Python Control Library Documentation

Release dev

RMM

Dec 28, 2020

CONTENTS

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

Features

- Linear input/output systems in state-space and frequency domain
- Block diagram algebra: serial, parallel, and feedback interconnections
- Time response: initial, step, impulse
- Frequency response: Bode and Nyquist plots
- Control analysis: stability, reachability, observability, stability margins
- Control design: eigenvalue placement, LQR, H2, Hinf
- Model reduction: balanced realizations, Hankel singular values
- Estimator design: linear quadratic estimator (Kalman filter)

Documentation

CHAPTER

ONE

INTRODUCTION

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

1.1 Overview of the toolbox

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

1.2 Some differences from MATLAB

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

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

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

1.3 Installation

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

To install using pip:

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

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

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

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

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

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

```
conda install numpy scipy matplotlib    # if not yet installed
conda install -c conda-forge control
```

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

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

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

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

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

1.4 Getting started

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

```
>>> import control
```

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

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

CHAPTER

TWO

LIBRARY CONVENTIONS

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

2.1 LTI system representation

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

2.1.1 State space systems

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

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

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

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

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

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

2.1.2 Transfer functions

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

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

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

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

sys = TransferFunction(num, den)

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

2.1.3 FRD (frequency response data) systems

The *FrequencyResponseData* (FRD) class is used to represent systems in frequency response data form.

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

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

2.1.4 Discrete time systems

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

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

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

Systems must have compatible timebases in order to be combined. A system with timebase *None* can be combined with a system having a specified timebase; the result will have the timebase of the latter system. Similarly, a discrete time system with unspecified sampling time (*dt = True*) can be combined with a system having a specified sampling time; the result will be a discrete time system with the sample time of the latter system. For continuous time systems, the *sample_system()* function or the *StateSpace.sample()* and *TransferFunction.sample()* methods can be used to create a discrete time system from a continuous time system. See *Utility functions and conversions*. The default value of 'dt' can be changed by changing the values of `control.config.defaults['statesp.default_dt']` and `control.config.defaults['xferfcn.default_dt']`.

2.1.5 Conversion between representations

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

2.2 Time series data

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

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

Types:

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

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

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

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

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

Same for X, Y
```

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

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

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

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

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

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

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

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

```
ft = D * U
```

2.3 Package configuration parameters

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

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

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

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

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

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

Selected variables that can be configured, along with their default values:

- bode.dB (False): Bode plot magnitude plotted in dB (otherwise powers of 10)
- bode.deg (True): Bode plot phase plotted in degrees (otherwise radians)
- bode.Hz (False): Bode plot frequency plotted in Hertz (otherwise rad/sec)
- bode.grid (True): Include grids for magnitude and phase plots
- freqplot.number_of_samples (None): Number of frequency points in Bode plots
- freqplot.feature_periphery_decade (1.0): How many decades to include in the frequency range on both sides of features (poles, zeros).
- statesp.use_numpy_matrix (True): set the return type for state space matrices to *numpy.matrix* (verus numpy.ndarray)
- statesp.default_dt and xferfcn.default_dt (None): set the default value of dt when constructing new LTI systems
- statesp.remove_useless_states (True): remove states that have no effect on the input-output dynamics of the system

Additional parameter variables are documented in individual functions

Functions that can be used to set standard configurations:

`reset_defaults`()	Reset configuration values to their default (initial) values.
`use_fbs_defaults`()	Use Feedback Systems (FBS) compatible settings.
`use_matlab_defaults`()	Use MATLAB compatible configuration settings.
`use_numpy_matrix`([flag, warn])	Turn on/off use of Numpy *matrix* class for state space operations.
`use_legacy_defaults`(version)	Sets the defaults to whatever they were in a given release.

2.3.1 control.reset_defaults

`control.reset_defaults()`

Reset configuration values to their default (initial) values.

2.3.2 control.use_fbs_defaults

`control.use_fbs_defaults()`

Use Feedback Systems (FBS) compatible settings.

The following conventions are used:

- Bode plots plot gain in powers of ten, phase in degrees, frequency in rad/sec, no grid

2.3.3 control.use_matlab_defaults

`control.use_matlab_defaults()`

Use MATLAB compatible configuration settings.

The following conventions are used:

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

2.3.4 control.use_numpy_matrix

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

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

Parameters

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

Notes

Prior to release 0.9.x, the default type for 2D arrays is the Numpy *matrix* class. Starting in release 0.9.0, the default type for state space operations is a 2D array.

2.3.5 control.use_legacy_defaults

`control.use_legacy_defaults(version)`

Sets the defaults to whatever they were in a given release.

Parameters **version** (*string*) – version number of the defaults desired. ranges from '0.1' to '0.8.4'.

CHAPTER

THREE

FUNCTION REFERENCE

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

3.1 System creation

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

3.1.1 control.ss

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

Create a state space system.

The function accepts either 1, 4 or 5 parameters:

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

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

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

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

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

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

Parameters

- **`sys`** (`StateSpace or TransferFunction`) – A linear system
- **`A`** (*`array_like or string`*) – System matrix

- **B** (*array_like or string*) – Control matrix
- **C** (*array_like or string*) – Output matrix
- **D** (*array_like or string*) – Feed forward matrix
- **dt** (*If present, specifies the sampling period and a discrete time*) – system is created

Returns **out** – The new linear system

Return type *StateSpace*

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

See also:

StateSpace, *tf*, *ss2tf*, *tf2ss*

Examples

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

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

3.1.2 control.tf

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

Create a transfer function system. Can create MIMO systems.

The function accepts either 1, 2, or 3 parameters:

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

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

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

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

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

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

Parameters

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

- **den** (*array_like, or list of list of array_like*) – Polynomial coefficients of the denominator

Returns **out** – The new linear system

Return type *TransferFunction*

Raises

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

See also:

TransferFunction, *ss*, *ss2tf*, *tf2ss*

Notes

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

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

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

Examples

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

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

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

3.1.3 control.frd

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

Construct a frequency response data model

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

This function can be called in different ways:

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

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

Parameters

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

Returns sys – New frequency response system

Return type FRD

See also:

FRD, *ss*, *tf*

3.1.4 control.rss

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

Create a stable *continuous* random state space object.

Parameters

- **states** (*integer*) – Number of state variables
- **inputs** (*integer*) – Number of system inputs
- **outputs** (*integer*) – Number of system outputs

Returns sys – The randomly created linear system

Return type *StateSpace*

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

See also:

drss

Notes

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

3.1.5 control.drss

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

Create a stable *discrete* random state space object.

Parameters

- **states** (*integer*) – Number of state variables
- **inputs** (*integer*) – Number of system inputs
- **outputs** (*integer*) – Number of system outputs

Returns sys – The randomly created linear system

Return type *StateSpace*

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

See also:

rss

Notes

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

3.2 System interconnections

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

3.2.1 control.append

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

Group models by appending their inputs and outputs

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

Parameters

- **sys1** (`StateSpace or Transferfunction`) – LTI systems to combine
- **sys2** (`StateSpace or Transferfunction`) – LTI systems to combine
- **..** (`StateSpace or Transferfunction`) – LTI systems to combine
- **sysn** (`StateSpace or Transferfunction`) – LTI systems to combine

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

Return type LTI system

Examples

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

3.2.2 control.connect

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

Index-based interconnection of an LTI system.

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

NOTE: Inputs and outputs are indexed starting at 1 and negative values correspond to a negative feedback interconnection.

Parameters

- **sys** (*StateSpace Transferfunction*) – System to be connected
- **Q** (*2D array*) – Interconnection matrix. First column gives the input to be connected. The second column gives the index of an output that is to be fed into that input. Each additional column gives the index of an additional input that may be optionally added to that input. Negative values mean the feedback is negative. A zero value is ignored. Inputs and outputs are indexed starting at 1 to communicate sign information.
- **inputv** (*1D array*) – list of final external inputs, indexed starting at 1
- **outputv** (*1D array*) – list of final external outputs, indexed starting at 1

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

Return type LTI system

Examples

```
>>> sys1 = ss([[1., -2], [3., -4]], [[5.], [7]], [[6, 8]], [[9.]])
>>> sys2 = ss([[-1.]], [[1.]], [[1.]], [[0.]])
>>> sys = append(sys1, sys2)
>>> Q = [[1, 2], [2, -1]]  # negative feedback interconnection
>>> sysc = connect(sys, Q, [2], [1, 2])
```

3.2.3 control.feedback

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

Feedback interconnection between two I/O systems.

Parameters

- **sys1** (*scalar, StateSpace, TransferFunction, FRD*) – The primary process.
- **sys2** (*scalar, StateSpace, TransferFunction, FRD*) – The feedback process (often a feedback controller).
- **sign** (*scalar*) – The sign of feedback. *sign* = -1 indicates negative feedback, and *sign* = 1 indicates positive feedback. *sign* is an optional argument; it assumes a value of -1 if not specified.

Returns **out**

Return type *StateSpace* or *TransferFunction*

Raises

- **ValueError** – if *sys1* does not have as many inputs as *sys2* has outputs, or if *sys2* does not have as many inputs as *sys1* has outputs
- **NotImplementedError** – if an attempt is made to perform a feedback on a MIMO TransferFunction object

See also:

`series`, `parallel`

Notes

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

3.2.4 control.negate

`control.negate(sys)`

Return the negative of a system.

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

Returns out

Return type *StateSpace* or *TransferFunction*

Notes

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

Examples

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

3.2.5 control.parallel

`control.parallel(sys1, *sysn)`

Return the parallel connection sys1 + sys2 (+ … + sysn)

Parameters

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

Returns out

Return type scalar, *StateSpace*, or *TransferFunction*

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

See also:

series, *feedback*

Notes

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

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

Examples

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

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

3.2.6 control.series

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

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

Parameters

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

Returns out

Return type scalar, *StateSpace*, or *TransferFunction*

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

See also:

parallel, *feedback*

Notes

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

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

Examples

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

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

See also the *Input/output systems* module, which can be used to create and interconnect nonlinear input/output systems.

3.3 Frequency domain plotting

`bode_plot`(syslist[, omega, plot, ...])	Bode plot for a system
`nyquist_plot`(syslist[, omega, plot, ...])	Nyquist plot for a system
`gangof4_plot`(P, C[, omega])	Plot the "Gang of 4" transfer functions for a system
`nichols_plot`(sys_list[, omega, grid])	Nichols plot for a system
`nichols_grid`([cl_mags, cl_phases, line_style])	Nichols chart grid

3.3.1 control.bode_plot

control.**bode_plot**(*syslist*, *omega=None*, *plot=True*, *omega_limits=None*, *omega_num=None*, *margins=None*, **args*, ***kwargs*)

Bode plot for a system

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

Parameters

- **syslist** (*linsys*) – List of linear input/output systems (single system is OK)
- **omega** (*list*) – List of frequencies in rad/sec to be used for frequency response
- **dB** (*bool*) – If True, plot result in dB. Default is false.
- **Hz** (*bool*) – If True, plot frequency in Hz (omega must be provided in rad/sec). Default value (False) set by config.defaults['bode.Hz']
- **deg** (*bool*) – If True, plot phase in degrees (else radians). Default value (True) config.defaults['bode.deg']
- **plot** (*bool*) – If True (default), plot magnitude and phase
- **omega_limits** (*tuple, list, .. of two values*) – Limits of the to generate frequency vector. If Hz=True the limits are in Hz otherwise in rad/s.
- **omega_num** (*int*) – Number of samples to plot. Defaults to config.defaults['freqplot.number_of_samples'].
- **margins** (*bool*) – If True, plot gain and phase margin.
- ***args** (`matplotlib.pyplot.plot()` positional properties, optional) – Additional arguments for *matplotlib* plots (color, linestyle, etc)
- ****kwargs** (`matplotlib.pyplot.plot()` keyword properties, optional) – Additional keywords (passed to *matplotlib*)

Returns

- **mag** (*array (list if len(syslist) > 1)*) – magnitude

- **phase** (*array (list if len(syslist) > 1)*) – phase in radians
- **omega** (*array (list if len(syslist) > 1)*) – frequency in rad/sec

Other Parameters grid (*bool*) – If True, plot grid lines on gain and phase plots. Default is set by *config.defaults['bode.grid']*.

The default values for Bode plot configuration parameters can be reset using the *config.defaults* dictionary, with module name 'bode'.

Notes

1. Alternatively, you may use the lower-level method `(mag, phase, freq) = sys.freqresp(freq)` to generate the frequency response for a system, but it returns a MIMO response.
2. If a discrete time model is given, the frequency response is plotted along the upper branch of the unit circle, using the mapping z = exp(j omega dt) where omega ranges from 0 to pi/dt and dt is the discrete timebase. If not timebase is specified (dt = True), dt is set to 1.

Examples

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

3.3.2 control.nyquist_plot

`control.nyquist_plot`(*syslist*, *omega=None*, *plot=True*, *label_freq=0*, *arrowhead_length=0.1*, *arrowhead_width=0.1*, *color=None*, **args*, ***kwargs*)

Nyquist plot for a system

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

Parameters

- **syslist** (*list of LTI*) – List of linear input/output systems (single system is OK)
- **omega** (*freq_range*) – Range of frequencies (list or bounds) in rad/sec
- **Plot** (*boolean*) – If True, plot magnitude
- **color** (*string*) – Used to specify the color of the plot
- **label_freq** (*int*) – Label every nth frequency on the plot
- **arrowhead_width** (*arrow head width*) –
- **arrowhead_length** (*arrow head length*) –
- ***args** (`matplotlib.pyplot.plot()` positional properties, optional) – Additional arguments for *matplotlib* plots (color, linestyle, etc)
- ****kwargs** (`matplotlib.pyplot.plot()` keyword properties, optional) – Additional keywords (passed to *matplotlib*)

Returns

- **real** (*array*) – real part of the frequency response array
- **imag** (*array*) – imaginary part of the frequency response array
- **freq** (*array*) – frequencies

Examples

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

3.3.3 control.gangof4_plot

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

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** (*LTI*) – Linear input/output systems (process and control)
- **C** (*LTI*) – Linear input/output systems (process and control)
- **omega** (*array*) – Range of frequencies (list or bounds) in rad/sec
- ****kwargs** (`matplotlib.pyplot.plot()` keyword properties, optional) – Additional keywords (passed to *matplotlib*)

Returns

Return type None

3.3.4 control.nichols_plot

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

Nichols plot for a system

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

Parameters

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

Returns

Return type None

3.3.5 control.nichols_grid

control.**nichols_grid**(*cl_mags=None*, *cl_phases=None*, *line_style='dotted'*)

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
- **line_style** (*string, optional*) – Matplotlib linestyle

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

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

3.4 Time domain simulation

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

3.4.1 control.forced_response

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

Simulate the output of a linear system.

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

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

Parameters

- **sys** (*LTI (StateSpace or TransferFunction)*) – LTI system to simulate
- **T** (array_like, optional for discrete LTI *sys*) – Time steps at which the input is defined; values must be evenly spaced.
- **U** (*array_like or float, 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 float, optional*) – Initial condition (default = 0).
- **transpose** (*bool, optional (default=False)*) – If True, transpose all input and output arrays (for backward compatibility with MATLAB and scipy.signal.lsim())
- **interpolate** (*bool, optional (default=False)*) – If True and system is a discrete time system, the input will be interpolated between the given time steps and the

output will be given at system sampling rate. Otherwise, only return the output at the times given in *T*. No effect on continuous time simulations (default = False).

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

Returns

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

See also:

step_response, *initial_response*, *impulse_response*

Notes

For discrete time systems, the input/output response is computed using the `scipy.signal.dlsim()` function.

For continuous time systems, the output is computed using the matrix exponential *exp(A t)* and assuming linear interpolation of the inputs between time points.

Examples

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

See *Time series data*.

3.4.2 control.impulse_response

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

Impulse response of a linear system

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

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

Parameters

- **sys** (`StateSpace, TransferFunction`) – LTI system to simulate
- **T** (*array_like or float, optional*) – Time vector, or simulation time duration if a scalar (time vector is autocomputed if not given; see *step_response()* for more detail)
- **X0** (*array_like or float, optional*) – Initial condition (default = 0)

  Numbers are converted to constant arrays with the correct shape.
- **input** (*int*) – Index of the input that will be used in this simulation.

- **output** (*int*) – Index of the output that will be used in this simulation. Set to None to not trim outputs
- **T_num** (*int, optional*) – Number of time steps to use in simulation if T is not provided as an array (autocomputed if not given); ignored if sys is discrete-time.
- **transpose** (*bool*) – If True, transpose all input and output arrays (for backward compatibility with MATLAB and `scipy.signal.lsim()`)
- **return_x** (*bool*) – If True, return the state vector (default = False).
- **squeeze** (*bool, optional (default=True)*) – If True, remove single-dimensional entries from the shape of the output. For single output systems, this converts the output response to a 1D array.

Returns

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

See also:

forced_response, *initial_response*, *step_response*

Notes

This function uses the *forced_response* function to compute the time response. For continuous time systems, the initial condition is altered to account for the initial impulse.

Examples

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

3.4.3 control.initial_response

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

Initial condition response of a linear system

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

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

Parameters

- **sys** (`StateSpace or TransferFunction`) – LTI system to simulate
- **T** (*array_like or float, optional*) – Time vector, or simulation time duration if a number (time vector is autocomputed if not given; see `step_response()` for more detail)
- **X0** (*array_like or float, 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
- **T_num** (*int, optional*) – Number of time steps to use in simulation if T is not provided as an array (autocomputed if not given); ignored if sys is discrete-time.
- **transpose** (*bool*) – If True, transpose all input and output arrays (for backward compatibility with MATLAB and `scipy.signal.lsim()`)
- **return_x** (*bool*) – If True, return the state vector (default = False).
- **squeeze** (*bool, optional (default=True)*) – If True, remove single-dimensional entries from the shape of the output. For single output systems, this converts the output response to a 1D array.

Returns

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

See also:

forced_response, *impulse_response*, *step_response*

Notes

This function uses the *forced_response* function with the input set to zero.

Examples

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

3.4.4 control.input_output_response

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

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

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

Parameters

- **sys** (`InputOutputSystem`) – Input/output system to simulate.
- **T** (*array-like*) – Time steps at which the input is defined; values must be evenly spaced.
- **U** (*array-like or number, optional*) – Input array giving input at each time *T* (default = 0).
- **X0** (*array-like or number, optional*) – Initial condition (default = 0).
- **return_x** (*bool, optional*) – If True, return the values of the state at each time (default = False).

- **squeeze** (*bool, optional*) – If True (default), squeeze unused dimensions out of the output response. In particular, for a single output system, return a vector of shape (nsteps) instead of (nsteps, 1).

Returns

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

Raises

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

3.4.5 control.step_response

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

Step response of a linear system

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

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

Parameters

- **sys** (StateSpace or TransferFunction) – LTI system to simulate
- **T** (*array_like or float, optional*) – Time vector, or simulation time duration if a number. If T is not provided, an attempt is made to create it automatically from the dynamics of sys. If sys is continuous-time, the time increment dt is chosen small enough to show the fastest mode, and the simulation time period tfinal long enough to show the slowest mode, excluding poles at the origin and pole-zero cancellations. If this results in too many time steps (>5000), dt is reduced. If sys is discrete-time, only tfinal is computed, and final is reduced if it requires too many simulation steps.
- **X0** (*array_like or float, optional*) – Initial condition (default = 0)

  Numbers are converted to constant arrays with the correct shape.
- **input** (*int*) – Index of the input that will be used in this simulation.
- **output** (*int*) – Index of the output that will be used in this simulation. Set to None to not trim outputs
- **T_num** (*int, optional*) – Number of time steps to use in simulation if T is not provided as an array (autocomputed if not given); ignored if sys is discrete-time.
- **transpose** (*bool*) – If True, transpose all input and output arrays (for backward compatibility with MATLAB and scipy.signal.lsim())
- **return_x** (*bool*) – If True, return the state vector (default = False).
- **squeeze** (*bool, optional (default=True)*) – If True, remove single-dimensional entries from the shape of the output. For single output systems, this converts the output response to a 1D array.

Returns

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

See also:

forced_response, *initial_response*, *impulse_response*

Notes

This function uses the *forced_response* function with the input set to a unit step.

Examples

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

3.4.6 control.phase_plot

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

Phase plot for 2D dynamical systems

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

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

Parameters

- **func** (*callable(x, t, ..)*) – Computes the time derivative of y (compatible with odeint). The function should be the same for as used for `scipy.integrate`. Namely, it should be a function of the form dxdt = F(x, t) that accepts a state x of dimension 2 and returns a derivative dx/dt of dimension 2.
- **X** (*3-element sequences, optional, as [start, stop, npts]*) – Two 3-element sequences specifying x and y coordinates of a grid. These arguments are passed to linspace and meshgrid to generate the points at which the vector field is plotted. If absent (or None), the vector field is not plotted.
- **Y** (*3-element sequences, optional, as [start, stop, npts]*) – Two 3-element sequences specifying x and y coordinates of a grid. These arguments are passed to linspace and meshgrid to generate the points at which the vector field is plotted. If absent (or None), the vector field is not plotted.
- **scale** (*float, optional*) – Scale size of arrows; default = 1
- **X0** (*ndarray of initial conditions, optional*) – List of initial conditions from which streamlines are plotted. Each initial condition should be a pair of numbers.
- **T** (*array-like or number, optional*) – Length of time to run simulations that generate streamlines. If a single number, the same simulation time is used for all initial conditions. Otherwise, should be a list of length len(X0) that gives the simulation time for each initial condition. Default value = 50.

- **lingrid** (*integer or 2-tuple of integers, optional*) – Argument is either N or (N, M). 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** (*integer or tuple (integer, float), optional*) – If a single integer N is given, draw N arrows using equally space time points. If a tuple (N, lambda) is given, draw N arrows using exponential time constant lambda
- **timepts** (*array-like, optional*) – Draw arrows at the given list times [t1, t2, . . .]
- **parms** (*tuple, optional*) – List of parameters to pass to vector field: *func(x, t, *parms)*

See also:

box_grid construct box-shaped grid of initial conditions

Examples

3.5 Block diagram algebra

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

3.6 Control system analysis

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

3.6.1 control.dcgain

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

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

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

Return type ndarray

3.6.2 control.evalfr

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

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

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

Parameters

- **sys** (StateSpace or TransferFunction) – Linear system
- **x** (*scalar*) – Complex number

Returns **fresp**

Return type ndarray

See also:

freqresp, bode

Notes

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

Examples

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

Todo: Add example with MIMO system

3.6.3 control.freqresp

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

Frequency response of an LTI system at multiple angular frequencies.

Parameters

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

Returns

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

See also:

`evalfr`, `bode`

Notes

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

Examples

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

Todo: Add example with MIMO system

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

3.6.4 control.margin

`control.margin(sysdata)`

Calculate gain and phase margins and associated crossover frequencies

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

sys [StateSpace or TransferFunction] Linear SISO system

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

Returns

- **gm** (*float*) – Gain margin
- **pm** (*float*) – Phase margin (in degrees)
- **wg** (*float*) – Frequency for gain margin (at phase crossover, phase = -180 degrees)

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

Examples

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

3.6.5 control.stability_margins

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

Calculate stability margins and associated crossover frequencies.

Parameters

- **`sysdata`** (*`LTI system or (mag, phase, omega) sequence`*) –

  sys [LTI system] Linear SISO system

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

- **`returnall`** (*`bool, optional`*) – If true, return all margins found. If False (default), return only the minimum stability margins. For frequency data or FRD systems, only margins in the given frequency region can be found and returned.
- **`epsw`** (*`float, optional`*) – Frequencies below this value (default 0.0) are considered static gain, and not returned as margin.

Returns

- **gm** (*float or array_like*) – Gain margin
- **pm** (*float or array_loke*) – Phase margin
- **sm** (*float or array_like*) – Stability margin, the minimum distance from the Nyquist plot to -1
- **wg** (*float or array_like*) – Frequency for gain margin (at phase crossover, phase = -180 degrees)
- **wp** (*float or array_like*) – Frequency for phase margin (at gain crossover, gain = 1)
- **ws** (*float or array_like*) – Frequency for stability margin (complex gain closest to -1)

3.6.6 control.phase_crossover_frequencies

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

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

Parameters **sys** (*SISO LTI system*) –

Returns

- **omega** (*ndarray*) – 1d array of (non-negative) frequencies where Nyquist plot intersects the real axis
- **gain** (*ndarray*) – 1d array of corresponding gains

Examples

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

3.6.7 control.pole

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

Compute system poles.

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

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

Return type ndarray

Raises **NotImplementedError** – when called on a TransferFunction object

See also:

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

3.6.8 control.zero

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

Compute system zeros.

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

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

Return type ndarray

Raises **NotImplementedError** – when called on a MIMO system

See also:

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

3.6.9 control.pzmap

control.**pzmap**(*sys*, *plot=None*, *grid=None*, *title='Pole Zero Map'*, ***kwargs*)

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, optional*) – If `True` a graph is generated with Matplotlib, otherwise the poles and zeros are only computed and returned.
- **grid** (*boolean (default = False)*) – If True plot omega-damping grid.

Returns

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

3.6.10 control.root_locus

control.**root_locus**(*sys*, *kvect=None*, *xlim=None*, *ylim=None*, *plotstr=None*, *plot=True*, *print_gain=None*, *grid=None*, *ax=None*, ***kwargs*)

Root locus plot

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

Parameters

- **sys** (*LTI object*) – Linear input/output systems (SISO only, for now).
- **kvect** (*list or ndarray, optional*) – List of gains to use in computing diagram.
- **xlim** (*tuple or list, optional*) – Set limits of x axis, normally with tuple (see matplotlib.axes).
- **ylim** (*tuple or list, optional*) – Set limits of y axis, normally with tuple (see matplotlib.axes).
- **plotstr** (`matplotlib.pyplot.plot()` format string, optional) – plotting style specification
- **plot** (*boolean, optional*) – If True (default), plot root locus diagram.
- **print_gain** (*bool*) – If True (default), report mouse clicks when close to the root locus branches, calculate gain, damping and print.
- **grid** (*bool*) – If True plot omega-damping grid. Default is False.
- **ax** (`matplotlib.axes.Axes`) – Axes on which to create root locus plot

Returns

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

3.6.11 control.sisotool

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

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

Parameters

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

Examples

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

3.7 Matrix computations

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

continues on next page

Table 7 – continued from previous page

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

3.7.1 control.care

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

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

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

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

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

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

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

Parameters

- **A** (*2D arrays*) – Input matrices for the Riccati equation
- **B** (*2D arrays*) – Input matrices for the Riccati equation
- **Q** (*2D arrays*) – Input matrices for the Riccati equation
- **R** (*2D arrays, optional*) – Input matrices for generalized Riccati equation
- **S** (*2D arrays, optional*) – Input matrices for generalized Riccati equation
- **E** (*2D arrays, optional*) – Input matrices for generalized Riccati equation

Returns

- **X** (*2D array (or matrix)*) – Solution to the Ricatti equation
- **L** (*1D array*) – Closed loop eigenvalues
- **G** (*2D array (or matrix)*) – Gain matrix

Notes

The return type for 2D arrays depends on the default class set for state space operations. See `use_numpy_matrix()`.

3.7.2 control.dare

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

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

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

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

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

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

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

Parameters

- **A** (*2D arrays*) – Input matrices for the Riccati equation
- **B** (*2D arrays*) – Input matrices for the Riccati equation
- **Q** (*2D arrays*) – Input matrices for the Riccati equation
- **R** (*2D arrays, optional*) – Input matrices for generalized Riccati equation
- **S** (*2D arrays, optional*) – Input matrices for generalized Riccati equation
- **E** (*2D arrays, optional*) – Input matrices for generalized Riccati equation

Returns

- **X** (*2D array (or matrix)*) – Solution to the Ricatti equation
- **L** (*1D array*) – Closed loop eigenvalues
- **G** (*2D array (or matrix)*) – Gain matrix

Notes

The return type for 2D arrays depends on the default class set for state space operations. See *use_numpy_matrix()*.

3.7.3 control.lyap

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

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

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

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

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

$$AX + XQ + C = 0$$

where A and Q are square matrices.

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

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

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

Parameters

- **A** (*2D array*) – Dynamics matrix
- **C** (*2D array, optional*) – If present, solve the Slyvester equation
- **E** (*2D array, optional*) – If present, solve the generalized Laypunov equation

Returns **Q** – Solution to the Lyapunov or Sylvester equation

Return type 2D array (or matrix)

Notes

The return type for 2D arrays depends on the default class set for state space operations. See *use_numpy_matrix()*.

3.7.4 control.dlyap

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

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

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

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

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

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

where A and Q are square matrices.

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

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

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

3.7.5 control.ctrb

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

Controllabilty matrix

Parameters

- **A** (*array_like or string*) – Dynamics and input matrix of the system
- **B** (*array_like or string*) – Dynamics and input matrix of the system

Returns **C** – Controllability matrix

Return type 2D array (or matrix)

Notes

The return type for 2D arrays depends on the default class set for state space operations. See *use_numpy_matrix()*.

Examples

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

3.7.6 control.obsv

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

Observability matrix

Parameters

- **A** (*array_like or string*) – Dynamics and output matrix of the system
- **C** (*array_like or string*) – Dynamics and output matrix of the system

Returns **O** – Observability matrix

Return type 2D array (or matrix)

Notes

The return type for 2D arrays depends on the default class set for state space operations. See *use_numpy_matrix()*.

Examples

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

3.7.7 control.gram

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

Gramian (controllability or observability)

Parameters

- **sys** (StateSpace) – System description
- **type** (*String*) – Type of desired computation. *type* is either 'c' (controllability) or 'o' (observability). To compute the Cholesky factors of Gramians use 'cf' (controllability) or 'of' (observability)

Returns **gram** – Gramian of system

Return type 2D array (or matrix)

Raises

- **ValueError** –

    - if system is not instance of StateSpace class * if *type* is not 'c', 'o', 'cf' or 'of' * if system is unstable (sys.A has eigenvalues not in left half plane)
- **ImportError** – if slycot routine sb03md cannot be found if slycot routine sb03od cannot be found

Notes

The return type for 2D arrays depends on the default class set for state space operations. See *use_numpy_matrix()*.

Examples

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

3.8 Control system synthesis

acker(A, B, poles)	Pole placement using Ackermann method
h2syn(P, nmeas, ncon)	H_2 control synthesis for plant P.
hinfsyn(P, nmeas, ncon)	H_{inf} control synthesis for plant P.
lqr(A, B, Q, R[, N])	Linear quadratic regulator design
lqe(A, G, C, QN, RN, [, N])	Linear quadratic estimator design (Kalman filter) for continuous-time systems.
mixsyn(g[, w1, w2, w3])	Mixed-sensitivity H-infinity synthesis.
place(A, B, p)	Place closed loop eigenvalues

3.8.1 control.acker

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

Pole placement using Ackermann method

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

Parameters

- **A** (*2D arrays*) – State and input matrix of the system
- **B** (*2D arrays*) – State and input matrix of the system
- **poles** (*1D list*) – Desired eigenvalue locations

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

Return type 2D array (or matrix)

Notes

The return type for 2D arrays depends on the default class set for state space operations. See *use_numpy_matrix()*.

3.8.2 control.h2syn

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

H_2 control synthesis for plant P.

Parameters

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

Returns K

Return type controller to stabilize P (State-space sys)

Raises **ImportError** – if slycot routine sb10hd is not loaded

See also:

StateSpace

Examples

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

3.8.3 control.hinfsyn

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

H_{inf} control synthesis for plant P.

Parameters

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

Returns

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

Raises **ImportError** – if slycot routine sb10ad is not loaded

See also:

StateSpace

Examples

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

3.8.4 control.lqr

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

Linear quadratic regulator design

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

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

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

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

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

Parameters

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

Returns

- **K** (*2D array (or matrix)*) – State feedback gains
- **S** (*2D array (or matrix)*) – Solution to Riccati equation
- **E** (*1D array*) – Eigenvalues of the closed loop system

See also:

lqe

Notes

The return type for 2D arrays depends on the default class set for state space operations. See *use_numpy_matrix()*.

Examples

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

3.8.5 control.lqe

`control.lqe`(*A*, *G*, *C*, *QN*, *RN*[, *N*])

Linear quadratic estimator design (Kalman filter) for continuous-time systems. Given the system

$$x = Ax + Bu + Gw$$
$$y = Cx + Du + v$$

with unbiased process noise w and measurement noise v with covariances

$$Eww' = QN, Evv' = RN, Ewv' = NN$$

The lqe() function computes the observer gain matrix L such that the stationary (non-time-varying) Kalman filter

$$x_e = Ax_e + Bu + L(y - Cx_e - Du)$$

produces a state estimate that x_e that minimizes the expected squared error using the sensor measurements y. The noise cross-correlation *NN* is set to zero when omitted.

Parameters

- **A** (*2D array*) – Dynamics and noise input matrices
- **G** (*2D array*) – Dynamics and noise input matrices
- **QN** (*2D array*) – Process and sensor noise covariance matrices
- **RN** (*2D array*) – Process and sensor noise covariance matrices
- **NN** (*2D array, optional*) – Cross covariance matrix

Returns

- **L** (*2D array (or matrix)*) – Kalman estimator gain
- **P** (*2D array (or matrix)*) – Solution to Riccati equation

$$AP + PA^T - (PC^T + GN)R^{-1}(CP + N^T G^T) + GQG^T = 0$$

- **E** (*1D array*) – Eigenvalues of estimator poles eig(A - L C)

Notes

The return type for 2D arrays depends on the default class set for state space operations. See *use_numpy_matrix()*.

Examples

