. Note that for frequency data or FRD systems, only one
   margin is found and returned.
epsw=1e-10: float :
   frequencies below this value are considered static gain, and not returned as margin.

Returns gm, pm, sm, wg, wp, ws: float or array_like :
   Gain margin gm, phase margin pm, stability margin sm, and associated crossover fre-
   quencies wg, wp, and ws of SISO open-loop. If more than one crossover frequency is
detected, returns the lowest corresponding margin. When requesting all margins, the
return values are array_like, and all margins are returns for linear systems not equal to
FRD

control.zer0(sys)
   Compute system zeros.

Parameters sys: StateSpace or TransferFunction :
   Linear system

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

Raises Not Implemented Error :
   when called on a TransferFunction object or a MIMO StateSpace object

See also:

pole

Notes

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

2.4 Frequency Domain Plotting

2.4.1 Plotting routines

control.bode_plot(syslist, omega=None, dB=None, Hz=None, deg=None, Plot=True, *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 (list or bounds) 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, return phase in degrees (else radians)

Plot : boolean
    If True, plot magnitude and phase

*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

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

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)
```

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)
```

**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**

**control.nichols_plot** (**syslist**, **omega=None**, **grid=True**)  
Nichols plot for a system

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

**Parameters**

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

**Returns**

**2.4.2 Utility functions**

**control.freqplot.default_frequency_range** (**syslist**)  
Compute a reasonable default frequency range for frequency domain plots.

Finds a reasonable default frequency range by examining the features (poles and zeros) of the systems in syslist.

**Parameters**

- **syslist**: list of Lti  
  List of linear input/output systems (single system is OK)

**Returns**

**omega**: array
Range of frequencies in rad/sec

Examples

```python
>>> from matlab import ss
>>> sys = ss("1. -2; 3. -4", "5.; 7", "6. 8", "9.")
>>> omega = default_frequency_range(sys)
```

2.5 Time Domain Simulation

Time domain simulation.

This file contains a collection of functions that calculate time responses for linear systems.

2.5.1 Convention for Time Series

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):

\[ T = [[t_1, t_2, t_3, \ldots, t_n]] \]

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:

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

Same for X, Y

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

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

\[
X_0 = [\begin{bmatrix}
    x_1 \\
    x_2 \\
    \vdots
\end{bmatrix}]
\]
As all simulation functions return arrays, plotting is convenient:

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

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

```python
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
```

### 2.5.2 Time responses

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

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: Convention for Time Series

**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)

- **keywords**: 
  Additional keyword arguments control the solution algorithm for the differential equations. These arguments are passed on to the function `scipy.integrate.odeint()`. See the documentation for `scipy.integrate.odeint()` for information about these arguments.
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)
```

callable

*initial_response*(sys, *T=None, X0=0.0, input=0, output=None, transpose=False, **keywords*)

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: *Convention for Time Series*

**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`)

- ****keywords: **
  - **Additional keyword arguments control the solution algorithm for the differential equations. These arguments are passed on to the function `lsim()`, which in turn passes them on to `scipy.integrate.odeint()`. See the documentation for `scipy.integrate.odeint()` for information about these arguments.**

**Returns**

- **T**: array
Time values of the output

\textbf{yout}: array :

Response of the system

\textbf{See also:}

\texttt{forced_response, impulse_response, step_response}

\textbf{Examples}

\begin{verbatim}
>>> T, yout = initial_response(sys, T, X0)
\end{verbatim}

\texttt{control.step_response(sys, T=None, X0=0.0, input=None, output=None, transpose=False, **keywords)}

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 \texttt{input} and \texttt{output} do this. All other inputs are set to 0, all other outputs are ignored.

For information on the shape of parameters \texttt{T, X0} and return values \texttt{T, yout} see: \textit{Convention for Time Series}

\textbf{Parameters}

\texttt{sys}: StateSpace, or \texttt{TransferFunction} :

LTI system to simulate

\texttt{T}: array-like object, \texttt{optional} :

Time vector (argument is autocomputed if not given)

\texttt{X0}: array-like or number, \texttt{optional} :

Initial condition (default = 0)

Numbers are converted to constant arrays with the correct shape.

\texttt{input}: int :

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

\texttt{output}: int :

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

\texttt{transpose}: bool :

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

\textbf{**keywords:} :

Additional keyword arguments control the solution algorithm for the differential equations. These arguments are passed on to the function \texttt{lsim()}, which in turn passes them on to \texttt{scipy.integrate.odeint()}. See the documentation for \texttt{scipy.integrate.odeint()} for information about these arguments.

\textbf{Returns}

\texttt{T}: array :

Time values of the output

\texttt{yout}: array :

Response of the system
See also:
forced_response, initial_response, impulse_response

Examples

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

2.5.3 Phase portraits