```
>>> K, P, E = lqe(A, G, C, QN, RN)
>>> K, P, E = lqe(A, G, C, QN, RN, NN)
```

See also:

lqr

3.8.6 control.mixsyn

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

Mixed-sensitivity H-infinity synthesis.

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

Parameters

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

Returns

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

See also:

hinfsyn, *augw*

3.8.7 control.place

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

Place closed loop eigenvalues

K = place(A, B, p)

Parameters

- **A** (*2D array*) – Dynamics matrix
- **B** (*2D array*) – Input matrix
- **p** (*1D list*) – Desired eigenvalue locations

Returns **K** – Gain such that A - B K has eigenvalues given in p

Return type 2D array (or matrix)

Notes

Algorithm This is a wrapper function for `scipy.signal.place_poles()`, which implements the Tits and Yang algorithm[1]. It will handle SISO, MISO, and MIMO systems. If you want more control over the algorithm, use `scipy.signal.place_poles()` directly.

Limitations The algorithm will not place poles at the same location more than rank(B) times.

The return type for 2D arrays depends on the default class set for state space operations. See *use_numpy_matrix()*.

References

Examples

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

See also:

place_varga, *acker*

Notes

The return type for 2D arrays depends on the default class set for state space operations. See *use_numpy_matrix()*.

[1] A.L. Tits and Y. Yang, "Globally convergent algorithms for robust pole assignment by state feedback, IEEE Transactions on Automatic Control, Vol. 41, pp. 1432-1452, 1996.

3.9 Model simplification tools

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

3.9.1 control.minreal

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

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

Parameters

- **sys** (`StateSpace or TransferFunction`) – Original system
- **tol** (*real*) – Tolerance
- **verbose** (*bool*) – Print results if True

Returns **rsys** – Cleaned model

Return type *StateSpace* or *TransferFunction*

3.9.2 control.balred

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

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

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

Parameters

- **sys** (`StateSpace`) – Original system to reduce
- **orders** (*integer or array of integer*) – Desired order of reduced order model (if a vector, returns a vector of systems)
- **method** (*string*) – Method of removing states, either `'truncate'` or `'matchdc'`.
- **alpha** (*float*) – Redefines the stability boundary for eigenvalues of the system matrix A. By default for continuous-time systems, alpha <= 0 defines the stability boundary for the real part of A's eigenvalues and for discrete-time systems, 0 <= alpha <= 1 defines the stability boundary for the modulus of A's eigenvalues. See SLICOT routines AB09MD and AB09ND for more information.

Returns **rsys** – A reduced order model or a list of reduced order models if orders is a list.

Return type *StateSpace*

Raises

- **ValueError** – If *method* is not `'truncate'` or `'matchdc'`
- **ImportError** – if slycot routine ab09ad, ab09md, or ab09nd is not found
- **ValueError** – if there are more unstable modes than any value in orders

Examples

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

3.9.3 control.hsvd

`control.hsvd(sys)`

Calculate the Hankel singular values.

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

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

Return type array

See also:

gram

Notes

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

Examples

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

3.9.4 control.modred

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

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

Parameters

- **sys** (`StateSpace`) – Original system to reduce
- **ELIM** (*array*) – Vector of states to eliminate
- **method** (*string*) – Method of removing states in *ELIM*: either `'truncate'` or `'matchdc'`.

Returns **rsys** – A reduced order model

Return type *StateSpace*

Raises **ValueError** – Raised under the following conditions:

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

* if eigenvalues of *sys.A* are not all in left half plane (*sys* must be stable)

Examples

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

3.9.5 control.era

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

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

Note: This function is not implemented yet.

Parameters

- **YY** (*array*) – *nout* x *nin* dimensional impulse-response data
- **m** (*integer*) – Number of rows in Hankel matrix
- **n** (*integer*) – Number of columns in Hankel matrix
- **nin** (*integer*) – Number of input variables
- **nout** (*integer*) – Number of output variables
- **r** (*integer*) – Order of model

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

Return type *StateSpace*

Examples

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

3.9.6 control.markov

`control.markov(Y, U, m=None, transpose=None)`

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

This function computes the Markov parameters for a discrete time system

$$\begin{aligned} x[k+1] &= Ax[k] + Bu[k] \\ y[k] &= Cx[k] + Du[k] \end{aligned}$$

given data for u and y. The algorithm assumes that that C A^k B = 0 for k > m-2 (see[1]). Note that the problem is ill-posed if the length of the input data is less than the desired number of Markov parameters (a warning message is generated in this case).

Parameters

- **Y** (*array_like*) – Output data. If the array is 1D, the system is assumed to be single input. If the array is 2D and transpose=False, the columns of *Y* are taken as time points, otherwise the rows of *Y* are taken as time points.
- **U** (*array_like*) – Input data, arranged in the same way as *Y*.
- **m** (*int, optional*) – Number of Markov parameters to output. Defaults to len(U).
- **transpose** (*bool, optional*) – Assume that input data is transposed relative to the standard *Time series data*. The default value is true for backward compatibility with legacy code.

Returns **H** – First m Markov parameters, [D CB CAB . . .]

Return type ndarray

References

Notes

Currently only works for SISO systems.

This function does not currently comply with the Python Control Library *Time series data* for representation of time series data. Use *transpose=False* to make use of the standard convention (this will be updated in a future release).

Examples

```
>>> T = numpy.linspace(0, 10, 100)
>>> U = numpy.ones((1, 100))
>>> T, Y, _ = forced_response(tf([1], [1, 0.5], True), T, U)
>>> H = markov(Y, U, 3, transpose=False)
```

3.10 Nonlinear system support

find_eqpt(sys, x0[, u0, y0, t, params, iu, . . .])	Find the equilibrium point for an input/output system.
linearize(sys, xeq[, ueq, t, params])	Linearize an input/output system at a given state and input.
input_output_response(sys, T[, U, X0, . . .])	Compute the output response of a system to a given input.
ss2io(*args, **kw)	Create an I/O system from a state space linear system.
tf2io(*args, **kw)	Convert a transfer function into an I/O system
flatsys.point_to_point(sys, x0, u0, xf, uf, Tf)	Compute trajectory between an initial and final conditions.

[1] J.-N. Juang, M. Phan, L. G. Horta, and R. W. Longman, Identification of observer/Kalman filter Markov parameters - Theory and experiments. Journal of Guidance Control and Dynamics, 16(2), 320-329, 2012. http://doi.org/10.2514/3.21006

3.10.1 control.iosys.find_eqpt

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

Find the equilibrium point for an input/output system.

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

Parameters

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

Returns

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

- **result** (`scipy.optimize.OptimizeResult`, optional) – If *return_result* is True, returns the *result* from the `scipy.optimize.root()` function.

3.10.2 control.iosys.linearize

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

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

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

Parameters

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

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

Return type *LinearIOSystem*

3.10.3 control.iosys.input_output_response

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

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

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

Parameters

- **sys** (`InputOutputSystem`) – Input/output system to simulate.
- **T** (*array-like*) – Time steps at which the input is defined; values must be evenly spaced.
- **U** (*array-like or number, optional*) – Input array giving input at each time *T* (default = 0).
- **X0** (*array-like or number, optional*) – Initial condition (default = 0).
- **return_x** (*bool, optional*) – If True, return the values of the state at each time (default = False).
- **squeeze** (*bool, optional*) – If True (default), squeeze unused dimensions out of the output response. In particular, for a single output system, return a vector of shape (nsteps) instead of (nsteps, 1).

Returns

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

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

Raises

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

3.10.4 control.iosys.ss2io

`control.iosys.ss2io(*args, **kw)`

Create an I/O system from a state space linear system.

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

Parameters

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

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

Return type *LinearIOSystem*

3.10.5 control.iosys.tf2io

`control.iosys.tf2io(*args, **kw)`

Convert a transfer function into an I/O system

3.10.6 control.flatsys.point_to_point

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

Compute trajectory between an initial and final conditions.

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

Parameters

- **flatsys** (*FlatSystem object*) – Description of the differentially flat system. This object must define a function flatsys.forward() that takes the system state and produceds the flag of flat outputs and a system flatsys.reverse() that takes the flag of the flat output and prodes the state and input.
- **x0** (*1D arrays*) – Define the desired initial and final conditions for the system. If any of the values are given as None, they are replaced by a vector of zeros of the appropriate dimension.
- **u0** (*1D arrays*) – Define the desired initial and final conditions for the system. If any of the values are given as None, they are replaced by a vector of zeros of the appropriate dimension.
- **xf** (*1D arrays*) – Define the desired initial and final conditions for the system. If any of the values are given as None, they are replaced by a vector of zeros of the appropriate dimension.
- **uf** (*1D arrays*) – Define the desired initial and final conditions for the system. If any of the values are given as None, they are replaced by a vector of zeros of the appropriate dimension.
- **Tf** (*float*) – The final time for the trajectory (corresponding to xf)
- **T0** (*float (optional)*) – The initial time for the trajectory (corresponding to x0). If not specified, its value is taken to be zero.
- **basis** (*BasisFamily object (optional)*) – The basis functions to use for generating the trajectory. If not specified, the PolyFamily basis family will be used, with the minimal number of elements required to find a feasible trajectory (twice the number of system states)

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

Return type SystemTrajectory object

3.11 Utility functions and conversions

augw(g[, w1, w2, w3])	Augment plant for mixed sensitivity problem.
canonical_form(xsys[, form])	Convert a system into canonical form
damp(sys[, doprint])	Compute natural frequency, damping ratio, and poles of a system
db2mag(db)	Convert a gain in decibels (dB) to a magnitude
isctime(sys[, strict])	Check to see if a system is a continuous-time system
isdtime(sys[, strict])	Check to see if a system is a discrete time system
issiso(sys[, strict])	Check to see if a system is single input, single output
issys(obj)	Return True if an object is a system, otherwise False

continues on next page

Table 11 – continued from previous page

mag2db(mag)	Convert a magnitude to decibels (dB)
observable_form(xsys)	Convert a system into observable canonical form
pade(T[, n, numdeg])	Create a linear system that approximates a delay.
reachable_form(xsys)	Convert a system into reachable canonical form
reset_defaults()	Reset configuration values to their default (initial) values.
sample_system(sysc, Ts[, method, alpha, . . .])	Convert a continuous time system to discrete time
ss2tf(sys)	Transform a state space system to a transfer function.
ssdata(sys)	Return state space data objects for a system
tf2ss(sys)	Transform a transfer function to a state space system.
tfdata(sys)	Return transfer function data objects for a system
timebase(sys[, strict])	Return the timebase for an LTI system
timebaseEqual(sys1, sys2)	Check to see if two systems have the same timebase
unwrap(angle[, period])	Unwrap a phase angle to give a continuous curve
use_fbs_defaults()	Use Feedback Systems (FBS) compatible settings.
use_matlab_defaults()	Use MATLAB compatible configuration settings.
use_numpy_matrix([flag, warn])	Turn on/off use of Numpy *matrix* class for state space operations.

3.11.1 control.augw

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

Augment plant for mixed sensitivity problem.

Parameters

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

Returns p

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

Raises ValueError –

- if all weightings are None

See also:

h2syn, *hinfsyn*, *mixsyn*

3.11.2 control.canonical_form

`control.canonical_form(xsys, form='reachable')`

Convert a system into canonical form

Parameters

- **xsys** (*StateSpace object*) – System to be transformed, with state 'x'
- **form** (*String*) –

  Canonical form for transformation. Chosen from:

  - 'reachable' - reachable canonical form
  - 'observable' - observable canonical form
  - 'modal' - modal canonical form

Returns

- **zsys** (*StateSpace object*) – System in desired canonical form, with state 'z'
- **T** (*matrix*) – Coordinate transformation matrix, z = T * x

3.11.3 control.damp

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

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

The function takes 1 or 2 parameters

Parameters

- **sys** (*LTI (StateSpace or TransferFunction)*) – A linear system object
- **doprint** – if true, print table with values

Returns

- **wn** (*array*) – Natural frequencies of the poles
- **damping** (*array*) – Damping values
- **poles** (*array*) – Pole locations
- *Algorithm*
- ———
- *If the system is continuous,* – wn = abs(poles) Z = -real(poles)/poles.
- *If the system is discrete, the discrete poles are mapped to their*
- *equivalent location in the s-plane via* – s = log10(poles)/dt
- *and* – wn = abs(s) Z = -real(s)/wn.

See also:

pole

3.11.4 control.db2mag

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

Convert a gain in decibels (dB) to a magnitude

If A is magnitude,

db = 20 * log10(A)

Parameters **db** (*float or ndarray*) – input value or array of values, given in decibels

Returns **mag** – corresponding magnitudes

Return type float or ndarray

3.11.5 control.isctime

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

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

Parameters

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

3.11.6 control.isdtime

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

Check to see if a system is a discrete time system

Parameters

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

3.11.7 control.issiso

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

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

Parameters

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

3.11.8 control.issys

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

3.11.9 control.mag2db

`control.mag2db(mag)`
Convert a magnitude to decibels (dB)

If A is magnitude,

db = 20 * log10(A)

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

Returns **db** – corresponding values in decibels

Return type float or ndarray

3.11.10 control.observable_form

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

Parameters **xsys** (*StateSpace object*) – System to be transformed, with state *x*

Returns

- **zsys** (*StateSpace object*) – System in observable canonical form, with state *z*
- **T** (*matrix*) – Coordinate transformation: z = T * x

3.11.11 control.pade

`control.pade(T, n=1, numdeg=None)`
Create a linear system that approximates a delay.

Return the numerator and denominator coefficients of the Pade approximation.

Parameters

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

Returns **num, den** – Polynomial coefficients of the delay model, in descending powers of s.

Return type array

Notes

Based on:

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

3.11.12 control.reachable_form

control.**reachable_form**(*xsys*)

Convert a system into reachable canonical form

Parameters **xsys** (*StateSpace object*) – System to be transformed, with state *x*

Returns

- **zsys** (*StateSpace object*) – System in reachable canonical form, with state *z*
- **T** (*matrix*) – Coordinate transformation: z = T * x

3.11.13 control.sample_system

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

Convert a continuous time system to discrete time

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

Parameters

- **sysc** (*linsys*) – Continuous time system to be converted
- **Ts** (*real*) – Sampling period
- **method** (*string*) – Method to use for conversion: 'matched', 'tustin', 'zoh' (default)
- **prewarp_frequency** (*float within [0, infinity)*) – The frequency [rad/s] at which to match with the input continuous- time system's magnitude and phase

Returns **sysd** – Discrete time system, with sampling rate Ts

Return type linsys

Notes

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

Examples

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

3.11.14 control.ss2tf

`control.ss2tf(sys)`

Transform a state space system to a transfer function.

The function accepts either 1 or 4 parameters:

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

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

For details see: *ss()*

Parameters

- **sys** (StateSpace) – A linear system
- **A** (*array_like or string*) – System matrix
- **B** (*array_like or string*) – Control matrix
- **C** (*array_like or string*) – Output matrix
- **D** (*array_like or string*) – Feedthrough matrix

Returns **out** – New linear system in transfer function form

Return type *TransferFunction*

Raises

- **ValueError** – if matrix sizes are not self-consistent, or if an invalid number of arguments is passed in
- **TypeError** – if *sys* is not a StateSpace object

See also:

tf, *ss*, *tf2ss*

Examples

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

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

3.11.15 control.ssdata

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

Return state space data objects for a system

Parameters **sys** (*LTI (StateSpace, or TransferFunction)*) – LTI system whose data will be returned

Returns **(A, B, C, D)** – State space data for the system

Return type list of matrices

3.11.16 control.tf2ss

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

Transform a transfer function to a state space system.

The function accepts either 1 or 2 parameters:

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

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

For details see: *tf()*

Parameters

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

Returns **out** – New linear system in state space form

Return type *StateSpace*

Raises

- **ValueError** – if *num* and *den* have invalid or unequal dimensions, or if an invalid number of arguments is passed in
- **TypeError** – if *num* or *den* are of incorrect type, or if sys is not a TransferFunction object

See also:

ss, *tf*, *ss2tf*

Examples

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

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

3.11.17 control.tfdata

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

Return transfer function data objects for a system

Parameters sys (`LTI (StateSpace, or TransferFunction)`) – LTI system whose data will be returned

Returns (num, den) – Transfer function coefficients (SISO only)

Return type numerator and denominator arrays

3.11.18 control.timebase

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

Return the timebase for an LTI system

dt = timebase(sys)

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

3.11.19 control.timebaseEqual

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

Check to see if two systems have the same timebase

timebaseEqual(sys1, sys2)

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

3.11.20 control.unwrap

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

Unwrap a phase angle to give a continuous curve

Parameters

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

Returns angle_out – Output array, with jumps of period/2 eliminated

Return type array_like

Examples

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

CHAPTER FOUR

CONTROL SYSTEM CLASSES

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

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

4.1 control.TransferFunction

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

A class for representing transfer functions

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

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

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

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

Discrete-time transfer functions are implemented by using the 'dt' instance variable and setting it to something other than 'None'. If 'dt' has a non-zero value, then it must match whenever two transfer functions are combined. If 'dt' is set to True, the system will be treated as a discrete time system with unspecified sampling time. The default value of 'dt' is None and can be changed by changing the value of `control.config.defaults['xferfcn.default_dt']`.

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

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

__init__(**args*)

TransferFunction(num, den[, dt])

Construct a transfer function.

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

Methods

__init__(*args)	TransferFunction(num, den[, dt])
damp()	Natural frequency, damping ratio of system poles
dcgain()	Return the zero-frequency (or DC) gain
evalfr(omega)	Evaluate a transfer function at a single angular frequency.
feedback([other, sign])	Feedback interconnection between two LTI objects.
freqresp(omega)	Evaluate the transfer function at a list of angular frequencies.
horner(s)	Evaluate the systems's transfer function for a complex variable
is_static_gain()	returns True if and only if all of the numerator and denominator polynomials of the (possibly MIMO) transfer function are zeroth order, that is, if the system has no dynamics.
isctime([strict])	Check to see if a system is a continuous-time system
isdtime([strict])	Check to see if a system is a discrete-time system
issiso()	Check to see if a system is single input, single output
minreal([tol])	Remove cancelling pole/zero pairs from a transfer function
pole()	Compute the poles of a transfer function.
returnScipySignalLTI()	Return a list of a list of `scipy.signal.lti` objects.
sample(Ts[, method, alpha, prewarp_frequency])	Convert a continuous-time system to discrete time
zero()	Compute the zeros of a transfer function.

Attributes

s
z

damp()

Natural frequency, damping ratio of system poles

Returns

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

dcgain()

Return the zero-frequency (or DC) gain

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

Returns **gain** – The zero-frequency gain

Return type ndarray

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

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

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

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

Reports the frequency response of the system,

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

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

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

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

Returns

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

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

Returns a matrix of values evaluated at complex variable s.

is_static_gain()
returns True if and only if all of the numerator and denominator polynomials of the (possibly MIMO) transfer function are zeroth order, that is, if the system has no dynamics.

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

Parameters

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

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

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

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

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

pole ()
Compute the poles of a transfer function.

returnScipySignalLTI ()
Return a list of a list of `scipy.signal.lti` objects.

For instance,

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

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

sample (*Ts*, *method='zoh'*, *alpha=None*, *prewarp_frequency=None*)
Convert a continuous-time system to discrete time

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

Parameters

- **Ts** (*float*) – Sampling period
- **method** (*{"gbt", "bilinear", "euler", "backward_diff",*) – "zoh", "matched"} Method to use for sampling:
  - gbt: generalized bilinear transformation
  - bilinear: Tustin's approximation ("gbt" with alpha=0.5)
  - euler: Euler (or forward difference) method ("gbt" with alpha=0)
  - backward_diff: Backwards difference ("gbt" with alpha=1.0)
  - zoh: zero-order hold (default)
- **alpha** (*float within [0, 1]*) – The generalized bilinear transformation weighting parameter, which should only be specified with method="gbt", and is ignored otherwise.
- **prewarp_frequency** (*float within [0, infinity)*) – The frequency [rad/s] at which to match with the input continuous- time system's magnitude and phase (the gain=1 crossover frequency, for example). Should only be specified with method='bilinear' or 'gbt' with alpha=0.5 and ignored otherwise.

Returns **sysd** – Discrete time system, with sampling rate Ts

Return type StateSpace system

Notes

1. Available only for SISO systems
2. Uses `scipy.signal.cont2discrete()`

Examples

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

zero()

Compute the zeros of a transfer function.

4.2 control.StateSpace

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

A class for representing state-space models

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

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

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

The main data members are the A, B, C, and D matrices. The class also keeps track of the number of states (i.e., the size of A). The data format used to store state space matrices is set using the value of *config.defaults['use_numpy_matrix']*. If True (default), the state space elements are stored as *numpy.matrix* objects; otherwise they are *numpy.ndarray* objects. The `use_numpy_matrix()` function can be used to set the storage type.

Discrete-time state space system are implemented by using the 'dt' instance variable and setting it to the sampling period. If 'dt' is not None, then it must match whenever two state space systems are combined. Setting dt = 0 specifies a continuous system, while leaving dt = None means the system timebase is not specified. If 'dt' is set to True, the system will be treated as a discrete time system with unspecified sampling time. The default value of 'dt' is None and can be changed by changing the value of `control.config.defaults['statesp.default_dt']`.

__init__(**args, **kw*)

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

Construct a state space object.

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

Methods

`__init__`(*args, **kw)	StateSpace(A, B, C, D[, dt])
`append`(other)	Append a second model to the present model.
`damp`()	Natural frequency, damping ratio of system poles
`dcgain`()	Return the zero-frequency gain
`evalfr`(omega)	Evaluate a SS system's transfer function at a single frequency.
`feedback`([other, sign])	Feedback interconnection between two LTI systems.
`freqresp`(omega)	Evaluate the system's transfer function at a list of frequencies

continues on next page

Table 4 – continued from previous page

horner(s)	Evaluate the systems's transfer function for a complex variable
is_static_gain()	True if and only if the system has no dynamics, that is, if A and B are zero.
isctime([strict])	Check to see if a system is a continuous-time system
isdtime([strict])	Check to see if a system is a discrete-time system
issiso()	Check to see if a system is single input, single output
lft(other[, nu, ny])	Return the Linear Fractional Transformation.
minreal([tol])	Calculate a minimal realization, removes unobservable and uncontrollable states
pole()	Compute the poles of a state space system.
returnScipySignalLTI()	Return a list of a list of `scipy.signal.lti` objects.
sample(Ts[, method, alpha, prewarp_frequency])	Convert a continuous time system to discrete time
zero()	Compute the zeros of a state space system.

append(*other*)

Append a second model to the present model.

The second model is converted to state-space if necessary, inputs and outputs are appended and their order is preserved

damp()

Natural frequency, damping ratio of system poles

Returns

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

dcgain()

Return the zero-frequency gain

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

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

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

Return type ndarray

evalfr(*omega*)

Evaluate a SS system's transfer function at a single frequency.

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

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

Feedback interconnection between two LTI systems.

freqresp(*omega*)

Evaluate the system's transfer function at a list of frequencies

Reports the frequency response of the system,

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

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

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

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

Returns

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

horner(*s*)

Evaluate the systems's transfer function for a complex variable

Returns a matrix of values evaluated at complex variable s.

is_static_gain()

True if and only if the system has no dynamics, that is, if A and B are zero.

isctime(*strict=False*)

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

Parameters

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

isdtime(*strict=False*)

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

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

issiso()

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

lft(*other*, *nu=- 1*, *ny=- 1*)

Return the Linear Fractional Transformation.

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

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

Parameters

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

minreal(*tol=0.0*)

Calculate a minimal realization, removes unobservable and uncontrollable states

pole()

Compute the poles of a state space system.

returnScipySignalLTI()

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

For instance,

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

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

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

Convert a continuous time system to discrete time

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

Parameters

- **Ts** (*float*) – Sampling period
- **method** (*{"gbt", "bilinear", "euler", "backward_diff", "zoh"}*) – Which method to use:
  - gbt: generalized bilinear transformation
  - bilinear: Tustin's approximation ("gbt" with alpha=0.5)
  - euler: Euler (or forward differencing) method ("gbt" with alpha=0)
  - backward_diff: Backwards differencing ("gbt" with alpha=1.0)
  - zoh: zero-order hold (default)
- **alpha** (*float within [0, 1]*) – The generalized bilinear transformation weighting parameter, which should only be specified with method="gbt", and is ignored otherwise
- **prewarp_frequency** (*float within [0, infinity)*) – The frequency [rad/s] at which to match with the input continuous- time system's magnitude and phase (the gain=1 crossover frequency, for example). Should only be specified with method='bilinear' or 'gbt' with alpha=0.5 and ignored otherwise.

Returns **sysd** – Discrete time system, with sampling rate Ts

Return type *StateSpace*

Notes

Uses `scipy.signal.cont2discrete()`

Examples

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

zero()

Compute the zeros of a state space system.

4.3 control.FrequencyResponseData

class control.**FrequencyResponseData**(*d*, *w*)

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

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

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

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

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

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

Construct an FRD object.

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

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

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

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

Methods

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

Attributes

epsw

damp()

Natural frequency, damping ratio of system poles

Returns

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

dcgain()

Return the zero-frequency gain

eval(*omega*)

Evaluate a transfer function at a single angular frequency.

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

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

evalfr(*omega*)

Evaluate a transfer function at a single angular frequency.

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

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

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

Feedback interconnection between two FRD objects.

freqresp(*omega*)

Evaluate the frequency response at a list of angular frequencies.

Reports the value of the magnitude, phase, and angular frequency of the requency response evaluated at omega, where omega is a list of angular frequencies, and is a sorted version of the input omega.

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

Returns

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

isctime(*strict=False*)

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

Parameters

- **sys** (*LTI system*) – System to be checked

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

isdtime (*strict=False*)

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

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

issiso ()

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

4.4 control.iosys.InputOutputSystem

class control.iosys.**InputOutputSystem** (*inputs=None*, *outputs=None*, *states=None*, *params={}*, *dt=None*, *name=None*)

A class for representing input/output systems.

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

Parameters

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

ninputs, noutputs, nstates

Number of input, output and state variables

Type int

input_index, output_index, state_index

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

Type dict

dt

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

Type None, True or float

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

Type dict, optional

name
System name (used for specifying signals)

Type string, optional

Notes

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

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

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

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

Parameters

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

Returns Input/output system object

Return type *InputOutputSystem*

Methods

`__init__`([inputs, outputs, states, params, . . .])	Create an input/output system.
`copy`([newname])	Make a copy of an input/output system.
`feedback`([other, sign, params])	Feedback interconnection between two input/output systems
`find_input`(name)	Find the index for an input given its name (*None* if not found)
`find_output`(name)	Find the index for an output given its name (*None* if not found)
`find_state`(name)	Find the index for a state given its name (*None* if not found)
`linearize`(x0, u0[, t, params, eps])	Linearize an input/output system at a given state and input.
`name_or_default`([name])	
`set_inputs`(inputs[, prefix])	Set the number/names of the system inputs.
`set_outputs`(outputs[, prefix])	Set the number/names of the system outputs.
`set_states`(states[, prefix])	Set the number/names of the system states.

Attributes

`idCounter`

copy(*newname=None*)

Make a copy of an input/output system.

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

Feedback interconnection between two input/output systems

Parameters

- **sys1** (`InputOutputSystem`) – The primary process.
- **sys2** (`InputOutputSystem`) – The feedback process (often a feedback controller).
- **sign** (`scalar, optional`) – The sign of feedback. *sign* = -1 indicates negative feedback, and *sign* = 1 indicates positive feedback. *sign* is an optional argument; it assumes a value of -1 if not specified.

Returns **out**

Return type *InputOutputSystem*

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

find_input(*name*)

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

find_output(*name*)

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

find_state(*name*)

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

linearize(*x0*, *u0*, *t=0*, *params={}*, *eps=1e-06*)

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

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

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

Parameters

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

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

Parameters

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

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

Parameters

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

4.5 Input/output system subclasses

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

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

4.5.1 control.iosys.LinearIOSystem

class control.iosys.**LinearIOSystem**(*linsys*, *inputs=None*, *outputs=None*, *states=None*, *name=None*)

Input/output representation of a linear (state space) system.

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

__init__(*linsys*, *inputs=None*, *outputs=None*, *states=None*, *name=None*)

Create an I/O system from a state space linear system.

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

Parameters

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

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

Return type *LinearIOSystem*

Methods

__init__(linsys[, inputs, outputs, states, name])	Create an I/O system from a state space linear system.
append(other)	Append a second model to the present model.
copy([newname])	Make a copy of an input/output system.
damp()	Natural frequency, damping ratio of system poles
dcgain()	Return the zero-frequency gain
evalfr(omega)	Evaluate a SS system's transfer function at a single frequency.
feedback([other, sign, params])	Feedback interconnection between two input/output systems

continues on next page

Table 10 – continued from previous page

find_input(name)	Find the index for an input given its name (*None* if not found)
find_output(name)	Find the index for an output given its name (*None* if not found)
find_state(name)	Find the index for a state given its name (*None* if not found)
freqresp(omega)	Evaluate the system's transfer function at a list of frequencies
horner(s)	Evaluate the systems's transfer function for a complex variable
is_static_gain()	True if and only if the system has no dynamics, that is, if A and B are zero.
isctime([strict])	Check to see if a system is a continuous-time system
isdtime([strict])	Check to see if a system is a discrete-time system
issiso()	Check to see if a system is single input, single output
lft(other[, nu, ny])	Return the Linear Fractional Transformation.
linearize(x0, u0[, t, params, eps])	Linearize an input/output system at a given state and input.
minreal([tol])	Calculate a minimal realization, removes unobservable and uncontrollable states
`name_or_default`([name])	
pole()	Compute the poles of a state space system.
returnScipySignalLTI()	Return a list of a list of `scipy.signal.lti` objects.
sample(Ts[, method, alpha, prewarp_frequency])	Convert a continuous time system to discrete time
set_inputs(inputs[, prefix])	Set the number/names of the system inputs.
set_outputs(outputs[, prefix])	Set the number/names of the system outputs.
set_states(states[, prefix])	Set the number/names of the system states.
zero()	Compute the zeros of a state space system.

Attributes

`idCounter`

append(*other*)
Append a second model to the present model.

The second model is converted to state-space if necessary, inputs and outputs are appended and their order is preserved

copy(*newname=None*)
Make a copy of an input/output system.

damp()
Natural frequency, damping ratio of system poles

Returns

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

dcgain()
Return the zero-frequency gain

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

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

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

Return type ndarray

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

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

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

Parameters

- **sys1** (`InputOutputSystem`) – The primary process.
- **sys2** (`InputOutputSystem`) – The feedback process (often a feedback controller).
- **sign** (`scalar, optional`) – The sign of feedback. *sign* = -1 indicates negative feedback, and *sign* = 1 indicates positive feedback. *sign* is an optional argument; it assumes a value of -1 if not specified.

Returns out

Return type *InputOutputSystem*

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

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

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

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

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

Reports the frequency response of the system,

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

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

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

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

Returns

- **mag** (*(self.outputs, self.inputs, len(omega)) ndarray*) – The magnitude (absolute value, not dB or log10) of the system frequency response.

- **phase** (*(self.outputs, self.inputs, len(omega)) ndarray*) – The wrapped phase in radians of the system frequency response.
- **omega** (*ndarray*) – The list of sorted frequencies at which the response was evaluated.

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

Returns a matrix of values evaluated at complex variable s.

is_static_gain()
True if and only if the system has no dynamics, that is, if A and B are zero.

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

Parameters

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

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

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

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

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

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

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

Parameters

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

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

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

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

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

returnScipySignalLTI()
Return a list of a list of `scipy.signal.lti` objects.

For instance,

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

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

sample (*Ts*, *method='zoh'*, *alpha=None*, *prewarp_frequency=None*)
Convert a continuous time system to discrete time

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

Parameters

- **Ts** (*float*) – Sampling period
- **method** (*{"gbt", "bilinear", "euler", "backward_diff", "zoh"}*) – Which method to use:
  - gbt: generalized bilinear transformation
  - bilinear: Tustin's approximation ("gbt" with alpha=0.5)
  - euler: Euler (or forward differencing) method ("gbt" with alpha=0)
  - backward_diff: Backwards differencing ("gbt" with alpha=1.0)
  - zoh: zero-order hold (default)
- **alpha** (*float within [0, 1]*) – The generalized bilinear transformation weighting parameter, which should only be specified with method="gbt", and is ignored otherwise
- **prewarp_frequency** (*float within [0, infinity)*) – The frequency [rad/s] at which to match with the input continuous- time system's magnitude and phase (the gain=1 crossover frequency, for example). Should only be specified with method='bilinear' or 'gbt' with alpha=0.5 and ignored otherwise.

Returns **sysd** – Discrete time system, with sampling rate Ts

Return type *StateSpace*

Notes

Uses `scipy.signal.cont2discrete()`

Examples

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

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

Parameters

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

- **prefix** (*string, optional*) – If *inputs* is an integer, create the names of the states using the given prefix (default = 'u'). The names of the input will be of the form *prefix[i]*.

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

Set the number/names of the system outputs.

Parameters

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

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

Set the number/names of the system states.

Parameters

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

zero()

Compute the zeros of a state space system.

4.5.2 control.iosys.NonlinearIOSystem

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

Nonlinear I/O system.

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

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

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

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

Parameters

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

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

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

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

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

- **inputs** (*int, list of str or None, optional*) – Description of the system inputs. This can be given as an integer count or as a list of strings that name the individual signals. If an integer count is specified, the names of the signal will be of the form *s[i]* (where *s* is one of *u*, *y*, or *x*). If this parameter is not given or given as *None*, the relevant quantity will be determined when possible based on other information provided to functions using the system.
- **outputs** (*int, list of str or None, optional*) – Description of the system outputs. Same format as *inputs*.
- **states** (*int, list of str, or None, optional*) – Description of the system states. Same format as *inputs*.
- **params** (*dict, optional*) – Parameter values for the systems. Passed to the evaluation functions for the system as default values, overriding internal defaults.
- **dt** (*timebase, optional*) – The timebase for the system, used to specify whether the system is operating in continuous or discrete time. It can have the following values:
  - dt = None No timebase specified
  - dt = 0 Continuous time system
  - dt > 0 Discrete time system with sampling time dt
  - dt = True Discrete time with unspecified sampling time
- **name** (*string, optional*) – System name (used for specifying signals). If unspecified, a generic name <sys[id]> is generated with a unique integer id.

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

Return type *NonlinearIOSystem*

Methods

`__init__`(updfcn[, outfcn, inputs, outputs, . . .])	Create a nonlinear I/O system given update and output functions.
`copy`([newname])	Make a copy of an input/output system.
`feedback`([other, sign, params])	Feedback interconnection between two input/output systems
`find_input`(name)	Find the index for an input given its name (*None* if not found)
`find_output`(name)	Find the index for an output given its name (*None* if not found)
`find_state`(name)	Find the index for a state given its name (*None* if not found)
`linearize`(x0, u0[, t, params, eps])	Linearize an input/output system at a given state and input.
`name_or_default`([name])	
`set_inputs`(inputs[, prefix])	Set the number/names of the system inputs.
`set_outputs`(outputs[, prefix])	Set the number/names of the system outputs.
`set_states`(states[, prefix])	Set the number/names of the system states.

Attributes

`idCounter`

copy (*newname=None*)
Make a copy of an input/output system.

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

Parameters

- **sys1** (InputOutputSystem) – The primary process.
- **sys2** (InputOutputSystem) – The feedback process (often a feedback controller).
- **sign** (*scalar, optional*) – The sign of feedback. *sign* = -1 indicates negative feedback, and *sign* = 1 indicates positive feedback. *sign* is an optional argument; it assumes a value of -1 if not specified.

Returns out

Return type *InputOutputSystem*

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

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

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

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

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

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

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

Parameters

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

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

Parameters

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

- **prefix** (*string, optional*) – If *outputs* is an integer, create the names of the states using the given prefix (default = 'y'). The names of the input will be of the form *prefix[i]*.

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

Parameters

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

4.5.3 control.iosys.InterconnectedSystem

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

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

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

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

Parameters

- **syslist** (*array_like of InputOutputSystems*) – The list of input/output systems to be connected
- **connections** (*list of tuple of connection specifications, optional*) – Description of the internal connections between the subsystems.

  [connection1, connection2, . . .]

  Each connection is a tuple that describes an input to one of the subsystems. The entries are of the form:

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

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

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

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

- **inplist** (*List of tuple of input specifications, optional*) – List of specifications for how the inputs for the overall system are mapped to the subsystem inputs. The input specification is similar to the form defined in the connection specification, except that connections do not specify an input-spec, since these are the system inputs. The entries are thus of the form:

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

  Each system input is added to the input for the listed subsystem.

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

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

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

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

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

- **states** (*int, list of str, or None, optional*) – Description of the system states. Same format as *inputs*, except the state names will be of the form '<subsys_name>.<state_name>', for each subsys in syslist and each state_name of each subsys.

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

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

  – dt = None No timebase specified

  – dt = 0 Continuous time system

  – dt > 0 Discrete time system with sampling time dt

  – dt = True Discrete time with unspecified sampling time

- **name** (*string, optional*) – System name (used for specifying signals). If unspecified, a generic name <sys[id]> is generated with a unique integer id.

Methods

`__init__`(syslist[, connections, inplist, . . .])	Create an I/O system from a list of systems + connection info.
`copy`([newname])	Make a copy of an input/output system.
`feedback`([other, sign, params])	Feedback interconnection between two input/output systems
`find_input`(name)	Find the index for an input given its name (*None* if not found)
`find_output`(name)	Find the index for an output given its name (*None* if not found)
`find_state`(name)	Find the index for a state given its name (*None* if not found)
`linearize`(x0, u0[, t, params, eps])	Linearize an input/output system at a given state and input.
`name_or_default`([name])	
`set_connect_map`(connect_map)	Set the connection map for an interconnected I/O system.
`set_input_map`(input_map)	Set the input map for an interconnected I/O system.
`set_inputs`(inputs[, prefix])	Set the number/names of the system inputs.
`set_output_map`(output_map)	Set the output map for an interconnected I/O system.
`set_outputs`(outputs[, prefix])	Set the number/names of the system outputs.
`set_states`(states[, prefix])	Set the number/names of the system states.

Attributes

`idCounter`

copy(*newname=None*)
Make a copy of an input/output system.

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

Parameters

- **sys1** (`InputOutputSystem`) – The primary process.
- **sys2** (`InputOutputSystem`) – The feedback process (often a feedback controller).
- **sign** (`scalar, optional`) – The sign of feedback. *sign* = -1 indicates negative feedback, and *sign* = 1 indicates positive feedback. *sign* is an optional argument; it assumes a value of -1 if not specified.

Returns out

Return type *InputOutputSystem*

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

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

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

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

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

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

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

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

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

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

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

Parameters

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

set_output_map(*output_map*)
Set the output map for an interconnected I/O system.

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

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

Parameters

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

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

Parameters

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

- **`prefix`** (*`string, optional`*) – If *states* is an integer, create the names of the states using the given prefix (default = 'x'). The names of the input will be of the form *prefix[i]*.

CHAPTER

FIVE

MATLAB COMPATIBILITY MODULE

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

5.1 Creating linear models

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

5.1.1 control.matlab.tf

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

Create a transfer function system. Can create MIMO systems.

The function accepts either 1, 2, or 3 parameters:

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

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

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

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

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

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

Parameters

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

- **den** (*array_like, or list of list of array_like*) – Polynomial coefficients of the denominator

Returns **out** – The new linear system

Return type `TransferFunction`

Raises

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

See also:

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

Notes

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

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

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

Examples

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

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

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

5.1.2 control.matlab.ss

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

Create a state space system.

The function accepts either 1, 4 or 5 parameters:

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

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

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

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

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

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

Parameters

- **sys** (`StateSpace or TransferFunction`) – A linear system
- **A** (*array_like or string*) – System matrix
- **B** (*array_like or string*) – Control matrix
- **C** (*array_like or string*) – Output matrix
- **D** (*array_like or string*) – Feed forward matrix
- **dt** (*If present, specifies the sampling period and a discrete time*) – system is created

Returns **out** – The new linear system

Return type `StateSpace`

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

See also:

`StateSpace`, *tf*, *ss2tf*, *tf2ss*

Examples

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

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

5.1.3 control.matlab.frd

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

Construct a frequency response data model

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

This function can be called in different ways:

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

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

Parameters

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

Returns sys – New frequency response system

Return type FRD

See also:

FRD, *ss*, *tf*

5.1.4 control.matlab.rss

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

Create a stable *continuous* random state space object.

Parameters

- **states** (*integer*) – Number of state variables
- **inputs** (*integer*) – Number of system inputs
- **outputs** (*integer*) – Number of system outputs

Returns sys – The randomly created linear system

Return type *StateSpace*

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

See also:

drss

Notes

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

5.1.5 control.matlab.drss

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

Create a stable *discrete* random state space object.

Parameters

- **states** (*integer*) – Number of state variables
- **inputs** (*integer*) – Number of system inputs
- **outputs** (*integer*) – Number of system outputs

Returns sys – The randomly created linear system

Return type *StateSpace*

Raises **`ValueError`** – if any input is not a positive integer

See also:

`rss`

Notes

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

5.2 Utility functions and conversions

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

5.2.1 control.matlab.mag2db

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

Convert a magnitude to decibels (dB)

If A is magnitude,

db = 20 * log10(A)

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

Returns **db** – corresponding values in decibels

Return type float or ndarray

5.2.2 control.matlab.db2mag

`control.matlab.`**`db2mag`**(*db*)

Convert a gain in decibels (dB) to a magnitude

If A is magnitude,

db = 20 * log10(A)

Parameters **`db`** (*`float or ndarray`*) – input value or array of values, given in decibels

Returns **mag** – corresponding magnitudes

Return type float or ndarray

5.2.3 control.matlab.c2d

control.matlab.**c2d**(*sysc*, *Ts*, *method='zoh'*, *prewarp_frequency=None*)

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
- **prewarp_frequency** (*float within [0, infinity)*) – The frequency [rad/s] at which to match with the input continuous- time system's magnitude and phase

5.2.4 control.matlab.ss2tf

control.matlab.**ss2tf**(*sys*)

Transform a state space system to a transfer function.

The function accepts either 1 or 4 parameters:

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

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

For details see: *ss()*

Parameters

- **sys** (StateSpace) – A linear system
- **A** (*array_like or string*) – System matrix
- **B** (*array_like or string*) – Control matrix
- **C** (*array_like or string*) – Output matrix
- **D** (*array_like or string*) – Feedthrough matrix

Returns **out** – New linear system in transfer function form

Return type *TransferFunction*

Raises

- **ValueError** – if matrix sizes are not self-consistent, or if an invalid number of arguments is passed in
- **TypeError** – if *sys* is not a StateSpace object

See also:

tf, *ss*, *tf2ss*

Examples

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

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

5.2.5 control.matlab.tf2ss

`control.matlab.tf2ss(sys)`

Transform a transfer function to a state space system.

The function accepts either 1 or 2 parameters:

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

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

For details see: `tf()`

Parameters

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

Returns **out** – New linear system in state space form

Return type *StateSpace*

Raises

- **ValueError** – if *num* and *den* have invalid or unequal dimensions, or if an invalid number of arguments is passed in
- **TypeError** – if *num* or *den* are of incorrect type, or if sys is not a TransferFunction object

See also:

ss, *tf*, *ss2tf*

Examples

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

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

5.2.6 control.matlab.tfdata

`control.matlab.tfdata(sys)`

Return transfer function data objects for a system

Parameters **sys** (*LTI (StateSpace, or TransferFunction)*) – LTI system whose data will be returned

Returns **(num, den)** – Transfer function coefficients (SISO only)

Return type numerator and denominator arrays

5.3 System interconnections

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

5.3.1 control.matlab.series

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

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

Parameters

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

Returns **out**

Return type scalar, *StateSpace*, or *TransferFunction*

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

See also:

parallel, *feedback*

Notes

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

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

Examples

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

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

5.3.2 control.matlab.parallel

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

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

Parameters

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

Returns out

Return type scalar, *StateSpace*, or *TransferFunction*

Raises `ValueError` – if *sys1* and *sys2* do not have the same numbers of inputs and outputs

See also:

series, *feedback*

Notes

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

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

Examples

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

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

5.3.3 control.matlab.feedback

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

Feedback interconnection between two I/O systems.

Parameters

- **sys1** (*scalar, StateSpace, TransferFunction, FRD*) – The primary process.
- **sys2** (*scalar, StateSpace, TransferFunction, FRD*) – The feedback process (often a feedback controller).
- **sign** (*scalar*) – The sign of feedback. *sign* = -1 indicates negative feedback, and *sign* = 1 indicates positive feedback. *sign* is an optional argument; it assumes a value of -1 if not specified.

Returns out

Return type *StateSpace* or *TransferFunction*

Raises

- **ValueError** – if *sys1* does not have as many inputs as *sys2* has outputs, or if *sys2* does not have as many inputs as *sys1* has outputs
- **NotImplementedError** – if an attempt is made to perform a feedback on a MIMO TransferFunction object

See also:

series, *parallel*

Notes

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

5.3.4 control.matlab.negate

`control.matlab.negate(sys)`

Return the negative of a system.

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

Returns out

Return type *StateSpace* or *TransferFunction*

Notes

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

Examples

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

5.3.5 control.matlab.connect

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

Index-based interconnection of an LTI system.

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

NOTE: Inputs and outputs are indexed starting at 1 and negative values correspond to a negative feedback interconnection.

Parameters

- **sys** (*StateSpace Transferfunction*) – System to be connected
- **Q** (*2D array*) – Interconnection matrix. First column gives the input to be connected. The second column gives the index of an output that is to be fed into that input. Each additional column gives the index of an additional input that may be optionally added to that input. Negative values mean the feedback is negative. A zero value is ignored. Inputs and outputs are indexed starting at 1 to communicate sign information.
- **inputv** (*1D array*) – list of final external inputs, indexed starting at 1
- **outputv** (*1D array*) – list of final external outputs, indexed starting at 1

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

Return type LTI system

Examples