```python
control.phaseplot.phase_plot(odefun, X=None, Y=None, scale=1, X0=None, T=None, lin-
grid=None, lintime=None, logtime=None, timepts=None, parms=(), verbose=True)
```

Phase plot for 2D dynamical systems

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

**Call signatures:**
- `phase_plot(func, X, Y, ...) - display vector field on meshgrid`
- `phase_plot(func, X, Y, scale, ...) - scale arrows`
- `phase_plot(func, X0=(...), T=Tmax, ...) - display stream lines`
- `phase_plot(func, 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 dx/dt of dimension 2.

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

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

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

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

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

- **lintime = N: integer, optional**:
  - Draw N arrows using equally space time points
logtime = (N, lambda): (integer, float), optional:
    Draw N arrows using exponential time constant lambda

timepts = [t1, t2, ...]: array-like, optional:
    Draw arrows at the given list times

draw = tuple, optional:
    List of parameters to pass to vector field: func(x, t, *draw)

See also:

box_grid, Y

control.phaseplot.box_grid(xlimp, ylimp)

box_grid generate list of points on edge of box
list = box_grid([xmin xmax xnum], [ymin ymax ynum]) generates a list of points that correspond to a uniform
grid at the end of the box defined by the corners [xmin ymin] and [xmax ymax].

2.6 Control System Synthesis

control.lqr(*args, **keywords)

Linear quadratic regulator design

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

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

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)

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])
```

control.place(A, B, p)
Place closed loop eigenvalues

**Parameters**

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

**Returns**

- K: 2-d array
  Gains such that A - B K has given eigenvalues

Examples

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

### 2.7 Model Simplification Tools

control.balred(sys, orders, method='truncate')
Balanced reduced order model of sys of a given order. States are eliminated based on Hankel singular value.

**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'.

**Returns**

- rsys: StateSpace:
  A reduced order model

**Raises**

- ValueError:
  - if method is not 'truncate'
  - if eigenvalues of sys.A are not all in left half plane (sys must be stable)
ImportError:
if slycot routine ab09ad is not found

Examples

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

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)
```

countrol.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')
```

## 2.8 Utility Functions

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

Unwrap a phase angle to give a continuous curve

**Parameters**

- **X**: array_like
  
  Input array

- **period**: number
  
  Input period (usually either 2**pi or 360)

**Returns**

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

**Examples**

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

lti.py

The lti module contains the Lti parent class to the child classes StateSpace and TransferFunction. It is designed for use in the python-control library.

Routines in this module:
Lti.__init__ isdtime() isctime() timebase() timebaseEqual()

class control.lti.Lti (inputs=1, outputs=1, dt=None)

Lti is a parent class to linear time invariant control (LTI) objects.

Lti is the parent to the StateSpace and TransferFunction child classes. It contains the number of inputs and outputs, and the timebase (dt) for the system.

The timebase for the system, dt, is used to specify whether the system is operating in continuous or discrete time. It can have the following values:

• dt = None No timebase specified
• dt = 0 Continuous time system
• dt > 0 Discrete time system with sampling time dt
• dt = True Discrete time system with unspecified sampling time

When two Lti systems are combined, their timebases must match. A system with timebase None can be combined with a system having a specified timebase, and the result will have the timebase of the latter system.

The StateSpace and TransferFunction child classes contain several common “virtual” functions. These are:
__init__ copy __str__ __neg__ __add__ __radd__ __sub__ __rsub__ __mul__ __rmul__ __div__ __rdiv__

evalfr freqresp pole zero feedback returnScipySignalLti

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

```python
control.lti.isdtime(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

```python
control.lti.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

```python
control.lti.isdtime(sys, strict=False)
```

**3.2 State Space Class**

`statesp.py`

State space representation and functions.

This file contains the StateSpace class, which is used to represent linear systems in state space. This is the primary representation for the python-control library.

Routines in this module:

```
```
class `control.statesp.StateSpace`(*args)
The StateSpace class represents state space instances and functions.

The StateSpace class is used throughout the python-control library to represent systems in state space form. This class is derived from the Lti base class.

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’ class 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
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.

```python
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.

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

```python
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.

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

Returns a matrix of values evaluated at complex variable s.

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

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

```python
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.

```python
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.

```python
Parameters
Ts : float
    Sampling period
```

3.2. State Space Class
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.

### 3.3 Transfer Function Class

xferfcn.py

Transfer function representation and functions.

This file contains the TransferFunction class and also functions that operate on transfer functions. This is the primary representation for the python-control library.

Routines in this module:

TransferFunction.__init__ TransferFunction._truncatecoeff TransferFunction.copy TransferFunction.__str__ TransferFunction._repr__ TransferFunction._neg__ TransferFunction._add__ TransferFunction._radd__ TransferFunction._sub__ TransferFunction._rsub__ TransferFunction._mul__ TransferFunction._rmul__ TransferFunction._div__ TransferFunction._rdiv__ TransferFunction._truediv__ TransferFunction._rtruediv__ TransferFunction.evalfr TransferFunction.freqresp TransferFunction.pole TransferFunction.zero TransferFunction.feedback TransferFunction.minreal TransferFunction.returnScipySignalLti TransferFunction._common_den _tfpolyToString _addSISO _convertToTransferFunction

class control.xferfcn.TransferFunction(*args)

The TransferFunction class represents TF instances and functions.

The TransferFunction class is derived from the Lti parent class. It is used through the python-control library 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’ class 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.

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

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

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

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

mag, phase, omega = self.freqresp(omega)
reports the value of the magnitude, phase, and angular frequency of the transfer function matrix evaluated at $s = i \omega$, where omega is a list of angular frequencies, and is a sorted version of the input omega.

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

Returns a matrix of values evaluated at complex variable s.