```
>>> sys1 = ss([[1., -2], [3., -4]], [[5.], [7]], [[6, 8]], [[9.]])
>>> sys2 = ss([[-1.]], [[1.]], [[1.]], [[0.]])
>>> sys = append(sys1, sys2)
>>> Q = [[1, 2], [2, -1]]  # negative feedback interconnection
>>> sysc = connect(sys, Q, [2], [1, 2])
```

5.3.6 control.matlab.append

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

Group models by appending their inputs and outputs

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

Parameters

- **sys1** (StateSpace *or* Transferfunction) – LTI systems to combine
- **sys2** (StateSpace *or* Transferfunction) – LTI systems to combine
- **..** (StateSpace *or* Transferfunction) – LTI systems to combine
- **sysn** (StateSpace *or* Transferfunction) – LTI systems to combine

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

Return type LTI system

Examples

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

5.4 System gain and dynamics

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

5.4.1 control.matlab.dcgain

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

Compute the gain of the system in steady state.

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

Parameters

- **A** (*array-like*) – A linear system in state space form.
- **B** (*array-like*) – A linear system in state space form.
- **C** (*array-like*) – A linear system in state space form.
- **D** (*array-like*) – A linear system in state space form.

- **Z** (*array-like, array-like, number*) – A linear system in zero, pole, gain form.
- **P** (*array-like, array-like, number*) – A linear system in zero, pole, gain form.
- **k** (*array-like, array-like, number*) – A linear system in zero, pole, gain form.
- **num** (*array-like*) – A linear system in transfer function form.
- **den** (*array-like*) – A linear system in transfer function form.
- **sys** (*LTI (StateSpace or TransferFunction)*) – A linear system object.

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

Return type ndarray

Notes

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

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

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

5.4.2 control.matlab.pole

`control.matlab.pole(sys)`

Compute system poles.

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

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

Return type ndarray

Raises `NotImplementedError` – when called on a TransferFunction object

See also:

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

5.4.3 control.matlab.zero

`control.matlab.zero(sys)`

Compute system zeros.

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

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

Return type ndarray

Raises `NotImplementedError` – when called on a MIMO system

See also:

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

5.4.4 control.matlab.damp

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

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

The function takes 1 or 2 parameters

Parameters

- **sys** (*LTI (StateSpace or TransferFunction)*) – A linear system object
- **doprint** – if true, print table with values

Returns

- **wn** (*array*) – Natural frequencies of the poles
- **damping** (*array*) – Damping values
- **poles** (*array*) – Pole locations
- *Algorithm*
- ———
- *If the system is continuous,* – wn = abs(poles) Z = -real(poles)/poles.
- *If the system is discrete, the discrete poles are mapped to their*
- *equivalent location in the s-plane via* – s = log10(poles)/dt
- *and* – wn = abs(s) Z = -real(s)/wn.

See also:

pole

5.4.5 control.matlab.pzmap

control.matlab.**pzmap**(*sys*, *plot=None*, *grid=None*, *title='Pole Zero Map'*, ***kwargs*)

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, optional*) – If True a graph is generated with Matplotlib, otherwise the poles and zeros are only computed and returned.
- **grid** (*boolean (default = False)*) – If True plot omega-damping grid.

Returns

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

5.5 Time-domain analysis

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

5.5.1 control.matlab.step

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

Step response of a linear system

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

Parameters

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

  Numbers are converted to constant arrays with the correct shape.
- **input** (*int*) – Index of the input that will be used in this simulation.
- **output** (*int*) – If given, index of the output that is returned by this simulation.

Returns

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

See also:

lsim, *initial*, *impulse*

Examples

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

5.5.2 control.matlab.impulse

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

Impulse response of a linear system

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

Parameters

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

  Numbers are converted to constant arrays with the correct shape.
- **input** (*int*) – Index of the input that will be used in this simulation.
- **output** (*int*) – Index of the output that will be used in this simulation.

Returns

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

See also:

lsim, *step*, *initial*

Examples

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

5.5.3 control.matlab.initial

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

Initial condition response of a linear system

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

Parameters

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

  Numbers are converted to constant arrays with the correct shape.
- **input** (*int*) – This input is ignored, but present for compatibility with step and impulse.
- **output** (*int*) – If given, index of the output that is returned by this simulation.

Returns

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

See also:

`lsim`, `step`, `impulse`

Examples

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

5.5.4 control.matlab.lsim

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

Simulate the output of a linear system.

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

Parameters

- **sys** (*LTI (StateSpace, or TransferFunction)*) – LTI system to simulate
- **U** (*array-like or number, optional*) – Input array giving input at each time *T* (default = 0).

  If *U* is `None` or `0`, a special algorithm is used. This special algorithm is faster than the general algorithm, which is used otherwise.
- **T** (array-like, optional for discrete LTI *sys*) – Time steps at which the input is defined; values must be evenly spaced.
- **X0** (*array-like or number, optional*) – Initial condition (default = 0).

Returns

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

See also:

`step`, `initial`, `impulse`

Examples

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

5.6 Frequency-domain analysis

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

5.6.1 control.matlab.bode

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

Bode plot of the frequency response

Plots a bode gain and phase diagram

Parameters

- **sys** (*LTI, or list of LTI*) – System for which the Bode response is plotted and give. Optionally a list of systems can be entered, or several systems can be specified (i.e. several parameters). The sys arguments may also be interspersed with format strings. A frequency argument (array_like) may also be added, some examples: * >>> bode(sys, w) # one system, freq vector * >>> bode(sys1, sys2, . . . , sysN) # several systems * >>> bode(sys1, sys2, . . . , sysN, w) * >>> bode(sys1, 'plotstyle1', . . . , sysN, 'plotstyleN') # + plot formats
- **omega** (*freq_range*) – Range of frequencies in rad/s
- **dB** (*boolean*) – If True, plot result in dB
- **Hz** (*boolean*) – If True, plot frequency in Hz (omega must be provided in rad/sec)
- **deg** (*boolean*) – If True, return phase in degrees (else radians)
- **Plot** (*boolean*) – If True, plot magnitude and phase

Examples

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

Todo: Document these use cases

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

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

5.6.2 control.matlab.nyquist

`control.matlab.`**`nyquist`**(*syslist*, *omega=None*, *plot=True*, *label_freq=0*, *arrowhead_length=0.1*, *arrowhead_width=0.1*, *color=None*, **args*, ***kwargs*)

Nyquist plot for a system

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

Parameters

- **syslist** (*list of LTI*) – List of linear input/output systems (single system is OK)
- **omega** (*freq_range*) – Range of frequencies (list or bounds) in rad/sec
- **Plot** (*boolean*) – If True, plot magnitude
- **color** (*string*) – Used to specify the color of the plot
- **label_freq** (*int*) – Label every nth frequency on the plot
- **arrowhead_width** (*arrow head width*) –
- **arrowhead_length** (*arrow head length*) –
- ***args** (`matplotlib.pyplot.plot()` positional properties, optional) – Additional arguments for *matplotlib* plots (color, linestyle, etc)
- ****kwargs** (`matplotlib.pyplot.plot()` keyword properties, optional) – Additional keywords (passed to *matplotlib*)

Returns

- **real** (*array*) – real part of the frequency response array
- **imag** (*array*) – imaginary part of the frequency response array
- **freq** (*array*) – frequencies

Examples

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

5.6.3 control.matlab.nichols

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

Nichols plot for a system

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

Parameters

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

Returns

Return type None

5.6.4 control.matlab.margin

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

Calculate gain and phase margins and associated crossover frequencies

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

sys [StateSpace or TransferFunction] Linear SISO system

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

Returns

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

Examples

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

5.6.5 control.matlab.freqresp

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

Frequency response of an LTI system at multiple angular frequencies.

Parameters

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

Returns

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

See also:

evalfr, *bode*

Notes

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

Examples

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

Todo: Add example with MIMO system

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

5.6.6 control.matlab.evalfr

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

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

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

Parameters

- **sys** (StateSpace *or* TransferFunction) – Linear system
- **x** (*scalar*) – Complex number

Returns fresp

Return type ndarray

See also:

freqresp, *bode*

Notes

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

Examples

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

Todo: Add example with MIMO system

5.7 Compensator design

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

5.7.1 control.matlab.rlocus

control.matlab.**rlocus**(*sys*, *kvect=None*, *xlim=None*, *ylim=None*, *plotstr=None*, *plot=True*, *print_gain=None*, *grid=None*, *ax=None*, ***kwargs*)

Root locus plot

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

Parameters

- **sys** (*LTI object*) – Linear input/output systems (SISO only, for now).
- **kvect** (*list or ndarray, optional*) – List of gains to use in computing diagram.
- **xlim** (*tuple or list, optional*) – Set limits of x axis, normally with tuple (see matplotlib.axes).
- **ylim** (*tuple or list, optional*) – Set limits of y axis, normally with tuple (see matplotlib.axes).
- **plotstr** (matplotlib.pyplot.plot() format string, optional) – plotting style specification
- **plot** (*boolean, optional*) – If True (default), plot root locus diagram.
- **print_gain** (*bool*) – If True (default), report mouse clicks when close to the root locus branches, calculate gain, damping and print.
- **grid** (*bool*) – If True plot omega-damping grid. Default is False.
- **ax** (matplotlib.axes.Axes) – Axes on which to create root locus plot

Returns

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

5.7.2 control.matlab.sisotool

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

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

Parameters

- **sys** (*LTI object*) – Linear input/output systems (SISO only)
- **kvect** (*list or ndarray, optional*) – List of gains to use for plotting root locus
- **xlim_rlocus** (*tuple or list, optional*) – control of x-axis range, normally with tuple (see matplotlib.axes).
- **ylim_rlocus** (*tuple or list, optional*) – control of y-axis range
- **plotstr_rlocus** (matplotlib.pyplot.plot() format string, optional) – plotting style for the root locus plot(color, linestyle, etc)
- **rlocus_grid** (*boolean (default = False)*) – If True plot s-plane grid.

- **omega** (*freq_range*) – Range of frequencies in rad/sec for the bode plot
- **dB** (*boolean*) – If True, plot result in dB for the bode plot
- **Hz** (*boolean*) – If True, plot frequency in Hz for the bode plot (omega must be provided in rad/sec)
- **deg** (*boolean*) – If True, plot phase in degrees for the bode plot (else radians)
- **omega_limits** (*tuple, list, .. of two values*) – Limits of the to generate frequency vector. If Hz=True the limits are in Hz otherwise in rad/s.
- **omega_num** (*int*) – number of samples
- **margins_bode** (*boolean*) – If True, plot gain and phase margin in the bode plot
- **tvect** (*list or ndarray, optional*) – List of timesteps to use for closed loop step response

Examples

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

5.7.3 control.matlab.place

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

Place closed loop eigenvalues

K = place(A, B, p)

Parameters

- **A** (*2D array*) – Dynamics matrix
- **B** (*2D array*) – Input matrix
- **p** (*1D list*) – Desired eigenvalue locations

Returns **K** – Gain such that A - B K has eigenvalues given in p

Return type 2D array (or matrix)

Notes

Algorithm This is a wrapper function for `scipy.signal.place_poles()`, which implements the Tits and Yang algorithm[1]. It will handle SISO, MISO, and MIMO systems. If you want more control over the algorithm, use `scipy.signal.place_poles()` directly.

Limitations The algorithm will not place poles at the same location more than rank(B) times.

The return type for 2D arrays depends on the default class set for state space operations. See *use_numpy_matrix()*.

[1] A.L. Tits and Y. Yang, "Globally convergent algorithms for robust pole assignment by state feedback, IEEE Transactions on Automatic Control, Vol. 41, pp. 1432-1452, 1996.

References

Examples

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

See also:

place_varga, acker

Notes

The return type for 2D arrays depends on the default class set for state space operations. See *use_numpy_matrix()*.

5.7.4 control.matlab.lqr

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

Linear quadratic regulator design

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

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

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

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

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

Parameters

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

Returns

- **K** (*2D array (or matrix)*) – State feedback gains
- **S** (*2D array (or matrix)*) – Solution to Riccati equation
- **E** (*1D array*) – Eigenvalues of the closed loop system

See also:

lqe

Notes

The return type for 2D arrays depends on the default class set for state space operations. See *use_numpy_matrix()*.

Examples

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

5.8 State-space (SS) models

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

5.8.1 control.matlab.ctrb

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

Controllabilty matrix

Parameters

- **A** (*array_like or string*) – Dynamics and input matrix of the system
- **B** (*array_like or string*) – Dynamics and input matrix of the system

Returns **C** – Controllability matrix

Return type 2D array (or matrix)

Notes

The return type for 2D arrays depends on the default class set for state space operations. See *use_numpy_matrix()*.

Examples

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

5.8.2 control.matlab.obsv

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

Parameters

- **A** (*array_like or string*) – Dynamics and output matrix of the system
- **C** (*array_like or string*) – Dynamics and output matrix of the system

Returns **O** – Observability matrix

Return type 2D array (or matrix)

Notes

The return type for 2D arrays depends on the default class set for state space operations. See *use_numpy_matrix()*.

Examples

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

5.8.3 control.matlab.gram

control.matlab.**gram**(*sys*, *type*)
Gramian (controllability or observability)

Parameters

- **sys** (StateSpace) – System description
- **type** (*String*) – Type of desired computation. *type* is either 'c' (controllability) or 'o' (observability). To compute the Cholesky factors of Gramians use 'cf' (controllability) or 'of' (observability)

Returns **gram** – Gramian of system

Return type 2D array (or matrix)

Raises

- **ValueError** –
  - if system is not instance of StateSpace class * if *type* is not 'c', 'o', 'cf' or 'of' * if system is unstable (sys.A has eigenvalues not in left half plane)
- **ImportError** – if slycot routine sb03md cannot be found if slycot routine sb03od cannot be found

Notes

The return type for 2D arrays depends on the default class set for state space operations. See *use_numpy_matrix()*.

Examples

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

5.9 Model simplification

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

5.9.1 control.matlab.minreal

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

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

Parameters

- **sys** (StateSpace *or* TransferFunction) – Original system
- **tol** (*real*) – Tolerance
- **verbose** (*bool*) – Print results if True

Returns **rsys** – Cleaned model

Return type *StateSpace* or *TransferFunction*

5.9.2 control.matlab.hsvd

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

Calculate the Hankel singular values.

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

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

Return type array

See also:

gram

Notes

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

Examples

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

5.9.3 control.matlab.balred

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

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

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

Parameters

- **sys** (StateSpace) – Original system to reduce
- **orders** (*integer or array of integer*) – Desired order of reduced order model (if a vector, returns a vector of systems)
- **method** (*string*) – Method of removing states, either 'truncate' or 'matchdc'.
- **alpha** (*float*) – Redefines the stability boundary for eigenvalues of the system matrix A. By default for continuous-time systems, alpha <= 0 defines the stability boundary for the real part of A's eigenvalues and for discrete-time systems, 0 <= alpha <= 1 defines the stability boundary for the modulus of A's eigenvalues. See SLICOT routines AB09MD and AB09ND for more information.

Returns **rsys** – A reduced order model or a list of reduced order models if orders is a list.

Return type *StateSpace*

Raises

- **ValueError** – If *method* is not 'truncate' or 'matchdc'

- **ImportError** – if slycot routine ab09ad, ab09md, or ab09nd is not found
- **ValueError** – if there are more unstable modes than any value in orders

Examples

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

5.9.4 control.matlab.modred

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

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

Parameters

- **sys** (StateSpace) – Original system to reduce
- **ELIM** (*array*) – Vector of states to eliminate
- **method** (*string*) – Method of removing states in *ELIM*: either 'truncate' or 'matchdc'.

Returns **rsys** – A reduced order model

Return type *StateSpace*

Raises **ValueError** – Raised under the following conditions:

* if *method* is not either 'matchdc' or 'truncate'

* if eigenvalues of *sys.A* are not all in left half plane (*sys* must be stable)

Examples

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

5.9.5 control.matlab.era

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

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

Note: This function is not implemented yet.

Parameters

- **YY** (*array*) – *nout* x *nin* dimensional impulse-response data
- **m** (*integer*) – Number of rows in Hankel matrix
- **n** (*integer*) – Number of columns in Hankel matrix
- **nin** (*integer*) – Number of input variables
- **nout** (*integer*) – Number of output variables
- **r** (*integer*) – Order of model

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

Return type *StateSpace*

Examples

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

5.9.6 control.matlab.markov

`control.matlab.markov(Y, U, m=None, transpose=None)`

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

This function computes the Markov parameters for a discrete time system

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

given data for u and y. The algorithm assumes that that C A^k B = 0 for k > m-2 (see[1]). Note that the problem is ill-posed if the length of the input data is less than the desired number of Markov parameters (a warning message is generated in this case).

Parameters

- **Y** (*array_like*) – Output data. If the array is 1D, the system is assumed to be single input. If the array is 2D and transpose=False, the columns of *Y* are taken as time points, otherwise the rows of *Y* are taken as time points.
- **U** (*array_like*) – Input data, arranged in the same way as *Y*.
- **m** (*int, optional*) – Number of Markov parameters to output. Defaults to len(U).
- **transpose** (*bool, optional*) – Assume that input data is transposed relative to the standard *Time series data*. The default value is true for backward compatibility with legacy code.

Returns **H** – First m Markov parameters, [D CB CAB ...]

Return type ndarray

References

Notes

Currently only works for SISO systems.

This function does not currently comply with the Python Control Library *Time series data* for representation of time series data. Use *transpose=False* to make use of the standard convention (this will be updated in a future release).

[1] J.-N. Juang, M. Phan, L. G. Horta, and R. W. Longman, Identification of observer/Kalman filter Markov parameters - Theory and experiments. Journal of Guidance Control and Dynamics, 16(2), 320-329, 2012. http://doi.org/10.2514/3.21006

Examples