**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,

```python
>>> 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)

\textbf{alpha} : float within \([0, 1]\)

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

\textbf{Returns} \texttt{sysd} : StateSpace system

Discrete time system, with sampling rate Ts

\textbf{Notes}

1. Available only for SISO systems
2. Uses the command \texttt{cont2discrete} from \texttt{scipy.signal}

\textbf{Examples}

\begin{verbatim}
>>> sys = TransferFunction(1, [1,1])
>>> sysd = sys.sample(0.5, method='bilinear')
\end{verbatim}

\textbf{zero} ()

Compute the zeros of a transfer function.

\subsection*{3.4 FRD Class}

\textbf{class} \texttt{control.frdata.FRD(*args, **kwargs)}

The FRD class represents (measured?) frequency response TF instances and functions.

The FRD class is derived from the Lti parent class. It is used throughout the python-control library to represent systems in frequency response data form.

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

\begin{verbatim}
>>> frdata[2,5,:] = numpy.array([1., 0.8-0.2j, 0.2-0.8j])
\end{verbatim}

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.

\textbf{evalfr} (\texttt{omega})

Evaluate a transfer function at a single angular frequency.

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

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

\textbf{feedback} (\texttt{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 \( \omega \).
MATLAB Compatibility Module

This file 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). Most of the functions are just calls to python-control functions defined elsewhere. Use `from control.matlab import *` in python to include all of the functions defined here. Functions that are defined in other libraries that have the same names as their MATLAB equivalents are automatically imported here.

The following tables give an overview of the module `control.matlab`. They also show the implementation progress and the planned features of the module.

The symbols in the first column show the current state of a feature:

- `*`: The feature is currently implemented.
- `-`: The feature is not planned for implementation.
- `s`: A similar feature from another library (Scipy) is imported into the module, until the feature is implemented here.

### 4.1 Creating linear models

<table>
<thead>
<tr>
<th>*</th>
<th><code>tf()</code></th>
<th>create transfer function (TF) models</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><code>zpk</code></td>
<td>create zero/pole/gain (ZPK) models.</td>
</tr>
<tr>
<td>*</td>
<td><code>ss()</code></td>
<td>create state-space (SS) models</td>
</tr>
<tr>
<td></td>
<td><code>dss</code></td>
<td>create descriptor state-space models</td>
</tr>
<tr>
<td></td>
<td><code>delayss</code></td>
<td>create state-space models with delayed terms</td>
</tr>
<tr>
<td>*</td>
<td><code>frd()</code></td>
<td>create frequency response data (FRD) models</td>
</tr>
<tr>
<td></td>
<td><code>lti/exp</code></td>
<td>create pure continuous-time delays (TF and ZPK only)</td>
</tr>
<tr>
<td></td>
<td><code>filt</code></td>
<td>specify digital filters</td>
</tr>
<tr>
<td>-</td>
<td><code>lti/set</code></td>
<td>set/modify properties of LTI models</td>
</tr>
<tr>
<td>-</td>
<td><code>setdelaymodel</code></td>
<td>specify internal delay model (state space only)</td>
</tr>
<tr>
<td>*</td>
<td><code>rss()</code></td>
<td>create a random continuous state space model</td>
</tr>
<tr>
<td>*</td>
<td><code>drss()</code></td>
<td>create a random discrete state space model</td>
</tr>
</tbody>
</table>
4.2 Data extraction

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tfdata()</td>
<td>extract numerators and denominators</td>
</tr>
<tr>
<td>lti/zpkdata</td>
<td>extract zero/pole/gain data</td>
</tr>
<tr>
<td>lti/ssdata</td>
<td>extract state-space matrices</td>
</tr>
<tr>
<td>lti/dssdata</td>
<td>descriptor version of SSDATA</td>
</tr>
<tr>
<td>frd/frdata</td>
<td>extract frequency response data</td>
</tr>
<tr>
<td>lti/get</td>
<td>access values of LTI model properties</td>
</tr>
<tr>
<td>ss/getDelayModel</td>
<td>access internal delay model (state space)</td>
</tr>
</tbody>
</table>

4.3 Conversions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tf()</td>
<td>conversion to transfer function</td>
</tr>
<tr>
<td>zpk</td>
<td>conversion to zero/pole/gain</td>
</tr>
<tr>
<td>ss()</td>
<td>conversion to state space</td>
</tr>
<tr>
<td>frd()</td>
<td>conversion to frequency data</td>
</tr>
<tr>
<td>c2d()</td>
<td>continuous to discrete conversion</td>
</tr>
<tr>
<td>d2c</td>
<td>discrete to continuous conversion</td>
</tr>
<tr>
<td>d2d</td>
<td>resample discrete-time model</td>
</tr>
<tr>
<td>upsample</td>
<td>upsample discrete-time LTI systems</td>
</tr>
<tr>
<td>ss2tf()</td>
<td>state space to transfer function</td>
</tr>
<tr>
<td>ss2zpk</td>
<td>transfer function to zero-pole-gain</td>
</tr>
<tr>
<td>tf2ss()</td>
<td>transfer function to state space</td>
</tr>
<tr>
<td>tf2zpk</td>
<td>transfer function to zero-pole-gain</td>
</tr>
<tr>
<td>zpk2ss</td>
<td>zero-pole-gain to state space</td>
</tr>
<tr>
<td>zpk2tf</td>
<td>zero-pole-gain to transfer function</td>
</tr>
</tbody>
</table>

4.4 System interconnections

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>append()</td>
<td>group LTI models by appending inputs/outputs</td>
</tr>
<tr>
<td>parallel()</td>
<td>connect LTI models in parallel (see also overloaded +)</td>
</tr>
<tr>
<td>series()</td>
<td>connect LTI models in series (see also overloaded *)</td>
</tr>
<tr>
<td>feedback()</td>
<td>connect lti models with a feedback loop</td>
</tr>
<tr>
<td>lti/lft</td>
<td>generalized feedback interconnection</td>
</tr>
<tr>
<td>:func:<code>~bdalg.connect</code></td>
<td>arbitrary interconnection of lti models</td>
</tr>
<tr>
<td>sumblk</td>
<td>summing junction (for use with connect)</td>
</tr>
<tr>
<td>strseq</td>
<td>builds sequence of indexed strings (for I/O naming)</td>
</tr>
</tbody>
</table>
4.5 System gain and dynamics

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dcgain()</td>
<td>steady-state (D.C.) gain</td>
</tr>
<tr>
<td>lti/bandwidth</td>
<td>system bandwidth</td>
</tr>
<tr>
<td>lti/norm</td>
<td>h2 and Hinfinity norms of LTI models</td>
</tr>
<tr>
<td>pole()</td>
<td>system poles</td>
</tr>
<tr>
<td>zero()</td>
<td>system (transmission) zeros</td>
</tr>
<tr>
<td>lti/order</td>
<td>model order (number of states)</td>
</tr>
<tr>
<td>pzmap()</td>
<td>pole-zero map (TF only)</td>
</tr>
<tr>
<td>lti/iopzmap</td>
<td>input/output pole-zero map</td>
</tr>
<tr>
<td>damp()</td>
<td>natural frequency, damping of system poles</td>
</tr>
<tr>
<td>esort</td>
<td>sort continuous poles by real part</td>
</tr>
<tr>
<td>dsort</td>
<td>sort discrete poles by magnitude</td>
</tr>
<tr>
<td>lti/stabsep</td>
<td>stable/unstable decomposition</td>
</tr>
<tr>
<td>lti/modsep</td>
<td>region-based modal decomposition</td>
</tr>
</tbody>
</table>

4.6 Time-domain analysis

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>step()</td>
<td>step response</td>
</tr>
<tr>
<td>stepinfo</td>
<td>step response characteristics</td>
</tr>
<tr>
<td>impulse()</td>
<td>impulse response</td>
</tr>
<tr>
<td>initial()</td>
<td>free response with initial conditions</td>
</tr>
<tr>
<td>lsim()</td>
<td>response to user-defined input signal</td>
</tr>
<tr>
<td>lsiminfo</td>
<td>linear response characteristics</td>
</tr>
<tr>
<td>gensig</td>
<td>generate input signal for LSIM</td>
</tr>
<tr>
<td>covar</td>
<td>covariance of response to white noise</td>
</tr>
</tbody>
</table>

4.7 Frequency-domain analysis

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bode()</td>
<td>Bode plot of the frequency response</td>
</tr>
<tr>
<td>lti/bodemag</td>
<td>Bode magnitude diagram only</td>
</tr>
<tr>
<td>sigma</td>
<td>singular value frequency plot</td>
</tr>
<tr>
<td>nyquist()</td>
<td>Nyquist plot</td>
</tr>
<tr>
<td>nichols()</td>
<td>Nichols plot</td>
</tr>
<tr>
<td>margin()</td>
<td>gain and phase margins</td>
</tr>
<tr>
<td>lti/allmargin</td>
<td>all crossover frequencies and margins</td>
</tr>
<tr>
<td>freqresp()</td>
<td>frequency response over a frequency grid</td>
</tr>
<tr>
<td>evalfr()</td>
<td>frequency response at single frequency</td>
</tr>
</tbody>
</table>

4.8 Model simplification

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>minreal()</td>
<td>minimal realization; pole/zero cancellation</td>
</tr>
<tr>
<td>ss/sminreal</td>
<td>structurally minimal realization</td>
</tr>
<tr>
<td>hsvd()</td>
<td>hankel singular values (state contributions)</td>
</tr>
<tr>
<td>balred()</td>
<td>reduced-order approximations of LTI models</td>
</tr>
<tr>
<td>modred()</td>
<td>model order reduction</td>
</tr>
</tbody>
</table>
4.9 Compensator design

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rlocus()</td>
<td>Evans root locus</td>
</tr>
<tr>
<td>place()</td>
<td>Pole placement</td>
</tr>
<tr>
<td>estim</td>
<td>Form estimator given estimator gain</td>
</tr>
<tr>
<td>reg</td>
<td>Form regulator given state-feedback and estimator gains</td>
</tr>
</tbody>
</table>

4.10 LQR/LQG design

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>lqr()</td>
<td>Linear quadratic (LQ) state-fbk regulator</td>
</tr>
<tr>
<td>dlqr</td>
<td>Discrete-time LQ state-feedback regulator</td>
</tr>
<tr>
<td>lqrd</td>
<td>Discrete LQ regulator for continuous plant</td>
</tr>
<tr>
<td>ss/lqi</td>
<td>Linear-Quadratic-Integral (LQI) controller</td>
</tr>
<tr>
<td>ss/kalman</td>
<td>Kalman state estimator</td>
</tr>
<tr>
<td>ss/kalmd</td>
<td>Discrete Kalman estimator for cts plant</td>
</tr>
<tr>
<td>ss/lqgreg</td>
<td>Build LQG regulator from LQ gain and Kalman estimator</td>
</tr>
<tr>
<td>ss/lqgtrack</td>
<td>Build LQG servo-controller</td>
</tr>
<tr>
<td>augstate</td>
<td>Augment output by appending states</td>
</tr>
</tbody>
</table>

4.11 State-space (SS) models

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rss()</td>
<td>Random stable cts-time state-space models</td>
</tr>
<tr>
<td>drrs()</td>
<td>Random stable disc-time state-space models</td>
</tr>
<tr>
<td>ss2ss</td>
<td>State coordinate transformation</td>
</tr>
<tr>
<td>canon</td>
<td>Canonical forms of state-space models</td>
</tr>
<tr>
<td>ctrb()</td>
<td>Controllability matrix</td>
</tr>
<tr>
<td>obsv()</td>
<td>Observability matrix</td>
</tr>
<tr>
<td>gram()</td>
<td>Controllability and observability gramians</td>
</tr>
<tr>
<td>ss/prescale</td>
<td>Optimal scaling of state-space models</td>
</tr>
<tr>
<td>balreal</td>
<td>Gramian-based input/output balancing</td>
</tr>
<tr>
<td>ss/xperm</td>
<td>Reorder states</td>
</tr>
</tbody>
</table>

4.12 Frequency response data (FRD) models

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>frd/chgunits</td>
<td>Change frequency vector units</td>
</tr>
<tr>
<td>frd/fcat</td>
<td>Merge frequency responses</td>
</tr>
<tr>
<td>frd/fselect</td>
<td>Select frequency range or subgrid</td>
</tr>
<tr>
<td>frd/fnorm</td>
<td>Peak gain as a function of frequency</td>
</tr>
<tr>
<td>frd/abs</td>
<td>Entrywise magnitude of frequency response</td>
</tr>
<tr>
<td>frd/real</td>
<td>Real part of the frequency response</td>
</tr>
<tr>
<td>frd/imag</td>
<td>Imaginary part of the frequency response</td>
</tr>
<tr>
<td>frd/interp</td>
<td>Interpolate frequency response data</td>
</tr>
<tr>
<td>mag2db</td>
<td>Convert magnitude to decibels (dB)</td>
</tr>
<tr>
<td>db2mag</td>
<td>Convert decibels (dB) to magnitude</td>
</tr>
</tbody>
</table>
4.13 Time delays

| lti/hasdelay | true for models with time delays |
| lti/totaldelay | total delay between each input/output pair |
| lti/delay2z | replace delays by poles at \( z=0 \) or FRD phase shift |
| * pade() | pade approximation of time delays |

4.14 Model dimensions and characteristics

| class | model type (‘tf’, ‘zpk’, ‘ss’, or ‘frd’) |
| isa | test if model is of given type |
| tf/size | model sizes |
| lti/ndims | number of dimensions |
| lti/isempty | true for empty models |
| lti/isct | true for continuous-time models |
| lti/isdt | true for discrete-time models |
| lti/isproper | true for proper models |
| lti/issiso | true for single-input/single-output models |
| lti/isstable | true for models with stable dynamics |
| lti/reshape | reshape array of linear models |

4.15 Overloaded arithmetic operations

| * + and - | add, subtract systems (parallel connection) |
| * * | multiply systems (series connection) |
| / | right divide – sys1*inv(sys2) |
| - \ | left divide – inv(sys1)*sys2 |
| ^ | powers of a given system |
| ‘ | pertransposition |
| .’ | transposition of input/output map |
| .* | element-by-element multiplication |
| [...] | concatenate models along inputs or outputs |
| lti/stack | stack models/arrays along some dimension |
| lti/inv | inverse of an LTI system |
| lti/conj | complex conjugation of model coefficients |

4.16 Matrix equation solvers and linear algebra

| * lyap() | solve continuous-time Lyapunov equations |
| * dlyap() | solve discrete-time Lyapunov equations |
| lyapchol, dlyapchol | square-root Lyapunov solvers |
| * care() | solve continuous-time algebraic Riccati equations |
| * dare() | solve disc-time algebraic Riccati equations |
| gcare, gdare | generalized Riccati solvers |
| bdschur | block diagonalization of a square matrix |
4.17 Additional functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>gangof4()</code></td>
<td>generate the Gang of 4 sensitivity plots</td>
</tr>
<tr>
<td><code>linspace()</code></td>
<td>generate a set of numbers that are linearly spaced</td>
</tr>
<tr>
<td><code>logspace()</code></td>
<td>generate a set of numbers that are logarithmically spaced</td>
</tr>
<tr>
<td><code>unwrap()</code></td>
<td>unwrap phase angle to give continuous curve</td>
</tr>
</tbody>
</table>

control.matlab.\texttt{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