```
>>> T = numpy.linspace(0, 10, 100)
>>> U = numpy.ones((1, 100))
>>> T, Y, _ = forced_response(tf([1], [1, 0.5], True), T, U)
>>> H = markov(Y, U, 3, transpose=False)
```

5.10 Time delays

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

5.10.1 control.matlab.pade

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

Create a linear system that approximates a delay.

Return the numerator and denominator coefficients of the Pade approximation.

Parameters

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

Returns **num, den** – Polynomial coefficients of the delay model, in descending powers of s.

Return type array

Notes

Based on:

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

5.11 Matrix equation solvers and linear algebra

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

5.11.1 control.matlab.lyap

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

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

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

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

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

$$AX + XQ + C = 0$$

where A and Q are square matrices.

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

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

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

Parameters

- **A** (*2D array*) – Dynamics matrix
- **C** (*2D array, optional*) – If present, solve the Slyvester equation
- **E** (*2D array, optional*) – If present, solve the generalized Laypunov equation

Returns **Q** – Solution to the Lyapunov or Sylvester equation

Return type 2D array (or matrix)

Notes

The return type for 2D arrays depends on the default class set for state space operations. See `use_numpy_matrix()`.

5.11.2 control.matlab.dlyap

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

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

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

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

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

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

where A and Q are square matrices.

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

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

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

5.11.3 control.matlab.care

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

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

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

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

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

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

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

Parameters

- **A** (*2D arrays*) – Input matrices for the Riccati equation
- **B** (*2D arrays*) – Input matrices for the Riccati equation
- **Q** (*2D arrays*) – Input matrices for the Riccati equation
- **R** (*2D arrays, optional*) – Input matrices for generalized Riccati equation
- **S** (*2D arrays, optional*) – Input matrices for generalized Riccati equation
- **E** (*2D arrays, optional*) – Input matrices for generalized Riccati equation

Returns

- **X** (*2D array (or matrix)*) – Solution to the Ricatti equation
- **L** (*1D array*) – Closed loop eigenvalues
- **G** (*2D array (or matrix)*) – Gain matrix

Notes

The return type for 2D arrays depends on the default class set for state space operations. See `use_numpy_matrix()`.

5.11.4 control.matlab.dare

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

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

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

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

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

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

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

Parameters

- **A** (*2D arrays*) – Input matrices for the Riccati equation
- **B** (*2D arrays*) – Input matrices for the Riccati equation
- **Q** (*2D arrays*) – Input matrices for the Riccati equation
- **R** (*2D arrays, optional*) – Input matrices for generalized Riccati equation
- **S** (*2D arrays, optional*) – Input matrices for generalized Riccati equation
- **E** (*2D arrays, optional*) – Input matrices for generalized Riccati equation

Returns

- **X** (*2D array (or matrix)*) – Solution to the Ricatti equation
- **L** (*1D array*) – Closed loop eigenvalues
- **G** (*2D array (or matrix)*) – Gain matrix

Notes

The return type for 2D arrays depends on the default class set for state space operations. See `use_numpy_matrix()`.

5.12 Additional functions

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

5.12.1 control.matlab.gangof4

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

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** (*LTI*) – Linear input/output systems (process and control)
- **C** (*LTI*) – Linear input/output systems (process and control)
- **omega** (*array*) – Range of frequencies (list or bounds) in rad/sec
- ****kwargs** (`matplotlib.pyplot.plot()` keyword properties, optional) – Additional keywords (passed to *matplotlib*)

Returns

Return type None

5.12.2 control.matlab.unwrap

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

Unwrap a phase angle to give a continuous curve

Parameters

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

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

Return type array_like

Examples

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

5.13 Functions imported from other modules

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

CHAPTER

SIX

DIFFERENTIALLY FLAT SYSTEMS

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

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

6.1 Overview of differential flatness

A nonlinear differential equation of the form

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

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

$$z = \alpha(x, u, \dot{u} \ldots, u^{(p)})$$

and we can write the solutions of the nonlinear system as functions of z and a finite number of derivatives

$$\begin{aligned} x &= \beta(z, \dot{z}, \ldots, z^{(q)}) \\ u &= \gamma(z, \dot{z}, \ldots, z^{(q)}). \end{aligned} \tag{6.1}$$

For a differentially flat system, all of the feasible trajectories for the system can be written as functions of a flat output $z(\cdot)$ and its derivatives. The number of flat outputs is always equal to the number of system inputs.

Differentially flat systems are useful in situations where explicit trajectory generation is required. Since the behavior of a flat system is determined by the flat outputs, we can plan trajectories in output space, and then map these to appropriate inputs. Suppose we wish to generate a feasible trajectory for the the nonlinear system

$$\dot{x} = f(x, u), \qquad x(0) = x_0,\, x(T) = x_f.$$

If the system is differentially flat then

$$\begin{aligned} x(0) &= \beta\big(z(0), \dot{z}(0), \ldots, z^{(q)}(0)\big) = x_0, \\ x(T) &= \gamma\big(z(T), \dot{z}(T), \ldots, z^{(q)}(T)\big) = x_f, \end{aligned}$$

and we see that the initial and final condition in the full state space depends on just the output z and its derivatives at the initial and final times. Thus any trajectory for z that satisfies these boundary conditions will be a feasible trajectory for the system, using equation (6.1) to determine the full state space and input trajectories.

In particular, given initial and final conditions on z and its derivatives that satisfy the initial and final conditions any curve $z(\cdot)$ satisfying those conditions will correspond to a feasible trajectory of the system. We can parameterize the flat output trajectory using a set of smooth basis functions $\psi_i(t)$:

$$z(t) = \sum_{i=1}^{N} \alpha_i \psi_i(t), \qquad \alpha_i \in R$$

We seek a set of coefficients α_i, $i = 1, \dots, N$ such that $z(t)$ satisfies the boundary conditions for $x(0)$ and $x(T)$. The derivatives of the flat output can be computed in terms of the derivatives of the basis functions:

$$\begin{aligned}\dot z(t) &= \sum_{i=1}^{N} \alpha_i \dot\psi_i(t) \\ &\vdots \\ \dot z^{(q)}(t) &= \sum_{i=1}^{N} \alpha_i \psi_i^{(q)}(t).\end{aligned}$$

We can thus write the conditions on the flat outputs and their derivatives as

$$\begin{bmatrix} \psi_1(0) & \psi_2(0) & \dots & \psi_N(0) \\ \dot\psi_1(0) & \dot\psi_2(0) & \dots & \dot\psi_N(0) \\ \vdots & \vdots & & \vdots \\ \psi_1^{(q)}(0) & \psi_2^{(q)}(0) & \dots & \psi_N^{(q)}(0) \\ \psi_1(T) & \psi_2(T) & \dots & \psi_N(T) \\ \dot\psi_1(T) & \dot\psi_2(T) & \dots & \dot\psi_N(T) \\ \vdots & \vdots & & \vdots \\ \psi_1^{(q)}(T) & \psi_2^{(q)}(T) & \dots & \psi_N^{(q)}(T) \end{bmatrix} \begin{bmatrix} \alpha_1 \\ \vdots \\ \alpha_N \end{bmatrix} = \begin{bmatrix} z(0) \\ \dot z(0) \\ \vdots \\ z^{(q)}(0) \\ z(T) \\ \dot z(T) \\ \vdots \\ z^{(q)}(T) \end{bmatrix}$$

This equation is a *linear* equation of the form

$$M\alpha = \begin{bmatrix} \bar z(0) \\ \bar z(T) \end{bmatrix}$$

where $\bar z$ is called the *flat flag* for the system. Assuming that M has a sufficient number of columns and that it is full column rank, we can solve for a (possibly non-unique) α that solves the trajectory generation problem.

6.2 Module usage

To create a trajectory for a differentially flat system, a `FlatSystem` object must be created. This is done by specifying the *forward* and *reverse* mappings between the system state/input and the differentially flat outputs and their derivatives ("flat flag").

The `forward()` method computes the flat flag given a state and input:

zflag = sys.forward(x, u)

The `reverse()` method computes the state and input given the flat flag:

x, u = sys.reverse(zflag)

The flag $\bar z$ is implemented as a list of flat outputs z_i and their derivatives up to order q_i:

zflag[i][j] = $z_i^{(j)}$

The number of flat outputs must match the number of system inputs.

For a linear system, a flat system representation can be generated using the `LinearFlatSystem` class:

> flatsys = control.flatsys.LinearFlatSystem(linsys)

For more general systems, the *FlatSystem* object must be created manually

> flatsys = control.flatsys.FlatSystem(nstate, ninputs, forward, reverse)

In addition to the flat system descriptionn, a set of basis functions $\phi_i(t)$ must be chosen. The *FlatBasis* class is used to represent the basis functions. A polynomial basis function of the form 1, t, t^2, ... can be computed using the *PolyBasis* class, which is initialized by passing the desired order of the polynomial basis set:

> polybasis = control.flatsys.PolyBasis(N)

Once the system and basis function have been defined, the `point_to_point()` function can be used to compute a trajectory between initial and final states and inputs:

> traj = control.flatsys.point_to_point(x0, u0, xf, uf, Tf, basis=polybasis)

The returned object has class `SystemTrajectory` and can be used to compute the state and input trajectory between the initial and final condition:

> xd, ud = traj.eval(T)

where *T* is a list of times on which the trajectory should be evaluated (e.g., *T = numpy.linspace(0, Tf, M)*.

6.3 Example

To illustrate how we can use a two degree-of-freedom design to improve the performance of the system, consider the problem of steering a car to change lanes on a road. We use the non-normalized form of the dynamics, which are derived *Feedback Systems* by Astrom and Murray, Example 3.11.

```
import control.flatsys as fs

# Function to take states, inputs and return the flat flag
def vehicle_flat_forward(x, u, params={}):
    # Get the parameter values
    b = params.get('wheelbase', 3.)

    # Create a list of arrays to store the flat output and its derivatives
    zflag = [np.zeros(3), np.zeros(3)]

    # Flat output is the x, y position of the rear wheels
    zflag[0][0] = x[0]
    zflag[1][0] = x[1]

    # First derivatives of the flat output
    zflag[0][1] = u[0] * np.cos(x[2])  # dx/dt
    zflag[1][1] = u[0] * np.sin(x[2])  # dy/dt

    # First derivative of the angle
    thdot = (u[0]/b) * np.tan(u[1])

    # Second derivatives of the flat output (setting vdot = 0)
    zflag[0][2] = -u[0] * thdot * np.sin(x[2])
    zflag[1][2] =  u[0] * thdot * np.cos(x[2])
```

(continues on next page)

(continued from previous page)

```
    return zflag

# Function to take the flat flag and return states, inputs
def vehicle_flat_reverse(zflag, params={}):
    # Get the parameter values
    b = params.get('wheelbase', 3.)

    # Create a vector to store the state and inputs
    x = np.zeros(3)
    u = np.zeros(2)

    # Given the flat variables, solve for the state
    x[0] = zflag[0][0]  # x position
    x[1] = zflag[1][0]  # y position
    x[2] = np.arctan2(zflag[1][1], zflag[0][1])  # tan(theta) = ydot/xdot

    # And next solve for the inputs
    u[0] = zflag[0][1] * np.cos(x[2]) + zflag[1][1] * np.sin(x[2])
    u[1] = np.arctan2(
        (zflag[1][2] * np.cos(x[2]) - zflag[0][2] * np.sin(x[2])), u[0]/b)

    return x, u

vehicle_flat = fs.FlatSystem(
    3, 2, forward=vehicle_flat_forward, reverse=vehicle_flat_reverse)
```

To find a trajectory from an initial state x_0 to a final state x_f in time T_f we solve a point-to-point trajectory generation problem. We also set the initial and final inputs, which sets the vehicle velocity v and steering wheel angle δ at the endpoints.

```
# Define the endpoints of the trajectory
x0 = [0., -2., 0.]; u0 = [10., 0.]
xf = [100., 2., 0.]; uf = [10., 0.]
Tf = 10

# Define a set of basis functions to use for the trajectories
poly = fs.PolyFamily(6)

# Find a trajectory between the initial condition and the final condition
traj = fs.point_to_point(vehicle_flat, x0, u0, xf, uf, Tf, basis=poly)