```python
>>> bode(sys, w)
>>> bode(sys1, sys2, ..., sysN)
>>> bode(sys1, sys2, ..., sysN, w)
>>> bode(sys1, 'plotstyle1', ..., sysN, 'plotstyleN')
```

control.matlab.\texttt{c2d} (sysc, Ts, method='zoh')

Return a discrete-time system
**Parameters**  
*sysc*: Lti (StateSpace or TransferFunction), *continuous*:  
System to be converted

*Ts*: number  
Sample time for the conversion

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

`control.matlab.damp(sys, doprint=True)`  
Compute natural frequency, damping and poles of a system

The function takes 1 or 2 parameters

**Parameters**  
*sys*: Lti (StateSpace or TransferFunction):  
A linear system object

*doprint*:  
if true, print table with values

**Returns**  
*wn*: array  
Natural frequencies of the poles

*damping*: array  
Damping values

*poles*: array  
Pole locations

See also:  
`pole`

`control.matlab.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*: matrix  
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:

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

code:control.matlab.drss(states=1, outputs=1, inputs=1)

Create a stable discrete random state space object.

- **Parameters**
  - **states**: integer
    - Number of state variables
  - **inputs**: integer
    - Number of system inputs
  - **outputs**: integer
    - Number of system outputs

- **Returns**
  - **sys**: StateSpace
    - The randomly created linear system

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

See also:

- rss

**Notes**

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

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

Todo

Add example with MIMO system

control.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`

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, 2.57645470])
# This is the magnitude of the frequency response from the 2nd input to the 1st output, and the phase (in radians) of the frequency response from the 1st input to the 2nd output, for s = 0.1i, i, 10i.
```