# Create the trajectory
t = np.linspace(0, Tf, 100)
x, u = traj.eval(t)
```

6.4 Module classes and functions

6.4.1 Flat systems classes

BasisFamily(N)	Base class for implementing basis functions for flat systems.
FlatSystem(forward, reverse[, updfcn, …])	Base class for representing a differentially flat system.

continues on next page

Table 1 – continued from previous page

`LinearFlatSystem`(linsys[, inputs, outputs, ...])	
`PolyFamily`(N)	Polynomial basis functions.
`SystemTrajectory`(sys, basis[, coeffs, flaglen])	Class representing a system trajectory.

control.flatsys.BasisFamily

class control.flatsys.**BasisFamily**(*N*)

Base class for implementing basis functions for flat systems.

A BasisFamily object is used to construct trajectories for a flat system. The class must implement a single function that computes the jth derivative of the ith basis function at a time t:

$z_i^{(q)}(t)$ = basis.eval_deriv(self, i, j, t)

__init__(*N*)

Create a basis family of order N.

Methods

`__init__`(N)	Create a basis family of order N.

control.flatsys.FlatSystem

class control.flatsys.**FlatSystem**(*forward*, *reverse*, *updfcn=None*, *outfcn=None*, *inputs=None*, *outputs=None*, *states=None*, *params={}*, *dt=None*, *name=None*)

Base class for representing a differentially flat system.

The FlatSystem class is used as a base class to describe differentially flat systems for trajectory generation. The class must implement two functions:

zflag = flatsys.foward(x, u) This function computes the flag (derivatives) of the flat output. The inputs to this function are the state 'x' and inputs 'u' (both 1D arrays). The output should be a 2D array with the first dimension equal to the number of system inputs and the second dimension of the length required to represent the full system dynamics (typically the number of states)

x, u = flatsys.reverse(zflag) This function system state and inputs give the the flag (derivatives) of the flat output. The input to this function is an 2D array whose first dimension is equal to the number of system inputs and whose second dimension is of length required to represent the full system dynamics (typically the number of states). The output is the state *x* and inputs *u* (both 1D arrays).

A flat system is also an input/output system supporting simulation, composition, and linearization. If the update and output methods are given, they are used in place of the flat coordinates.

__init__(*forward*, *reverse*, *updfcn=None*, *outfcn=None*, *inputs=None*, *outputs=None*, *states=None*, *params={}*, *dt=None*, *name=None*)

Create a differentially flat input/output system.

The FlatIOSystem constructor is used to create an input/output system object that also represents a differentially flat system. The output of the system does not need to be the differentially flat output.

Parameters

- **forward** (*callable*) – A function to compute the flat flag given the states and input.
- **reverse** (*callable*) – A function to compute the states and input given the flat flag.

- **updfcn** (*callable, optional*) – Function returning the state update function

  updfcn(t, x, u[, param]) -> array

  where *x* is a 1-D array with shape (nstates,), *u* is a 1-D array with shape (ninputs,), *t* is a float representing the currrent time, and *param* is an optional dict containing the values of parameters used by the function. If not specified, the state space update will be computed using the flat system coordinates.

- **outfcn** (*callable*) – Function returning the output at the given state

  outfcn(t, x, u[, param]) -> array

  where the arguments are the same as for *upfcn*. If not specified, the output will be the flat outputs.

- **inputs** (*int, list of str, or None*) – Description of the system inputs. This can be given as an integer count or as a list of strings that name the individual signals. If an integer count is specified, the names of the signal will be of the form *s[i]* (where *s* is one of *u*, *y*, or *x*). If this parameter is not given or given as *None*, the relevant quantity will be determined when possible based on other information provided to functions using the system.

- **outputs** (*int, list of str, or None*) – Description of the system outputs. Same format as *inputs*.

- **states** (*int, list of str, or None*) – Description of the system states. Same format as *inputs*.

- **dt** (*None, True or float, optional*) – System timebase. None (default) indicates continuous time, True indicates discrete time with undefined sampling time, positive number is discrete time with specified sampling time.

- **params** (*dict, optional*) – Parameter values for the systems. Passed to the evaluation functions for the system as default values, overriding internal defaults.

- **name** (*string, optional*) – System name (used for specifying signals)

Returns Input/output system object

Return type *InputOutputSystem*

Methods

__init__(forward, reverse[, updfcn, outfcn, . . .])	Create a differentially flat input/output system.
copy([newname])	Make a copy of an input/output system.
feedback([other, sign, params])	Feedback interconnection between two input/output systems
find_input(name)	Find the index for an input given its name (*None* if not found)
find_output(name)	Find the index for an output given its name (*None* if not found)
find_state(name)	Find the index for a state given its name (*None* if not found)
forward(x, u[, params])	Compute the flat flag given the states and input.
linearize(x0, u0[, t, params, eps])	Linearize an input/output system at a given state and input.
name_or_default([name])	

continues on next page

Table 3 – continued from previous page

`reverse`(zflag[, params])	Compute the states and input given the flat flag.
`set_inputs`(inputs[, prefix])	Set the number/names of the system inputs.
`set_outputs`(outputs[, prefix])	Set the number/names of the system outputs.
`set_states`(states[, prefix])	Set the number/names of the system states.

Attributes

idCounter

copy (*newname=None*)
Make a copy of an input/output system.

feedback (*other=1, sign=- 1, params={}*)
Feedback interconnection between two input/output systems

Parameters

- **sys1** (`InputOutputSystem`) – The primary process.
- **sys2** (`InputOutputSystem`) – The feedback process (often a feedback controller).
- **sign** (*scalar, optional*) – The sign of feedback. *sign* = -1 indicates negative feedback, and *sign* = 1 indicates positive feedback. *sign* is an optional argument; it assumes a value of -1 if not specified.

Returns **out**

Return type *InputOutputSystem*

Raises **ValueError** – if the inputs, outputs, or timebases of the systems are incompatible.

find_input (*name*)
Find the index for an input given its name (*None* if not found)

find_output (*name*)
Find the index for an output given its name (*None* if not found)

find_state (*name*)
Find the index for a state given its name (*None* if not found)

forward (*x, u, params={}*)
Compute the flat flag given the states and input.

Given the states and inputs for a system, compute the flat outputs and their derivatives (the flat "flag") for the system.

Parameters

- **x** (*list or array*) – The state of the system.
- **u** (*list or array*) – The input to the system.
- **params** (*dict, optional*) – Parameter values for the system. Passed to the evaluation functions for the system as default values, overriding internal defaults.

Returns **zflag** – For each flat output z_i, zflag[i] should be an ndarray of length q_i that contains the flat output and its first q_i derivatives.

Return type list of 1D arrays

linearize(*x0*, *u0*, *t=0*, *params={}*, *eps=1e-06*)
Linearize an input/output system at a given state and input.

Return the linearization of an input/output system at a given state and input value as a StateSpace system. See `linearize()` for complete documentation.

reverse(*zflag*, *params={}*)
Compute the states and input given the flat flag.

Parameters

- **zflag** (*list of arrays*) – For each flat output z_i, zflag[i] should be an ndarray of length q_i that contains the flat output and its first q_i derivatives.
- **params** (*dict, optional*) – Parameter values for the system. Passed to the evaluation functions for the system as default values, overriding internal defaults.

Returns

- **x** (*1D array*) – The state of the system corresponding to the flat flag.
- **u** (*1D array*) – The input to the system corresponding to the flat flag.

set_inputs(*inputs*, *prefix='u'*)
Set the number/names of the system inputs.

Parameters

- **inputs** (*int, list of str, or None*) – Description of the system inputs. This can be given as an integer count or as a list of strings that name the individual signals. If an integer count is specified, the names of the signal will be of the form *u[i]* (where the prefix *u* can be changed using the optional prefix parameter).
- **prefix** (*string, optional*) – If *inputs* is an integer, create the names of the states using the given prefix (default = 'u'). The names of the input will be of the form *prefix[i]*.

set_outputs(*outputs*, *prefix='y'*)
Set the number/names of the system outputs.

Parameters

- **outputs** (*int, list of str, or None*) – Description of the system outputs. This can be given as an integer count or as a list of strings that name the individual signals. If an integer count is specified, the names of the signal will be of the form *u[i]* (where the prefix *u* can be changed using the optional prefix parameter).
- **prefix** (*string, optional*) – If *outputs* is an integer, create the names of the states using the given prefix (default = 'y'). The names of the input will be of the form *prefix[i]*.

set_states(*states*, *prefix='x'*)
Set the number/names of the system states.

Parameters

- **states** (*int, list of str, or None*) – Description of the system states. This can be given as an integer count or as a list of strings that name the individual signals. If an integer count is specified, the names of the signal will be of the form *u[i]* (where the prefix *u* can be changed using the optional prefix parameter).
- **prefix** (*string, optional*) – If *states* is an integer, create the names of the states using the given prefix (default = 'x'). The names of the input will be of the form *prefix[i]*.

control.flatsys.LinearFlatSystem

class control.flatsys.**LinearFlatSystem**(*linsys*, *inputs=None*, *outputs=None*, *states=None*, *name=None*)

__init__(*linsys*, *inputs=None*, *outputs=None*, *states=None*, *name=None*)

Define a flat system from a SISO LTI system.

Given a reachable, single-input/single-output, linear time-invariant system, create a differentially flat system representation.

Parameters

- **linsys** (`StateSpace`) – LTI StateSpace system to be converted
- **inputs** (*int, list of str or None, optional*) – Description of the system inputs. This can be given as an integer count or as a list of strings that name the individual signals. If an integer count is specified, the names of the signal will be of the form *s[i]* (where *s* is one of *u*, *y*, or *x*). If this parameter is not given or given as *None*, the relevant quantity will be determined when possible based on other information provided to functions using the system.
- **outputs** (*int, list of str or None, optional*) – Description of the system outputs. Same format as *inputs*.
- **states** (*int, list of str, or None, optional*) – Description of the system states. Same format as *inputs*.
- **dt** (*None, True or float, optional*) – System timebase. None (default) indicates continuous time, True indicates discrete time with undefined sampling time, positive number is discrete time with specified sampling time.
- **params** (*dict, optional*) – Parameter values for the systems. Passed to the evaluation functions for the system as default values, overriding internal defaults.
- **name** (*string, optional*) – System name (used for specifying signals)

Returns **iosys** – Linear system represented as an flat input/output system

Return type *LinearFlatSystem*

Methods

`__init__`(linsys[, inputs, outputs, states, name])	Define a flat system from a SISO LTI system.
`append`(other)	Append a second model to the present model.
`copy`([newname])	Make a copy of an input/output system.
`damp`()	Natural frequency, damping ratio of system poles
`dcgain`()	Return the zero-frequency gain
`evalfr`(omega)	Evaluate a SS system's transfer function at a single frequency.
`feedback`([other, sign, params])	Feedback interconnection between two input/output systems
`find_input`(name)	Find the index for an input given its name (*None* if not found)
`find_output`(name)	Find the index for an output given its name (*None* if not found)

continues on next page

Table 5 – continued from previous page

`find_state`(name)	Find the index for a state given its name (*None* if not found)
`forward`(x, u)	Compute the flat flag given the states and input.
`freqresp`(omega)	Evaluate the system's transfer function at a list of frequencies
`horner`(s)	Evaluate the systems's transfer function for a complex variable
`is_static_gain`()	True if and only if the system has no dynamics, that is, if A and B are zero.
`isctime`([strict])	Check to see if a system is a continuous-time system
`isdtime`([strict])	Check to see if a system is a discrete-time system
`issiso`()	Check to see if a system is single input, single output
`lft`(other[, nu, ny])	Return the Linear Fractional Transformation.
`linearize`(x0, u0[, t, params, eps])	Linearize an input/output system at a given state and input.
`minreal`([tol])	Calculate a minimal realization, removes unobservable and uncontrollable states
`name_or_default`([name])	
`pole`()	Compute the poles of a state space system.
`returnScipySignalLTI`()	Return a list of a list of `scipy.signal.lti` objects.
`reverse`(zflag)	Compute the states and input given the flat flag.
`sample`(Ts[, method, alpha, prewarp_frequency])	Convert a continuous time system to discrete time
`set_inputs`(inputs[, prefix])	Set the number/names of the system inputs.
`set_outputs`(outputs[, prefix])	Set the number/names of the system outputs.
`set_states`(states[, prefix])	Set the number/names of the system states.
`zero`()	Compute the zeros of a state space system.

Attributes

`idCounter`

append(*other*)

Append a second model to the present model.

The second model is converted to state-space if necessary, inputs and outputs are appended and their order is preserved

copy(*newname=None*)

Make a copy of an input/output system.

damp()

Natural frequency, damping ratio of system poles

Returns

- **wn** (*array*) – Natural frequencies for each system pole
- **zeta** (*array*) – Damping ratio for each system pole
- **poles** (*array*) – Array of system poles

dcgain()

Return the zero-frequency gain

The zero-frequency gain of a continuous-time state-space system is given by:

and of a discrete-time state-space system by:

Returns gain – An array of shape (outputs,inputs); the array will either be the zero-frequency (or DC) gain, or, if the frequency response is singular, the array will be filled with np.nan.

Return type ndarray

evalfr(*omega*)
Evaluate a SS system's transfer function at a single frequency.

self._evalfr(omega) returns the value of the transfer function matrix with input value s = i * omega.

feedback(*other=1*, *sign=- 1*, *params={}*)
Feedback interconnection between two input/output systems

Parameters

- **sys1** (`InputOutputSystem`) – The primary process.
- **sys2** (`InputOutputSystem`) – The feedback process (often a feedback controller).
- **sign** (*scalar, optional*) – The sign of feedback. *sign* = -1 indicates negative feedback, and *sign* = 1 indicates positive feedback. *sign* is an optional argument; it assumes a value of -1 if not specified.

Returns out

Return type *InputOutputSystem*

Raises ValueError – if the inputs, outputs, or timebases of the systems are incompatible.

find_input(*name*)
Find the index for an input given its name (*None* if not found)

find_output(*name*)
Find the index for an output given its name (*None* if not found)

find_state(*name*)
Find the index for a state given its name (*None* if not found)

forward(*x*, *u*)
Compute the flat flag given the states and input.

See `control.flatsys.FlatSystem.forward()` for more info.

freqresp(*omega*)
Evaluate the system's transfer function at a list of frequencies

Reports the frequency response of the system,

G(j*omega) = mag*exp(j*phase)

for continuous time. For discrete time systems, the response is evaluated around the unit circle such that

G(exp(j*omega*dt)) = mag*exp(j*phase).

Parameters omega (*array_like*) – A list of frequencies in radians/sec at which the system should be evaluated. The list can be either a python list or a numpy array and will be sorted before evaluation.

Returns

- **mag** (*(self.outputs, self.inputs, len(omega)) ndarray*) – The magnitude (absolute value, not dB or log10) of the system frequency response.

- **phase** (*(self.outputs, self.inputs, len(omega)) ndarray*) – The wrapped phase in radians of the system frequency response.
- **omega** (*ndarray*) – The list of sorted frequencies at which the response was evaluated.

horner(*s*)
Evaluate the systems's transfer function for a complex variable

Returns a matrix of values evaluated at complex variable s.

is_static_gain()
True if and only if the system has no dynamics, that is, if A and B are zero.

isctime(*strict=False*)
Check to see if a system is a continuous-time system

Parameters

- **sys** (`LTI system`) – System to be checked
- **strict** (`bool, optional`) – If strict is True, make sure that timebase is not None. Default is False.

isdtime(*strict=False*)
Check to see if a system is a discrete-time system

Parameters **strict** (`bool, optional`) – If strict is True, make sure that timebase is not None. Default is False.

issiso()
Check to see if a system is single input, single output

lft(*other*, *nu=- 1*, *ny=- 1*)
Return the Linear Fractional Transformation.

A definition of the LFT operator can be found in Appendix A.7, page 512 in the 2nd Edition, Multivariable Feedback Control by Sigurd Skogestad.

An alternative definition can be found here: https://www.mathworks.com/help/control/ref/lft.html

Parameters

- **other** (`LTI`) – The lower LTI system
- **ny** (`int, optional`) – Dimension of (plant) measurement output.
- **nu** (`int, optional`) – Dimension of (plant) control input.

linearize(*x0*, *u0*, *t=0*, *params={}*, *eps=1e-06*)
Linearize an input/output system at a given state and input.

Return the linearization of an input/output system at a given state and input value as a StateSpace system. See `linearize()` for complete documentation.

minreal(*tol=0.0*)
Calculate a minimal realization, removes unobservable and uncontrollable states

pole()
Compute the poles of a state space system.

returnScipySignalLTI()
Return a list of a list of `scipy.signal.lti` objects.

For instance,

```
>>> out = ssobject.returnScipySignalLTI()
>>> out[3][5]
```

is a `scipy.signal.lti` object corresponding to the transfer function from the 6th input to the 4th output.

reverse(*zflag*)
Compute the states and input given the flat flag.

See `control.flatsys.FlatSystem.reverse()` for more info.

sample(*Ts*, *method='zoh'*, *alpha=None*, *prewarp_frequency=None*)
Convert a continuous time system to discrete time

Creates a discrete-time system from a continuous-time system by sampling. Multiple methods of conversion are supported.

Parameters

- **Ts** (*float*) – Sampling period
- **method** (*{"gbt", "bilinear", "euler", "backward_diff", "zoh"}*) – Which method to use:
  - gbt: generalized bilinear transformation
  - bilinear: Tustin's approximation ("gbt" with alpha=0.5)
  - euler: Euler (or forward differencing) method ("gbt" with alpha=0)
  - backward_diff: Backwards differencing ("gbt" with alpha=1.0)
  - zoh: zero-order hold (default)
- **alpha** (*float within [0, 1]*) – The generalized bilinear transformation weighting parameter, which should only be specified with method="gbt", and is ignored otherwise
- **prewarp_frequency** (*float within [0, infinity)*) – The frequency [rad/s] at which to match with the input continuous- time system's magnitude and phase (the gain=1 crossover frequency, for example). Should only be specified with method='bilinear' or 'gbt' with alpha=0.5 and ignored otherwise.

Returns **sysd** – Discrete time system, with sampling rate Ts

Return type *StateSpace*

Notes

Uses `scipy.signal.cont2discrete()`

Examples

```
>>> sys = StateSpace(0, 1, 1, 0)
>>> sysd = sys.sample(0.5, method='bilinear')
```

set_inputs (*inputs*, *prefix='u'*)
Set the number/names of the system inputs.

Parameters

- **inputs** (*int, list of str, or None*) – Description of the system inputs. This can be given as an integer count or as a list of strings that name the individual signals. If an integer count is specified, the names of the signal will be of the form *u[i]* (where the prefix *u* can be changed using the optional prefix parameter).
- **prefix** (*string, optional*) – If *inputs* is an integer, create the names of the states using the given prefix (default = 'u'). The names of the input will be of the form *prefix[i]*.

set_outputs (*outputs*, *prefix='y'*)
Set the number/names of the system outputs.

Parameters

- **outputs** (*int, list of str, or None*) – Description of the system outputs. This can be given as an integer count or as a list of strings that name the individual signals. If an integer count is specified, the names of the signal will be of the form *u[i]* (where the prefix *u* can be changed using the optional prefix parameter).
- **prefix** (*string, optional*) – If *outputs* is an integer, create the names of the states using the given prefix (default = 'y'). The names of the input will be of the form *prefix[i]*.

set_states (*states*, *prefix='x'*)
Set the number/names of the system states.

Parameters

- **states** (*int, list of str, or None*) – Description of the system states. This can be given as an integer count or as a list of strings that name the individual signals. If an integer count is specified, the names of the signal will be of the form *u[i]* (where the prefix *u* can be changed using the optional prefix parameter).
- **prefix** (*string, optional*) – If *states* is an integer, create the names of the states using the given prefix (default = 'x'). The names of the input will be of the form *prefix[i]*.

zero ()
Compute the zeros of a state space system.

control.flatsys.PolyFamily

class control.flatsys.**PolyFamily** (*N*)
Polynomial basis functions.

This class represents the family of polynomials of the form

$$\phi_i(t) = t^i$$

__init__ (*N*)
Create a polynomial basis of order N.

Methods

`__init__`(N)	Create a polynomial basis of order N.
`eval_deriv`(i, k, t)	Evaluate the kth derivative of the ith basis function at time t.

eval_deriv (*i*, *k*, *t*)
Evaluate the kth derivative of the ith basis function at time t.

control.flatsys.SystemTrajectory

class control.flatsys.**SystemTrajectory** (*sys*, *basis*, *coeffs=[]*, *flaglen=[]*)
Class representing a system trajectory.

The *SystemTrajectory* class is used to represent the trajectory of a (differentially flat) system. Used by the `point_to_point()` function to return a trajectory.

__init__ (*sys*, *basis*, *coeffs=[]*, *flaglen=[]*)
Initilize a system trajectory object.

Parameters

- **sys** (`FlatSystem`) – Flat system object associated with this trajectory.
- **basis** (`BasisFamily`) – Family of basis vectors to use to represent the trajectory.
- **coeffs** (*list of 1D arrays, optional*) – For each flat output, define the coefficients of the basis functions used to represent the trajectory. Defaults to an empty list.
- **flaglen** (*list of ints, optional*) – For each flat output, the number of derivatives of the flat output used to define the trajectory. Defaults to an empty list.

Methods

`__init__`(sys, basis[, coeffs, flaglen])	Initilize a system trajectory object.
`eval`(tlist)	Return the state and input for a trajectory at a list of times.

eval (*tlist*)
Return the state and input for a trajectory at a list of times.

Evaluate the trajectory at a list of time points, returning the state and input vectors for the trajectory:

x, u = traj.eval(tlist)

Parameters **tlist** (*1D array*) – List of times to evaluate the trajectory.

Returns

- **x** (*2D array*) – For each state, the values of the state at the given times.
- **u** (*2D array*) – For each input, the values of the input at the given times.

6.4.2 Flat systems functions

`point_to_point`(sys, x0, u0, xf, uf, Tf[, ...])	Compute trajectory between an initial and final conditions.

CHAPTER

SEVEN

INPUT/OUTPUT SYSTEMS

The *iosys* module contains the `InputOutputSystem` class that represents (possibly nonlinear) input/output systems. The `InputOutputSystem` class is a general class that defines any continuous or discrete time dynamical system. Input/output systems can be simulated and also used to compute equilibrium points and linearizations.

7.1 Module usage

An input/output system is defined as a dynamical system that has a system state as well as inputs and outputs (either inputs or states can be empty). The dynamics of the system can be in continuous or discrete time. To simulate an input/output system, use the *input_output_response()* function:

```
t, y = input_output_response(io_sys, T, U, X0, params)
```

An input/output system can be linearized around an equilibrium point to obtain a *StateSpace* linear system. Use the `find_eqpt()` function to obtain an equilibrium point and the `linearize()` function to linearize about that equilibrium point:

```
xeq, ueq = find_eqpt(io_sys, X0, U0)
ss_sys = linearize(io_sys, xeq, ueq)
```

Input/output systems can be created from state space LTI systems by using the `LinearIOSystem` class`:

```
io_sys = LinearIOSystem(ss_sys)
```

Nonlinear input/output systems can be created using the `NonlinearIOSystem` class, which requires the definition of an update function (for the right hand side of the differential or different equation) and and output function (computes the outputs from the state):

```
io_sys = NonlinearIOSystem(updfcn, outfcn, inputs=M, outputs=P, states=N)
```

More complex input/output systems can be constructed by using the `InterconnectedSystem` class, which allows a collection of input/output subsystems to be combined with internal connections between the subsystems and a set of overall system inputs and outputs that link to the subsystems:

```
steering = ct.InterconnectedSystem(
    (plant, controller), name='system',
    connections=(('controller.e', '-plant.y')),
    inplist=('controller.e'), inputs='r',
    outlist=('plant.y'), outputs='y')
```

Interconnected systems can also be created using block diagram manipulations such as the *series()*, *parallel()*, and *feedback()* functions. The `InputOutputSystem` class also supports various algebraic operations such as * (series interconnection) and + (parallel interconnection).

7.2 Example

To illustrate the use of the input/output systems module, we create a model for a predator/prey system, following the notation and parameter values in FBS2e.

We begin by defining the dynamics of the system

```
import control
import numpy as np
import matplotlib.pyplot as plt

def predprey_rhs(t, x, u, params):
    # Parameter setup
    a = params.get('a', 3.2)
    b = params.get('b', 0.6)
    c = params.get('c', 50.)
    d = params.get('d', 0.56)
    k = params.get('k', 125)
    r = params.get('r', 1.6)

    # Map the states into local variable names
    H = x[0]
    L = x[1]

    # Compute the control action (only allow addition of food)
    u_0 = u if u > 0 else 0

    # Compute the discrete updates
    dH = (r + u_0) * H * (1 - H/k) - (a * H * L)/(c + H)
    dL = b * (a * H *  L)/(c + H) - d * L

    return [dH, dL]
```

We now create an input/output system using these dynamics:

```
io_predprey = control.NonlinearIOSystem(
    predprey_rhs, None, inputs=('u'), outputs=('H', 'L'),
    states=('H', 'L'), name='predprey')
```

Note that since we have not specified an output function, the entire state will be used as the output of the system.

The *io_predprey* system can now be simulated to obtain the open loop dynamics of the system:

```
X0 = [25, 20]                 # Initial H, L
T = np.linspace(0, 70, 500)   # Simulation 70 years of time

# Simulate the system
t, y = control.input_output_response(io_predprey, T, 0, X0)

# Plot the response
plt.figure(1)
plt.plot(t, y[0])
plt.plot(t, y[1])
plt.legend(['Hare', 'Lynx'])
plt.show(block=False)
```

We can also create a feedback controller to stabilize a desired population of the system. We begin by finding the (unstable) equilibrium point for the system and computing the linearization about that point.

```
eqpt = control.find_eqpt(io_predprey, X0, 0)
xeq = eqpt[0]                          # choose the nonzero equilibrium point
lin_predprey = control.linearize(io_predprey, xeq, 0)
```

We next compute a controller that stabilizes the equilibrium point using eigenvalue placement and computing the feedforward gain using the number of lynxes as the desired output (following FBS2e, Example 7.5):

```
K = control.place(lin_predprey.A, lin_predprey.B, [-0.1, -0.2])
A, B = lin_predprey.A, lin_predprey.B
C = np.array([[0, 1]])                 # regulated output = number of lynxes
kf = -1/(C @ np.linalg.inv(A - B @ K) @ B)
```

To construct the control law, we build a simple input/output system that applies a corrective input based on deviations from the equilibrium point. This system has no dynamics, since it is a static (affine) map, and can constructed using the *~control.ios.NonlinearIOSystem* class:

```
io_controller = control.NonlinearIOSystem(
  None,
  lambda t, x, u, params: -K @ (u[1:] - xeq) + kf * (u[0] - xeq[1]),
  inputs=('Ld', 'u1', 'u2'), outputs=1, name='control')
```

The input to the controller is *u*, consisting of the vector of hare and lynx populations followed by the desired lynx population.

To connect the controller to the predatory-prey model, we create an *InterconnectedSystem*:

```
io_closed = control.InterconnectedSystem(
  (io_predprey, io_controller),        # systems
  connections=(
    ('predprey.u', 'control.y[0]'),
    ('control.u1',  'predprey.H'),
    ('control.u2',  'predprey.L')
  ),
  inplist=('control.Ld'),
  outlist=('predprey.H', 'predprey.L', 'control.y[0]')
)
```

Finally, we simulate the closed loop system:

```
# Simulate the system
t, y = control.input_output_response(io_closed, T, 30, [15, 20])

# Plot the response
plt.figure(2)
plt.subplot(2, 1, 1)
plt.plot(t, y[0])
plt.plot(t, y[1])
plt.legend(['Hare', 'Lynx'])
plt.subplot(2, 1, 2)
plt.plot(t, y[2])
plt.legend(['input'])
plt.show(block=False)
```

7.3 Module classes and functions

7.3.1 Input/output system classes

InputOutputSystem([inputs, outputs, states, …])	A class for representing input/output systems.
InterconnectedSystem(syslist[, connections, …])	Interconnection of a set of input/output systems.
LinearIOSystem(linsys[, inputs, outputs, …])	Input/output representation of a linear (state space) system.
NonlinearIOSystem(updfcn[, outfcn, inputs, …])	Nonlinear I/O system.

7.3.2 Input/output system functions

find_eqpt(sys, x0[, u0, y0, t, params, iu, …])	Find the equilibrium point for an input/output system.
linearize(sys, xeq[, ueq, t, params])	Linearize an input/output system at a given state and input.
input_output_response(sys, T[, U, X0, …])	Compute the output response of a system to a given input.
ss2io(*args, **kw)	Create an I/O system from a state space linear system.
tf2io(*args, **kw)	Convert a transfer function into an I/O system

CHAPTER

EIGHT

EXAMPLES

The source code for the examples below are available in the *examples/* subdirecory of the source code distribution. The can also be accessed online via the [python-control GitHub repository](https://github.com/python-control/python-control/tree/master/examples).

8.1 Python scripts

The following Python scripts document the use of a variety of methods in the Python Control Toolbox on examples drawn from standard control textbooks and other sources.

8.1.1 Secord order system (MATLAB module example)

This example computes time and frequency responses for a second-order system using the MATLAB compatibility module.

Code

```
# secord.py - demonstrate some standard MATLAB commands
# RMM, 25 May 09

import os
import matplotlib.pyplot as plt   # MATLAB plotting functions
from control.matlab import *  # MATLAB-like functions

# Parameters defining the system
m = 250.0           # system mass
k = 40.0            # spring constant
b = 60.0            # damping constant

# System matrices
A = [[0, 1.], [-k/m, -b/m]]
B = [[0], [1/m]]
C = [[1., 0]]
sys = ss(A, B, C, 0)

# Step response for the system
plt.figure(1)
yout, T = step(sys)
plt.plot(T.T, yout.T)
plt.show(block=False)
```

(continues on next page)

(continued from previous page)

```

# Bode plot for the system
plt.figure(2)
mag, phase, om = bode(sys, logspace(-2, 2), Plot=True)
plt.show(block=False)

# Nyquist plot for the system
plt.figure(3)
nyquist(sys, logspace(-2, 2))
plt.show(block=False)

# Root lcous plot for the system
rlocus(sys)

if 'PYCONTROL_TEST_EXAMPLES' not in os.environ:
    plt.show()
```

Notes

1. The environment variable *PYCONTROL_TEST_EXAMPLES* is used for testing to turn off plotting of the outputs.

8.1.2 Inner/outer control design for vertical takeoff and landing aircraft

This script demonstrates the use of the python-control package for analysis and design of a controller for a vectored thrust aircraft model that is used as a running example through the text Feedback Systems by Astrom and Murray. This example makes use of MATLAB compatible commands.

Code