control.matlab.impulse(sys, T=None, input=0, output=None, **keywords)

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)

`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.

**keywords**: Additional keyword arguments control the solution algorithm for the differential equations. These arguments are passed on to the function lsim(), which in turn passes them on to scipy.integrate.odeint(). See the documentation for scipy.integrate.odeint() for information about these arguments.

**Returns**

`yout`: array

Response of the system

`T`: array

Time values of the output
See also:

`lsim, step, initial`

Examples

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

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

Initial condition response of a linear system

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

Parameters:

- `sys`: StateSpace, or TransferFunction
  - LTI system to simulate
- `T`: array-like object, optional
  - Time vector (argument is autocomputed if not given)
- `X0`: array-like object or number, optional
  - Initial condition (default = 0)
  - Numbers are converted to constant arrays with the correct shape.
- `input`: int
  - This input is ignored, but present for compatibility with step and impulse.
- `output`: int
  - If given, index of the output that is returned by this simulation.
- `**keywords`: Additional keyword arguments control the solution algorithm for the differential equations. These arguments are passed on to the function `lsim()`, which in turn passes them on to `scipy.integrate.odeint()`. See the documentation for `scipy.integrate.odeint()` for information about these arguments.

Returns:

- `yout`: array
  - Response of the system
- `T`: array
  - Time values of the output

See also:

`lsim, step, impulse`

Examples

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

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

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).

**keywords**:

- Additional keyword arguments control the solution algorithm for the differential equations. These arguments are passed on to the function `scipy.integrate.odeint()`. See the documentation for `scipy.integrate.odeint()` for information about these arguments.

**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
gm, pm, Wcg, Wcp = margin(sys, U, T, X0)
```

control.matlab.margin(*args)

Calculate gain and phase margins and associated crossover frequencies

Function `margin` takes either 1 or 3 parameters.

**Parameters**

- **sys**: StateSpace or TransferFunction
  - Linear SISO system

- **mag, phase, w**: array_like
  - Input magnitude, phase (in deg.), and frequencies (rad/sec) from bode frequency response data

**Returns**

- **gm, pm, Wcg, Wcp**: float
Gain margin \(gm\), phase margin \(pm\) (in deg), gain crossover frequency (corresponding to phase margin) and phase crossover frequency (corresponding to gain margin), in rad/sec of SISO open-loop. If more than one crossover frequency is detected, returns the lowest corresponding margin.

**Examples**

```python
>>> sys = ss("1. -2; 3. -4", "5.; 7", "6. 8", "9.")
>>> gm, pm, wg, wp = margin(sys)
margin: no magnitude crossings found
```

**Todo**

better example system!

```python
#>>> gm, pm, wg, wp = margin(mag, phase, w)
```

**control.matlab.ngrid**

Nichols chart grid

Plots a Nichols chart grid on the current axis, or creates a new chart if no plot already exists.

**Parameters**

- `cl_mags` : array-like (dB), optional
  
  Array of closed-loop magnitudes defining the iso-gain lines on a custom Nichols chart.
  
- `cl_phases` : array-like (degrees), optional
  
  Array of closed-loop phases defining the iso-phase lines on a custom Nichols chart.

  Must be in the range \(-360 < cl_phases < 0\)

**Returns**

- `None`

**control.matlab.pole**

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`

**Notes**

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

**control.matlab.rlocus**

Root locus plot

The root-locus plot has a callback function that prints pole location, gain and damping to the Python console on mouseclicks on the root-locus graph.

4.17. Additional functions
Parameters `sys`: `StateSpace` or `TransferFunction`:

- Linear system

**klist**: `iterable`, `optional`:

  - optional list of gains

**xlim**: control of x-axis range, normally with tuple, for

  - other options, see matplotlib.axes

**ylim**: control of y-axis range

**Plot**: boolean (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`:

- list of roots for each gain

- `klist`:

  - list of gains used to compute roots

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

Create a stable `continuous` random state space object.

Parameters `states`: integer:

- Number of state variables

**inputs**: integer:

- Number of system inputs

**outputs**: integer:

- Number of system outputs

Returns `sys`: `StateSpace`:

- The randomly created linear system

Raises `ValueError`:

- if any input is not a positive integer

See also:

- `drss`

Notes

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

`control.matlab.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: Lti (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: 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)
```
control.matlab.\texttt{ss2tf}(\texttt{*args})

Transform a state space system to a transfer function.