```
# pvtol-nested.py - inner/outer design for vectored thrust aircraft
# RMM, 5 Sep 09
#
# This file works through a fairly complicated control design and
# analysis, corresponding to the planar vertical takeoff and landing
# (PVTOL) aircraft in Astrom and Murray, Chapter 11.  It is intended
# to demonstrate the basic functionality of the python-control
# package.
#

from __future__ import print_function

import os
import matplotlib.pyplot as plt  # MATLAB plotting functions
from control.matlab import *    # MATLAB-like functions
import numpy as np

# System parameters
m = 4                 # mass of aircraft
J = 0.0475            # inertia around pitch axis
r = 0.25              # distance to center of force
g = 9.8               # gravitational constant
c = 0.05              # damping factor (estimated)
```

(continues on next page)

(continued from previous page)

```

# Transfer functions for dynamics
Pi = tf([r], [J, 0, 0])  # inner loop (roll)
Po = tf([1], [m, c, 0])  # outer loop (position)

#
# Inner loop control design
#
# This is the controller for the pitch dynamics.  Goal is to have
# fast response for the pitch dynamics so that we can use this as a
# control for the lateral dynamics
#

# Design a simple lead controller for the system
k, a, b = 200, 2, 50
Ci = k*tf([1, a], [1, b])  # lead compensator
Li = Pi*Ci

# Bode plot for the open loop process
plt.figure(1)
bode(Pi)

# Bode plot for the loop transfer function, with margins
plt.figure(2)
bode(Li)

# Compute out the gain and phase margins
#! Not implemented
# gm, pm, wcg, wcp = margin(Li)

# Compute the sensitivity and complementary sensitivity functions
Si = feedback(1, Li)
Ti = Li*Si

# Check to make sure that the specification is met
plt.figure(3)
gangof4(Pi, Ci)

# Compute out the actual transfer function from u1 to v1 (see L8.2 notes)
# Hi = Ci*(1-m*g*Pi)/(1+Ci*Pi)
Hi = parallel(feedback(Ci, Pi), -m*g*feedback(Ci*Pi, 1))

plt.figure(4)
plt.clf()
plt.subplot(221)
bode(Hi)

# Now design the lateral control system
a, b, K = 0.02, 5, 2
Co = -K*tf([1, 0.3], [1, 10])  # another lead compensator
Lo = -m*g*Po*Co

plt.figure(5)
bode(Lo)  # margin(Lo)

# Finally compute the real outer-loop loop gain + responses
L = Co*Hi*Po
```

(continues on next page)

(continued from previous page)

```
S = feedback(1, L)
T = feedback(L, 1)

# Compute stability margins
gm, pm, wgc, wpc = margin(L)
print("Gain margin: %g at %g" % (gm, wgc))
print("Phase margin: %g at %g" % (pm, wpc))

plt.figure(6)
plt.clf()
bode(L, np.logspace(-4, 3))

# Add crossover line to the magnitude plot
#
# Note: in matplotlib before v2.1, the following code worked:
#
#   plt.subplot(211); hold(True);
#   loglog([1e-4, 1e3], [1, 1], 'k-')
#
# In later versions of matplotlib the call to plt.subplot will clear the
# axes and so we have to extract the axes that we want to use by hand.
# In addition, hold() is deprecated so we no longer require it.
#
for ax in plt.gcf().axes:
    if ax.get_label() == 'control-bode-magnitude':
        break
ax.semilogx([1e-4, 1e3], 20*np.log10([1, 1]), 'k-')

#
# Replot phase starting at -90 degrees
#
# Get the phase plot axes
for ax in plt.gcf().axes:
    if ax.get_label() == 'control-bode-phase':
        break

# Recreate the frequency response and shift the phase
mag, phase, w = freqresp(L, np.logspace(-4, 3))
phase = phase - 360

# Replot the phase by hand
ax.semilogx([1e-4, 1e3], [-180, -180], 'k-')
ax.semilogx(w, np.squeeze(phase), 'b-')
ax.axis([1e-4, 1e3, -360, 0])
plt.xlabel('Frequency [deg]')
plt.ylabel('Phase [deg]')
# plt.set(gca, 'YTick', [-360, -270, -180, -90, 0])
# plt.set(gca, 'XTick', [10^-4, 10^-2, 1, 100])

#
# Nyquist plot for complete design
#
plt.figure(7)
plt.clf()
nyquist(L, (0.0001, 1000))
plt.axis([-700, 5300, -3000, 3000])

```

(continues on next page)

(continued from previous page)

```
138 # Add a box in the region we are going to expand
139 plt.plot([-400, -400, 200, 200, -400], [-100, 100, 100, -100, -100], 'r-')
140 
141 # Expanded region
142 plt.figure(8)
143 plt.clf()
144 plt.subplot(231)
145 nyquist(L)
146 plt.axis([-10, 5, -20, 20])
147 
148 # set up the color
149 color = 'b'
150 
151 # Add arrows to the plot
152 # H1 = L.evalfr(0.4); H2 = L.evalfr(0.41);
153 # arrow([real(H1), imag(H1)], [real(H2), imag(H2)], AM_normal_arrowsize, \
154 #  'EdgeColor', color, 'FaceColor', color);
155 
156 # H1 = freqresp(L, 0.35); H2 = freqresp(L, 0.36);
157 # arrow([real(H2), -imag(H2)], [real(H1), -imag(H1)], AM_normal_arrowsize, \
158 #  'EdgeColor', color, 'FaceColor', color);
159 
160 plt.figure(9)
161 Yvec, Tvec = step(T, np.linspace(0, 20))
162 plt.plot(Tvec.T, Yvec.T)
163 
164 Yvec, Tvec = step(Co*S, np.linspace(0, 20))
165 plt.plot(Tvec.T, Yvec.T)
166 
167 plt.figure(10)
168 plt.clf()
169 P, Z = pzmap(T, plot=True, grid=True)
170 print("Closed loop poles and zeros: ", P, Z)
171 
172 # Gang of Four
173 plt.figure(11)
174 plt.clf()
175 gangof4(Hi*Po, Co)
176 
177 if 'PYCONTROL_TEST_EXAMPLES' not in os.environ:
178     plt.show()
```

Notes

1. Importing *print_function* from *__future__* in line 11 is only required if using Python 2.7.
2. The environment variable *PYCONTROL_TEST_EXAMPLES* is used for testing to turn off plotting of the outputs.

8.1.3 LQR control design for vertical takeoff and landing aircraft

This script demonstrates the use of the python-control package for analysis and design of a controller for a vectored thrust aircraft model that is used as a running example through the text Feedback Systems by Astrom and Murray. This example makes use of MATLAB compatible commands.

Code

```python
# pvtol_lqr.m - LQR design for vectored thrust aircraft
# RMM, 14 Jan 03
#
# This file works through an LQR based design problem, using the
# planar vertical takeoff and landing (PVTOL) aircraft example from
# Astrom and Murray, Chapter 5.  It is intended to demonstrate the
# basic functionality of the python-control package.
#

import os
import numpy as np
import matplotlib.pyplot as plt  # MATLAB plotting functions
from control.matlab import *  # MATLAB-like functions

#
# System dynamics
#
# These are the dynamics for the PVTOL system, written in state space
# form.
#

# System parameters
m = 4       # mass of aircraft
J = 0.0475  # inertia around pitch axis
r = 0.25    # distance to center of force
g = 9.8     # gravitational constant
c = 0.05    # damping factor (estimated)

# State space dynamics
xe = [0, 0, 0, 0, 0, 0]  # equilibrium point of interest
ue = [0, m*g]  # (note these are lists, not matrices)

# TODO: The following objects need converting from np.matrix to np.array
# This will involve re-working the subsequent equations as the shapes
# See below.

# Dynamics matrix (use matrix type so that * works for multiplication)
A = np.matrix(
    [[0, 0, 0, 1, 0, 0],
     [0, 0, 0, 0, 1, 0],
     [0, 0, 0, 0, 0, 1],
     [0, 0, (-ue[0]*np.sin(xe[2]) - ue[1]*np.cos(xe[2]))/m, -c/m, 0, 0],
     [0, 0, (ue[0]*np.cos(xe[2]) - ue[1]*np.sin(xe[2]))/m, 0, -c/m, 0],
     [0, 0, 0, 0, 0, 0]]
)

# Input matrix
B = np.matrix(
```

(continues on next page)

(continued from previous page)

```
    [[0, 0], [0, 0], [0, 0],
     [np.cos(xe[2])/m, -np.sin(xe[2])/m],
     [np.sin(xe[2])/m, np.cos(xe[2])/m],
     [r/J, 0]]
)

# Output matrix
C = np.matrix([[1, 0, 0, 0, 0, 0], [0, 1, 0, 0, 0, 0]])
D = np.matrix([[0, 0], [0, 0]])

#
# Construct inputs and outputs corresponding to steps in xy position
#
# The vectors xd and yd correspond to the states that are the desired
# equilibrium states for the system.  The matrices Cx and Cy are the
# corresponding outputs.
#
# The way these vectors are used is to compute the closed loop system
# dynamics as
#
#   xdot = Ax + B u  =>  xdot = (A-BK)x + K xd
#      u = -K(x - xd)       y = Cx
#
# The closed loop dynamics can be simulated using the "step" command,
# with K*xd as the input vector (assumes that the "input" is unit size,
# so that xd corresponds to the desired steady state.
#

xd = np.matrix([[1], [0], [0], [0], [0], [0]])
yd = np.matrix([[0], [1], [0], [0], [0], [0]])

#
# Extract the relevant dynamics for use with SISO library
#
# The current python-control library only supports SISO transfer
# functions, so we have to modify some parts of the original MATLAB
# code to extract out SISO systems.  To do this, we define the 'lat' and
# 'alt' index vectors to consist of the states that are are relevant
# to the lateral (x) and vertical (y) dynamics.
#

# Indices for the parts of the state that we want
lat = (0, 2, 3, 5)
alt = (1, 4)

# Decoupled dynamics
Ax = (A[lat, :])[:, lat]  # ! not sure why I have to do it this way
Bx = B[lat, 0]
Cx = C[0, lat]
Dx = D[0, 0]

Ay = (A[alt, :])[:, alt]  # ! not sure why I have to do it this way
By = B[alt, 1]
Cy = C[1, alt]
Dy = D[1, 1]

# Label the plot
```

(continues on next page)

(continued from previous page)

```
plt.clf()
plt.suptitle("LQR controllers for vectored thrust aircraft (pvtol-lqr)")

#
# LQR design
#

# Start with a diagonal weighting
Qx1 = np.diag([1, 1, 1, 1, 1, 1])
Qu1a = np.diag([1, 1])
K, X, E = lqr(A, B, Qx1, Qu1a)
K1a = np.matrix(K)

# Close the loop: xdot = Ax - B K (x-xd)
# Note: python-control requires we do this 1 input at a time
# H1a = ss(A-B*K1a, B*K1a*concatenate((xd, yd), axis=1), C, D);
# (T, Y) = step(H1a, T=np.linspace(0,10,100));

# TODO: The following equations will need modifying when converting from np.matrix to np.array
# because the results and even intermediate calculations will be different with numpy arrays
# For example:
# Bx = B[lat, 0]
# Will need to be changed to:
# Bx = B[lat, 0].reshape(-1, 1)
# (if we want it to have the same shape as before)

# For reference, here is a list of the correct shapes of these objects:
# A: (6, 6)
# B: (6, 2)
# C: (2, 6)
# D: (2, 2)
# xd: (6, 1)
# yd: (6, 1)
# Ax: (4, 4)
# Bx: (4, 1)
# Cx: (1, 4)
# Dx: ()
# Ay: (2, 2)
# By: (2, 1)
# Cy: (1, 2)

# Step response for the first input
H1ax = ss(Ax - Bx*K1a[0, lat], Bx*K1a[0, lat]*xd[lat, :], Cx, Dx)
Yx, Tx = step(H1ax, T=np.linspace(0, 10, 100))

# Step response for the second input
H1ay = ss(Ay - By*K1a[1, alt], By*K1a[1, alt]*yd[alt, :], Cy, Dy)
Yy, Ty = step(H1ay, T=np.linspace(0, 10, 100))

plt.subplot(221)
plt.title("Identity weights")
# plt.plot(T, Y[:,1, 1], '-', T, Y[:,2, 2], '--')
plt.plot(Tx.T, Yx.T, '-', Ty.T, Yy.T, '--')
plt.plot([0, 10], [1, 1], 'k-')

```

(continues on next page)

(continued from previous page)

```
plt.axis([0, 10, -0.1, 1.4])
plt.ylabel('position')
plt.legend(('x', 'y'), loc='lower right')

# Look at different input weightings
Qu1a = np.diag([1, 1])
K1a, X, E = lqr(A, B, Qx1, Qu1a)
H1ax = ss(Ax - Bx*K1a[0, lat], Bx*K1a[0, lat]*xd[lat, :], Cx, Dx)

Qu1b = (40 ** 2)*np.diag([1, 1])
K1b, X, E = lqr(A, B, Qx1, Qu1b)
H1bx = ss(Ax - Bx*K1b[0, lat], Bx*K1b[0, lat]*xd[lat, :], Cx, Dx)

Qu1c = (200 ** 2)*np.diag([1, 1])
K1c, X, E = lqr(A, B, Qx1, Qu1c)
H1cx = ss(Ax - Bx*K1c[0, lat], Bx*K1c[0, lat]*xd[lat, :], Cx, Dx)

[Y1, T1] = step(H1ax, T=np.linspace(0, 10, 100))
[Y2, T2] = step(H1bx, T=np.linspace(0, 10, 100))
[Y3, T3] = step(H1cx, T=np.linspace(0, 10, 100))

plt.subplot(222)
plt.title("Effect of input weights")
plt.plot(T1.T, Y1.T, 'b-')
plt.plot(T2.T, Y2.T, 'b-')
plt.plot(T3.T, Y3.T, 'b-')
plt.plot([0, 10], [1, 1], 'k-')

plt.axis([0, 10, -0.1, 1.4])

# arcarrow([1.3, 0.8], [5, 0.45], -6)
plt.text(5.3, 0.4, 'rho')

# Output weighting - change Qx to use outputs
Qx2 = C.T*C
Qu2 = 0.1*np.diag([1, 1])
K, X, E = lqr(A, B, Qx2, Qu2)
K2 = np.matrix(K)

H2x = ss(Ax - Bx*K2[0, lat], Bx*K2[0, lat]*xd[lat, :], Cx, Dx)
H2y = ss(Ay - By*K2[1, alt], By*K2[1, alt]*yd[alt, :], Cy, Dy)

plt.subplot(223)
plt.title("Output weighting")
[Y2x, T2x] = step(H2x, T=np.linspace(0, 10, 100))
[Y2y, T2y] = step(H2y, T=np.linspace(0, 10, 100))
plt.plot(T2x.T, Y2x.T, T2y.T, Y2y.T)
plt.ylabel('position')
plt.xlabel('time')
plt.ylabel('position')
plt.legend(('x', 'y'), loc='lower right')

#
# Physically motivated weighting
#
# Shoot for 1 cm error in x, 10 cm error in y.  Try to keep the angle
# less than 5 degrees in making the adjustments.  Penalize side forces
```

(continues on next page)

(continued from previous page)

```
# due to loss in efficiency.
#

Qx3 = np.diag([100, 10, 2*np.pi/5, 0, 0, 0])
Qu3 = 0.1*np.diag([1, 10])
(K, X, E) = lqr(A, B, Qx3, Qu3)
K3 = np.matrix(K)

H3x = ss(Ax - Bx*K3[0, lat], Bx*K3[0, lat]*xd[lat, :], Cx, Dx)
H3y = ss(Ay - By*K3[1, alt], By*K3[1, alt]*yd[alt, :], Cy, Dy)
plt.subplot(224)
# step(H3x, H3y, 10)
[Y3x, T3x] = step(H3x, T=np.linspace(0, 10, 100))
[Y3y, T3y] = step(H3y, T=np.linspace(0, 10, 100))
plt.plot(T3x.T, Y3x.T, T3y.T, Y3y.T)
plt.title("Physically motivated weights")
plt.xlabel('time')
plt.legend(('x', 'y'), loc='lower right')

if 'PYCONTROL_TEST_EXAMPLES' not in os.environ:
    plt.show()
```

Notes

1. The environment variable *PYCONTROL_TEST_EXAMPLES* is used for testing to turn off plotting of the outputs.

8.1.4 Balanced model reduction examples

Code

```
#!/usr/bin/env python

import os

import numpy as np
import control.modelsimp as msimp
import control.matlab as mt
from control.statesp import StateSpace
import matplotlib.pyplot as plt

plt.close('all')

# controllable canonical realization computed in MATLAB for the
# transfer function: num = [1 11 45 32], den = [1 15 60 200 60]
A = np.array([
    [-15., -7.5, -6.25, -1.875],
    [8., 0., 0., 0.],
    [0., 4., 0., 0.],
    [0., 0., 1., 0.]
])
B = np.array([
    [2.],
    [0.],
```

(continues on next page)

(continued from previous page)

```
    [0.],
    [0.]
])
C = np.array([[0.5, 0.6875, 0.7031, 0.5]])
D = np.array([[0.]])

# The full system
fsys = StateSpace(A, B, C, D)

# The reduced system, truncating the order by 1
n = 3
rsys = msimp.balred(fsys, n, method='truncate')

# Comparison of the step responses of the full and reduced systems
plt.figure(1)
y, t = mt.step(fsys)
yr, tr = mt.step(rsys)
plt.plot(t.T, y.T)
plt.plot(tr.T, yr.T)

# Repeat balanced reduction, now with 100-dimensional random state space
sysrand = mt.rss(100, 1, 1)
rsysrand = msimp.balred(sysrand, 10, method='truncate')

# Comparison of the impulse responses of the full and reduced random systems
plt.figure(2)
yrand, trand = mt.impulse(sysrand)
yrandr, trandr = mt.impulse(rsysrand)
plt.plot(trand.T, yrand.T, trandr.T, yrandr.T)

if 'PYCONTROL_TEST_EXAMPLES' not in os.environ:
    plt.show()
```

Notes

1. The environment variable *PYCONTROL_TEST_EXAMPLES* is used for testing to turn off plotting of the outputs.

8.1.5 Phase plot examples

Code

```
# phaseplots.py - examples of phase portraits
# RMM, 24 July 2011
#
# This file contains examples of phase portraits pulled from "Feedback
# Systems" by Astrom and Murray (Princeton University Press, 2008).

import os

import numpy as np
import matplotlib.pyplot as plt
from control.phaseplot import phase_plot
from numpy import pi
```

(continues on next page)

(continued from previous page)

```

# Clear out any figures that are present
plt.close('all')

#
# Inverted pendulum
#

# Define the ODEs for a damped (inverted) pendulum
def invpend_ode(x, t, m=1., l=1., b=0.2, g=1):
    return x[1], -b/m*x[1] + (g*l/m)*np.sin(x[0])


# Set up the figure the way we want it to look
plt.figure()
plt.clf()
plt.axis([-2*pi, 2*pi, -2.1, 2.1])
plt.title('Inverted pendulum')

# Outer trajectories
phase_plot(
    invpend_ode,
    X0=[[-2*pi, 1.6], [-2*pi, 0.5], [-1.8, 2.1],
        [-1, 2.1], [4.2, 2.1], [5, 2.1],
        [2*pi, -1.6], [2*pi, -0.5], [1.8, -2.1],
        [1, -2.1], [-4.2, -2.1], [-5, -2.1]],
    T=np.linspace(0, 40, 200),
    logtime=(3, 0.7)
)

# Separatrices
phase_plot(invpend_ode, X0=[[-2.3056, 2.1], [2.3056, -2.1]], T=6, lingrid=0)

#
# Systems of ODEs: damped oscillator example (simulation + phase portrait)
#

def oscillator_ode(x, t, m=1., b=1, k=1):
    return x[1], -k/m*x[0] - b/m*x[1]


# Generate a vector plot for the damped oscillator
plt.figure()
plt.clf()
phase_plot(oscillator_ode, [-1, 1, 10], [-1, 1, 10], 0.15)
#plt.plot([0], [0], '.')
# a=gca; set(a,'FontSize',20); set(a,'DataAspectRatio',[1,1,1])
plt.xlabel('$x_1$')
plt.ylabel('$x_2$')
plt.title('Damped oscillator, vector field')

# Generate a phase plot for the damped oscillator
plt.figure()
plt.clf()
plt.axis([-1, 1, -1, 1])  # set(gca, 'DataAspectRatio', [1, 1, 1]);
phase_plot(
    oscillator_ode,
```

(continues on next page)

(continued from previous page)

```
    X0=[
        [-1, 1], [-0.3, 1], [0, 1], [0.25, 1], [0.5, 1], [0.75, 1], [1, 1],
        [1, -1], [0.3, -1], [0, -1], [-0.25, -1], [-0.5, -1], [-0.75, -1], [-1, -1]
    ],
    T=np.linspace(0, 8, 80),
    timepts=[0.25, 0.8, 2, 3]
)
plt.plot([0], [0], 'k.')  # 'MarkerSize', AM_data_markersize*3)
# set(gca, 'DataAspectRatio', [1,1,1])
plt.xlabel('$x_1$')
plt.ylabel('$x_2$')
plt.title('Damped oscillator, vector field and stream lines')

#
# Stability definitions
#
# This set of plots illustrates the various types of equilibrium points.
#


def saddle_ode(x, t):
    """Saddle point vector field"""
    return x[0] - 3*x[1], -3*x[0] + x[1]


# Asy stable
m = 1
b = 1
k = 1  # default values
plt.figure()
plt.clf()
plt.axis([-1, 1, -1, 1])  # set(gca, 'DataAspectRatio', [1 1 1]);
phase_plot(
    oscillator_ode,
    X0=[
        [-1, 1], [-0.3, 1], [0, 1], [0.25, 1], [0.5, 1], [0.7, 1], [1, 1], [1.3, 1],
        [1, -1], [0.3, -1], [0, -1], [-0.25, -1], [-0.5, -1], [-0.7, -1], [-1, -1],
        [-1.3, -1]
    ],
    T=np.linspace(0, 10, 100),
    timepts=[0.3, 1, 2, 3],
    parms=(m, b, k)
)
plt.plot([0], [0], 'k.')  # 'MarkerSize', AM_data_markersize*3)
# plt.set(gca,'FontSize', 16)
plt.xlabel('$x_1$')
plt.ylabel('$x_2$')
plt.title('Asymptotically stable point')

# Saddle
plt.figure()
plt.clf()
plt.axis([-1, 1, -1, 1])  # set(gca, 'DataAspectRatio', [1 1 1])
phase_plot(
    saddle_ode,
    scale=2,
    timepts=[0.2, 0.5, 0.8],
```

(continues on next page)

(continued from previous page)

```
    X0=[
        [-1, -1], [1, 1],
        [-1, -0.95], [-1, -0.9], [-1, -0.8], [-1, -0.6], [-1, -0.4], [-1, -0.2],
        [-0.95, -1], [-0.9, -1], [-0.8, -1], [-0.6, -1], [-0.4, -1], [-0.2, -1],
        [1, 0.95], [1, 0.9], [1, 0.8], [1, 0.6], [1, 0.4], [1, 0.2],
        [0.95, 1], [0.9, 1], [0.8, 1], [0.6, 1], [0.4, 1], [0.2, 1],
        [-0.5, -0.45], [-0.45, -0.5], [0.5, 0.45], [0.45, 0.5],
        [-0.04, 0.04], [0.04, -0.04]
    ],
    T=np.linspace(0, 2, 20)
)
plt.plot([0], [0], 'k.')  # 'MarkerSize', AM_data_markersize*3)
# set(gca,'FontSize', 16)
plt.xlabel('$x_1$')
plt.ylabel('$x_2$')
plt.title('Saddle point')

# Stable isL
m = 1
b = 0
k = 1  # zero damping
plt.figure()
plt.clf()
plt.axis([-1, 1, -1, 1])  # set(gca, 'DataAspectRatio', [1 1 1]);
phase_plot(
    oscillator_ode,
    timepts=[pi/6, pi/3, pi/2, 2*pi/3, 5*pi/6, pi, 7*pi/6,
             4*pi/3, 9*pi/6, 5*pi/3, 11*pi/6, 2*pi],
    X0=[[0.2, 0], [0.4, 0], [0.6, 0], [0.8, 0], [1, 0], [1.2, 0], [1.4, 0]],
    T=np.linspace(0, 20, 200),
    parms=(m, b, k)
)
plt.plot([0], [0], 'k.')  # 'MarkerSize', AM_data_markersize*3)
# plt.set(gca,'FontSize', 16)
plt.xlabel('$x_1$')
plt.ylabel('$x_2$')
plt.title('Undamped system\nLyapunov stable, not asympt. stable')

if 'PYCONTROL_TEST_EXAMPLES' not in os.environ:
    plt.show()
```

Notes

1. The environment variable *PYCONTROL_TEST_EXAMPLES* is used for testing to turn off plotting of the outputs.

8.1.6 SISO robust control example (SP96, Example 2.1)

Code

```
"""robust_siso.py

Demonstrate mixed-sensitivity H-infinity design for a SISO plant.