The function accepts either 1 or 4 parameters:

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

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

For details see: \texttt{ss()}

\textbf{Parameters}  \texttt{sys} : StateSpace :

A linear system

\textbf{A: array_like or string} :

System matrix

\textbf{B: array_like or string} :

Control matrix

\textbf{C: array_like or string} :

Output matrix

\textbf{D: array_like or string} :

Feedthrough matrix

\textbf{Returns}  \texttt{out} : TransferFunction :

New linear system in transfer function form

\textbf{Raises}  \texttt{ValueError} :

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

\textbf{TypeError} :

if \texttt{sys} is not a StateSpace object

\textbf{See also:}  \texttt{tf, ss, tf2ss}

\textbf{Examples}

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

```python
>>> sys2 = ss2tf(sys1)
```  

control.matlab.\texttt{ssdata}(\texttt{sys})

Return state space data objects for a system

\textbf{Parameters}  \texttt{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

```python
control.matlab.step(sys, T=None, X0=0.0, input=0, output=None, **keywords)
```

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.
- `**keywords`: Additional keyword arguments control the solution algorithm for the differential equations. These arguments are passed on to the function `control.forced_response()`, which in turn passes them on to `scipy.integrate.odeint()`. See the documentation for `scipy.integrate.odeint()` for information about these arguments.

**Returns**

- `yout`: array
  - Response of the system
- `T`: array
  - Time values of the output

See also:

`lsim, initial, impulse`

**Examples**

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

```python
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: 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

Todo
The next paragraph contradicts the comment in the example! Also “input” should come before “output” in the sentence:
“from the (j+1)st output to the (i+1)st input”

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

The coefficients [2, 3, 4] denote the polynomial 2 · s^2 + 3 · s + 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)
```
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)
```

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)

```python
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 TransferFunction object or a MIMO StateSpace object

**See also**
- `pole`

**Notes**
This function is a wrapper for StateSpace.zero and TransferFunction.zero.

---

**Todo**
The following functions should be documented in their own modules! This is only a temporary solution.

```python
control.pzmap.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.

```python
control.freqplot.nyquist(syslist, omega=None, Plot=True, color='b', labelFreq=0, *args, **kwargs)
```
Nyquist plot for a system

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

**Parameters**
- `syslist`: list of Lti
  - List of linear input/output systems (single system is OK)
- `omega`: freq_range
  - Range of frequencies (list or bounds) in rad/sec
- `Plot`: boolean
  - If True, plot magnitude
labelFreq : int
    Label every nth frequency on the plot

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

Returns
real : array
    real part of the frequency response array
imag : array
    imaginary part of the frequency response array
freq : array
    frequencies

Examples

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

control.nichols.nichols(syslist, omega=None, grid=True)
    Nichols plot for a system
    Plots a Nichols plot for the system over a (optional) frequency range.

Parameters
syslist : list of Lti, or Lti
    List of linear input/output systems (single system is OK)
omega : array_like
    Range of frequencies (list or bounds) in rad/sec
grid : boolean, optional
    True if the plot should include a Nichols-chart grid. Default is True.

Returns
None :

control.statefbk.place(A, B, p)
    Place closed loop eigenvalues

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

Returns
K : 2-d array
    Gains such that A - B K has given eigenvalues
Examples

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

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

Linear quadratic regulator design

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

\[
J = \int_0^\infty x' Q x + u' R u + 2 x' N u
\]

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)

**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])
```

control.statefbk 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)
```

control.statefbk.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)
```

control.statefbk.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).

Returns gram: array:
Gramian of system

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

Examples

```python
>>> Wc = gram(sys,'c')
>>> Wo = gram(sys,'o')
```

control.delay.pade(T, n=1)
Create a linear system that approximates a delay.
Return the numerator and denominator coefficients of the Pade approximation.

Parameters T: number

4.17. Additional functions
time delay

**n**: integer

order of approximation

**Returns** **num, den**: array

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

**Notes**


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

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

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

**Parameters** **P, C**: Lti

Linear input/output systems (process and control)

**omega**: array

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

**Returns** **None**: 

**control.ctrlutil**.unwrap(*angle*, **period=6.28**)  
Unwrap a phase angle to give a continuous curve

**Parameters** **X**: array_like

Input array

**period**: number

Input period (usually either 2*pi or 360)

**Returns** **Y**: array_like

Output array, with jumps of period/2 eliminated

**Examples**

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

**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

\[ AX + XC + C = 0 \]

where A and Q are square matrices.

X = lyap(A,Q,None,E) solves the generalized continuous-time Lyapunov equation
\[ AXE^T + EXA^T + Q = 0 \]

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

**control.mateqn.dlyap(\( A, Q, C=None, E=None \))**
dlyap(\( A, Q \)) solves the discrete-time Lyapunov equation

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

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

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

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

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

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

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

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

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

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

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

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 \( A - B G \).

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

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

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

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

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

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

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.,

\[ \text{eigenvalues of } A - B G \]

\((X, L, G) = \text{dare}(A, B, Q, R, S=None, E=None)\) solves the generalized discrete-time algebraic Riccati equation

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

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 \( A - B G, E \).
Examples
Indices and tables

• genindex
• search
C
control, 5
control.frdata, 38
control.lti, 33
control.matlab, 41
control.statesp, 34
control.timeresp, 24
control.xferfcn, 36
C

c
control, 5
control.frdata, 38
control.lti, 33
control.matlab, 41
control.statesp, 34
control.timeresp, 24
control.xferfcn, 36
Index

A
ack() (in module control), 14
append() (control.statesp.StateSpace method), 35

B
balred() (in module control), 30
bode() (in module control.matlab), 46
bode_plot() (in module control), 21
box_grid() (in module control.phaseplot), 29

C
c2d() (in module control.matlab), 46
care() (in module control), 14
care() (in module control.mateqn), 65
control (module), 5
control.frdata (module), 38
control.lti (module), 33
control.matlab (module), 41
control.timeresp (module), 24
control.xferfcn (module), 36
ctrb() (in module control), 14
ctrb() (in module control.statefbk), 62

D
damp() (in module control.matlab), 47
dare() (in module control.mateqn), 65
dcgain() (in module control), 15
dcgain() (in module control.matlab), 47
default_frequency_range() (in module control.freqplot), 23
dlyap() (in module control), 15
dlyap() (in module control.mateqn), 65
drss() (in module control), 8
drss() (in module control.matlab), 48

E
evalfr() (control.frdata.FRD method), 38
evalfr() (control.statesp.StateSpace method), 35
evalfr() (control.xferfcn.TransferFunction method), 37
evalfr() (in module control), 16
evalfr() (in module control.matlab), 48

F
feedback() (control.frdata.FRD method), 38
feedback() (control.statesp.StateSpace method), 35
feedback() (control.xferfcn.TransferFunction method), 37
feedback() (in module control), 12
forced_response() (in module control), 25
FRD (class in control.frdata), 38
frd() (in module control.matlab), 49
freqresp() (control.frdata.FRD method), 38
freqresp() (control.statesp.StateSpace method), 35
freqresp() (control.xferfcn.TransferFunction method), 37
freqresp() (in module control), 17
freqresp() (in module control.matlab), 49

G
gangof4() (in module control.freqplot), 64
gangof4_plot() (in module control), 23
gram() (in module control), 16
gram() (in module control.statefbk), 63

H
horner() (control.statesp.StateSpace method), 35
horner() (control.xferfcn.TransferFunction method), 37
hsvd() (in module control), 31

I
impulse() (in module control.matlab), 50
initial() (in module control.matlab), 51
initial_response() (in module control), 26
isctime() (control.lti.Lti method), 33
isctime() (in module control), 8
isctime() (in module control.lti), 34
isdt ime() (control.lti.Lti method), 33
isdt ime() (in module control), 8
isdt ime() (in module control.lti), 34
issys() (in module control), 9

75
Index

L
lqr() (in module control), 29
lqr() (in module control.statefbk), 62
lsim() (in module control.matlab), 51
Lti (class in control.lti), 33
lyap() (in module control), 17
lyap() (in module control.mateqn), 64

M
margin() (in module control), 18
margin() (in module control.matlab), 52
markov() (in module control), 18
minreal() (control.statesp.StateSpace method), 35
minreal() (control.xferfcn.TransferFunction method), 37
modred() (in module control), 31

N
negate() (in module control), 12
ngrid() (in module control.matlab), 53
nichols() (in module control Nichols), 61
nichols_plot() (in module control), 23
nyquist() (in module control.freqplot), 60
nyquist_plot() (in module control), 22

O
obsv() (in module control), 19
obsv() (in module control.statefbk), 63

P
pade() (in module control), 9
pade() (in module control.delay), 63
parallel() (in module control), 13
phase_crossover_frequencies() (in module control), 19
phase_plot() (in module control.phaseplot), 28
place() (in module control), 30
place() (in module control.statefbk), 61
pole() (control.statesp.StateSpace method), 35
pole() (control.xferfcn.TransfertFunction method), 37
pole() (in module control), 19
pole() (in module control.matlab), 53
pzmap() (in module control.pzmap), 60

R
returnScipySignalLti() (control.statesp.StateSpace method), 35
returnScipySignalLti() (control.xferfcn.TransfertFunction method), 37
rlocus() (in module control.matlab), 53
root_locus() (in module control), 20
rss() (in module control.matlab), 54

S
sample() (control.statesp.StateSpace method), 35
sample_system() (in module control), 9
series() (in module control), 13
ss() (in module control), 5
ss() (in module control.matlab), 54
ss2tf() (module control), 9
ss2tf() (in module control.matlab), 55
ssdata() (in module control), 10
ssdata() (in module control.matlab), 56
stability_margins() (in module control), 20
StateSpace (class in control), 5
StateSpace (class in control.statesp), 34
step() (in module control.matlab), 57
step_response() (in module control), 27

T
tf() (in module control), 7
tf() (in module control.matlab), 57
tf2ss() (module control), 11
tf2ss() (in module control.matlab), 59
tfdata() (in module control), 11
tfdata() (in module control.matlab), 59
timebase() (in module control), 12
timebase() (in module control.lti), 34
timebaseEqual() (in module control), 12
timebaseEqual() (in module control.lti), 34
TransferFunction (class in control), 6
TransferFunction (class in control.xferfcn), 36

U
unwrap() (in module control), 32
unwrap() (in module control.ctrlutil), 64

Z
zero() (control.statesp.StateSpace method), 36
zero() (control.xferfcn.TransfertFunction method), 38
zero() (in module control), 21
zero() (in module control.matlab), 60