Based on Example 2.11 from Multivariable Feedback Control, Skogestad
and Postlethwaite, 1st Edition.
"""

import os

import numpy as np
import matplotlib.pyplot as plt

from control import tf, mixsyn, feedback, step_response

s = tf([1, 0], 1)
# the plant
g = 200/(10*s + 1) / (0.05*s + 1)**2
# disturbance plant
gd = 100/(10*s + 1)

# first design
# sensitivity weighting
M = 1.5
wb = 10
A = 1e-4
ws1 = (s/M + wb) / (s + wb*A)
# KS weighting
wu = tf(1, 1)

k1, cl1, info1 = mixsyn(g, ws1, wu)

# sensitivity (S) and complementary sensitivity (T) functions for
# design 1
s1 = feedback(1, g*k1)
t1 = feedback(g*k1, 1)

# second design
# this weighting differs from the text, where A**0.5 is used; if you use that,
# the frequency response doesn't match the figure.  The time responses
# are similar, though.
ws2 = (s/M ** 0.5 + wb)**2 / (s + wb*A)**2
# the KS weighting is the same as for the first design

k2, cl2, info2 = mixsyn(g, ws2, wu)

# S and T for design 2
s2 = feedback(1, g*k2)
t2 = feedback(g*k2, 1)

# frequency response
omega = np.logspace(-2, 2, 101)
ws1mag, _, _ = ws1.freqresp(omega)
```

(continues on next page)

(continued from previous page)

```
s1mag, _, _ = s1.freqresp(omega)
ws2mag, _, _ = ws2.freqresp(omega)
s2mag, _, _ = s2.freqresp(omega)

plt.figure(1)
# text uses log-scaled absolute, but dB are probably more familiar to most control␣
↪engineers
plt.semilogx(omega, 20*np.log10(s1mag.flat), label='$S_1$')
plt.semilogx(omega, 20*np.log10(s2mag.flat), label='$S_2$')
# -1 in logspace is inverse
plt.semilogx(omega, -20*np.log10(ws1mag.flat), label='$1/w_{P1}$')
plt.semilogx(omega, -20*np.log10(ws2mag.flat), label='$1/w_{P2}$')

plt.ylim([-80, 10])
plt.xlim([1e-2, 1e2])
plt.xlabel('freq [rad/s]')
plt.ylabel('mag [dB]')
plt.legend()
plt.title('Sensitivity and sensitivity weighting frequency responses')

# time response
time = np.linspace(0, 3, 201)
_, y1 = step_response(t1, time)
_, y2 = step_response(t2, time)

# gd injects into the output (that is, g and gd are summed), and the
# closed loop mapping from output disturbance->output is S.
_, y1d = step_response(s1*gd, time)
_, y2d = step_response(s2*gd, time)

plt.figure(2)
plt.subplot(1, 2, 1)
plt.plot(time, y1, label='$y_1(t)$')
plt.plot(time, y2, label='$y_2(t)$')

plt.ylim([-0.1, 1.5])
plt.xlim([0, 3])
plt.xlabel('time [s]')
plt.ylabel('signal [1]')
plt.legend()
plt.title('Tracking response')

plt.subplot(1, 2, 2)
plt.plot(time, y1d, label='$y_1(t)$')
plt.plot(time, y2d, label='$y_2(t)$')

plt.ylim([-0.1, 1.5])
plt.xlim([0, 3])
plt.xlabel('time [s]')
plt.ylabel('signal [1]')
plt.legend()
plt.title('Disturbance response')

if 'PYCONTROL_TEST_EXAMPLES' not in os.environ:
    plt.show()
```

Notes

1. The environment variable *PYCONTROL_TEST_EXAMPLES* is used for testing to turn off plotting of the outputs.

8.1.7 MIMO robust control example (SP96, Example 3.8)

Code

```
"""robust_mimo.py

Demonstrate mixed-sensitivity H-infinity design for a MIMO plant.

Based on Example 3.8 from Multivariable Feedback Control, Skogestad and Postlethwaite,
↪ 1st Edition.
"""

import os

import numpy as np
import matplotlib.pyplot as plt

from control import tf, ss, mixsyn, step_response


def weighting(wb, m, a):
    """weighting(wb,m,a) -> wf
    wb - design frequency (where |wf| is approximately 1)
    m - high frequency gain of 1/wf; should be > 1
    a - low frequency gain of 1/wf; should be < 1
    wf - SISO LTI object
    """
    s = tf([1, 0], [1])
    return (s/m + wb) / (s + wb*a)


def plant():
    """plant() -> g
    g - LTI object; 2x2 plant with a RHP zero, at s=0.5.
    """
    den = [0.2, 1.2, 1]
    gtf = tf([[[1], [1]],
              [[2, 1], [2]]],
             [[den, den],
              [den, den]])
    return ss(gtf)


# as of this writing (2017-07-01), python-control doesn't have an
# equivalent to Matlab's sigma function, so use a trivial stand-in.
def triv_sigma(g, w):
    """triv_sigma(g,w) -> s
    g - LTI object, order n
    w - frequencies, length m
    s - (m,n) array of singular values of g(1j*w)"""
    m, p, _ = g.freqresp(w)
    sjw = (m*np.exp(1j*p)).transpose(2, 0, 1)
```

(continues on next page)

(continued from previous page)

```
    sv = np.linalg.svd(sjw, compute_uv=False)
    return sv


def analysis():
    """Plot open-loop responses for various inputs"""
    g = plant()

    t = np.linspace(0, 10, 101)
    _, yu1 = step_response(g, t, input=0)
    _, yu2 = step_response(g, t, input=1)

    yu1 = yu1
    yu2 = yu2

    # linear system, so scale and sum previous results to get the
    # [1,-1] response
    yuz = yu1 - yu2

    plt.figure(1)
    plt.subplot(1, 3, 1)
    plt.plot(t, yu1[0], label='$y_1$')
    plt.plot(t, yu1[1], label='$y_2$')
    plt.xlabel('time')
    plt.ylabel('output')
    plt.ylim([-1.1, 2.1])
    plt.legend()
    plt.title('o/l response\nto input [1,0]')

    plt.subplot(1, 3, 2)
    plt.plot(t, yu2[0], label='$y_1$')
    plt.plot(t, yu2[1], label='$y_2$')
    plt.xlabel('time')
    plt.ylabel('output')
    plt.ylim([-1.1, 2.1])
    plt.legend()
    plt.title('o/l response\nto input [0,1]')

    plt.subplot(1, 3, 3)
    plt.plot(t, yuz[0], label='$y_1$')
    plt.plot(t, yuz[1], label='$y_2$')
    plt.xlabel('time')
    plt.ylabel('output')
    plt.ylim([-1.1, 2.1])
    plt.legend()
    plt.title('o/l response\nto input [1,-1]')


def synth(wb1, wb2):
    """synth(wb1,wb2) -> k,gamma
    wb1: S weighting frequency
    wb2: KS weighting frequency
    k: controller
    gamma: H-infinity norm of 'design', that is, of evaluation system
    with loop closed through design
    """
    g = plant()
```

(continues on next page)

(continued from previous page)

```
    wu = ss([], [], [], np.eye(2))
    wp1 = ss(weighting(wb=wb1, m=1.5, a=1e-4))
    wp2 = ss(weighting(wb=wb2, m=1.5, a=1e-4))
    wp = wp1.append(wp2)
    k, _, info = mixsyn(g, wp, wu)
    return k, info[0]


def step_opposite(g, t):
    """reponse to step of [-1,1]"""
    _, yu1 = step_response(g, t, input=0)
    _, yu2 = step_response(g, t, input=1)
    return yu1 - yu2


def design():
    """Show results of designs"""
    # equal weighting on each output
    k1, gam1 = synth(0.25, 0.25)
    # increase "bandwidth" of output 2 by moving crossover weighting frequency 100␣
↪times higher
    k2, gam2 = synth(0.25, 25)
    # now weight output 1 more heavily
    # won't plot this one, just want gamma
    _, gam3 = synth(25, 0.25)

    print('design 1 gamma {:.3g} (Skogestad: 2.80)'.format(gam1))
    print('design 2 gamma {:.3g} (Skogestad: 2.92)'.format(gam2))
    print('design 3 gamma {:.3g} (Skogestad: 6.73)'.format(gam3))

    # do the designs
    g = plant()
    w = np.logspace(-2, 2, 101)
    I = ss([], [], [], np.eye(2))
    s1 = I.feedback(g*k1)
    s2 = I.feedback(g*k2)

    # frequency response
    sv1 = triv_sigma(s1, w)
    sv2 = triv_sigma(s2, w)

    plt.figure(2)

    plt.subplot(1, 2, 1)
    plt.semilogx(w, 20*np.log10(sv1[:, 0]), label=r'$\sigma_1(S_1)$')
    plt.semilogx(w, 20*np.log10(sv1[:, 1]), label=r'$\sigma_2(S_1)$')
    plt.semilogx(w, 20*np.log10(sv2[:, 0]), label=r'$\sigma_1(S_2)$')
    plt.semilogx(w, 20*np.log10(sv2[:, 1]), label=r'$\sigma_2(S_2)$')
    plt.ylim([-60, 10])
    plt.ylabel('magnitude [dB]')
    plt.xlim([1e-2, 1e2])
    plt.xlabel('freq [rad/s]')
    plt.legend()
    plt.title('Singular values of S')

    # time response

```

(continues on next page)

(continued from previous page)

```
    # in design 1, both outputs have an inverse initial response; in
    # design 2, output 2 does not, and is very fast, while output 1
    # has a larger initial inverse response than in design 1
    time = np.linspace(0, 10, 301)
    t1 = (g*k1).feedback(I)
    t2 = (g*k2).feedback(I)

    y1 = step_opposite(t1, time)
    y2 = step_opposite(t2, time)

    plt.subplot(1, 2, 2)
    plt.plot(time, y1[0], label='des. 1 $y_1(t))$')
    plt.plot(time, y1[1], label='des. 1 $y_2(t))$')
    plt.plot(time, y2[0], label='des. 2 $y_1(t))$')
    plt.plot(time, y2[1], label='des. 2 $y_2(t))$')
    plt.xlabel('time [s]')
    plt.ylabel('response [1]')
    plt.legend()
    plt.title('c/l response to reference [1,-1]')


if __name__ == "__main__":
    analysis()
    design()
    if 'PYCONTROL_TEST_EXAMPLES' not in os.environ:
        plt.show()
```

Notes

1. The environment variable *PYCONTROL_TEST_EXAMPLES* is used for testing to turn off plotting of the outputs.

8.1.8 Cruise control design example (as a nonlinear I/O system)

Code

```
# cruise-control.py - Cruise control example from FBS
# RMM, 16 May 2019
#
# The cruise control system of a car is a common feedback system encountered
# in everyday life. The system attempts to maintain a constant velocity in the
# presence of disturbances primarily caused by changes in the slope of a
# road. The controller compensates for these unknowns by measuring the speed
# of the car and adjusting the throttle appropriately.
#
# This file explore the dynamics and control of the cruise control system,
# following the material presenting in Feedback Systems by Astrom and Murray.
# A full nonlinear model of the vehicle dynamics is used, with both PI and
# state space control laws.  Different methods of constructing control systems
# are show, all using the InputOutputSystem class (and subclasses).

import numpy as np
import matplotlib.pyplot as plt
from math import pi
```

(continues on next page)

(continued from previous page)

```python
import control as ct

#
# Section 4.1: Cruise control modeling and control
#

# Vehicle model: vehicle()
#
# To develop a mathematical model we start with a force balance for
# the car body. Let v be the speed of the car, m the total mass
# (including passengers), F the force generated by the contact of the
# wheels with the road, and Fd the disturbance force due to gravity,
# friction, and aerodynamic drag.

def vehicle_update(t, x, u, params={}):
    """Vehicle dynamics for cruise control system.

    Parameters
    ----------
    x : array
         System state: car velocity in m/s
    u : array
         System input: [throttle, gear, road_slope], where throttle is
         a float between 0 and 1, gear is an integer between 1 and 5,
         and road_slope is in rad.

    Returns
    -------
    float
        Vehicle acceleration

    """
    from math import copysign, sin
    sign = lambda x: copysign(1, x)         # define the sign() function

    # Set up the system parameters
    m = params.get('m', 1600.)
    g = params.get('g', 9.8)
    Cr = params.get('Cr', 0.01)
    Cd = params.get('Cd', 0.32)
    rho = params.get('rho', 1.3)
    A = params.get('A', 2.4)
    alpha = params.get(
        'alpha', [40, 25, 16, 12, 10])      # gear ratio / wheel radius

    # Define variables for vehicle state and inputs
    v = x[0]                           # vehicle velocity
    throttle = np.clip(u[0], 0, 1)     # vehicle throttle
    gear = u[1]                        # vehicle gear
    theta = u[2]                       # road slope

    # Force generated by the engine

    omega = alpha[int(gear)-1] * v      # engine angular speed
    F = alpha[int(gear)-1] * motor_torque(omega, params) * throttle

    # Disturbance forces
```

(continues on next page)

(continued from previous page)

```
    #
    # The disturbance force Fd has three major components: Fg, the forces due
    # to gravity; Fr, the forces due to rolling friction; and Fa, the
    # aerodynamic drag.

    # Letting the slope of the road be \theta (theta), gravity gives the
    # force Fg = m g sin \theta.

    Fg = m * g * sin(theta)

    # A simple model of rolling friction is Fr = m g Cr sgn(v), where Cr is
    # the coefficient of rolling friction and sgn(v) is the sign of v (+/- 1) or
    # zero if v = 0.

    Fr  = m * g * Cr * sign(v)

    # The aerodynamic drag is proportional to the square of the speed: Fa =
    # 1/\rho Cd A |v| v, where \rho is the density of air, Cd is the
    # shape-dependent aerodynamic drag coefficient, and A is the frontal area
    # of the car.

    Fa = 1/2 * rho * Cd * A * abs(v) * v

    # Final acceleration on the car
    Fd = Fg + Fr + Fa
    dv = (F - Fd) / m

    return dv

# Engine model: motor_torque
#
# The force F is generated by the engine, whose torque is proportional to
# the rate of fuel injection, which is itself proportional to a control
# signal 0 <= u <= 1 that controls the throttle position. The torque also
# depends on engine speed omega.

def motor_torque(omega, params={}):
    # Set up the system parameters
    Tm = params.get('Tm', 190.)             # engine torque constant
    omega_m = params.get('omega_m', 420.)   # peak engine angular speed
    beta = params.get('beta', 0.4)          # peak engine rolloff

    return np.clip(Tm * (1 - beta * (omega/omega_m - 1)**2), 0, None)

# Define the input/output system for the vehicle
vehicle = ct.NonlinearIOSystem(
    vehicle_update, None, name='vehicle',
    inputs = ('u', 'gear', 'theta'), outputs = ('v'), states=('v'))

# Figure 1.11: A feedback system for controlling the speed of a vehicle. In
# this example, the speed of the vehicle is measured and compared to the
# desired speed.  The controller is a PI controller represented as a transfer
# function.  In the textbook, the simulations are done for LTI systems, but
# here we simulate the full nonlinear system.

# Construct a PI controller with rolloff, as a transfer function
Kp = 0.5                        # proportional gain
```

(continues on next page)

(continued from previous page)

```
Ki = 0.1                        # integral gain
control_tf = ct.tf2io(
    ct.TransferFunction([Kp, Ki], [1, 0.01*Ki/Kp]),
    name='control', inputs='u', outputs='y')

# Construct the closed loop control system
# Inputs: vref, gear, theta
# Outputs: v (vehicle velocity)
cruise_tf = ct.InterconnectedSystem(
    (control_tf, vehicle), name='cruise',
    connections = (
        ('control.u', '-vehicle.v'),
        ('vehicle.u', 'control.y')),
    inplist = ('control.u', 'vehicle.gear', 'vehicle.theta'),
    inputs = ('vref', 'gear', 'theta'),
    outlist = ('vehicle.v', 'vehicle.u'),
    outputs = ('v', 'u'))

# Define the time and input vectors
T = np.linspace(0, 25, 101)
vref = 20 * np.ones(T.shape)
gear = 4 * np.ones(T.shape)
theta0 = np.zeros(T.shape)

# Now simulate the effect of a hill at t = 5 seconds
plt.figure()
plt.suptitle('Response to change in road slope')
vel_axes = plt.subplot(2, 1, 1)
inp_axes = plt.subplot(2, 1, 2)
theta_hill = np.array([
    0 if t <= 5 else
    4./180. * pi * (t-5) if t <= 6 else
    4./180. * pi for t in T])

for m in (1200, 1600, 2000):
    # Compute the equilibrium state for the system
    X0, U0 = ct.find_eqpt(
        cruise_tf, [0, vref[0]], [vref[0], gear[0], theta0[0]],
        iu=[1, 2], y0=[vref[0], 0], iy=[0], params={'m':m})

    t, y = ct.input_output_response(
        cruise_tf, T, [vref, gear, theta_hill], X0, params={'m':m})

    # Plot the velocity
    plt.sca(vel_axes)
    plt.plot(t, y[0])

    # Plot the input
    plt.sca(inp_axes)
    plt.plot(t, y[1])

# Add labels to the plots
plt.sca(vel_axes)
plt.ylabel('Speed [m/s]')
plt.legend(['m = 1000 kg', 'm = 2000 kg', 'm = 3000 kg'], frameon=False)

plt.sca(inp_axes)
```

(continues on next page)

(continued from previous page)

```
plt.ylabel('Throttle')
plt.xlabel('Time [s]')

# Figure 4.2: Torque curves for a typical car engine. The graph on the
# left shows the torque generated by the engine as a function of the
# angular velocity of the engine, while the curve on the right shows
# torque as a function of car speed for different gears.

plt.figure()
plt.suptitle('Torque curves for typical car engine')

# Figure 4.2a - single torque curve as function of omega
omega_range = np.linspace(0, 700, 701)
plt.subplot(2, 2, 1)
plt.plot(omega_range, [motor_torque(w) for w in omega_range])
plt.xlabel('Angular velocity $\omega$ [rad/s]')
plt.ylabel('Torque $T$ [Nm]')
plt.grid(True, linestyle='dotted')

# Figure 4.2b - torque curves in different gears, as function of velocity
plt.subplot(2, 2, 2)
v_range = np.linspace(0, 70, 71)
alpha = [40, 25, 16, 12, 10]
for gear in range(5):
    omega_range = alpha[gear] * v_range
    plt.plot(v_range, [motor_torque(w) for w in omega_range],
             color='blue', linestyle='solid')

# Set up the axes and style
plt.axis([0, 70, 100, 200])
plt.grid(True, linestyle='dotted')

# Add labels
plt.text(11.5, 120, '$n$=1')
plt.text(24, 120, '$n$=2')
plt.text(42.5, 120, '$n$=3')
plt.text(58.5, 120, '$n$=4')
plt.text(58.5, 185, '$n$=5')
plt.xlabel('Velocity $v$ [m/s]')
plt.ylabel('Torque $T$ [Nm]')

plt.show(block=False)

# Figure 4.3: Car with cruise control encountering a sloping road

# PI controller model: control_pi()
#
# We add to this model a feedback controller that attempts to regulate the
# speed of the car in the presence of disturbances. We shall use a
# proportional-integral controller

def pi_update(t, x, u, params={}):
    # Get the controller parameters that we need
    ki = params.get('ki', 0.1)
    kaw = params.get('kaw', 2)  # anti-windup gain

    # Assign variables for inputs and states (for readability)
```

(continues on next page)

(continued from previous page)

```
    v = u[0]                     # current velocity
    vref = u[1]                  # reference velocity
    z = x[0]                     # integrated error

    # Compute the nominal controller output (needed for anti-windup)
    u_a = pi_output(t, x, u, params)

    # Compute anti-windup compensation (scale by ki to account for structure)
    u_aw = kaw/ki * (np.clip(u_a, 0, 1) - u_a) if ki != 0 else 0

    # State is the integrated error, minus anti-windup compensation
    return (vref - v) + u_aw

def pi_output(t, x, u, params={}):
    # Get the controller parameters that we need
    kp = params.get('kp', 0.5)
    ki = params.get('ki', 0.1)

    # Assign variables for inputs and states (for readability)
    v = u[0]                     # current velocity
    vref = u[1]                  # reference velocity
    z = x[0]                     # integrated error

    # PI controller
    return kp * (vref - v) + ki * z

control_pi = ct.NonlinearIOSystem(
    pi_update, pi_output, name='control',
    inputs = ['v', 'vref'], outputs = ['u'], states = ['z'],
    params = {'kp':0.5, 'ki':0.1})

# Create the closed loop system
cruise_pi = ct.InterconnectedSystem(
    (vehicle, control_pi), name='cruise',
    connections=(
        ('vehicle.u', 'control.u'),
        ('control.v', 'vehicle.v')),
    inplist=('control.vref', 'vehicle.gear', 'vehicle.theta'),
    outlist=('control.u', 'vehicle.v'), outputs=['u', 'v'])

# Figure 4.3b shows the response of the closed loop system.  The figure shows
# that even if the hill is so steep that the throttle changes from 0.17 to
# almost full throttle, the largest speed error is less than 1 m/s, and the
# desired velocity is recovered after 20 s.

# Define a function for creating a "standard" cruise control plot
def cruise_plot(sys, t, y, t_hill=5, vref=20, antiwindup=False,
                linetype='b-', subplots=[None, None]):
    # Figure out the plot bounds and indices
    v_min = vref-1.2; v_max = vref+0.5; v_ind = sys.find_output('v')
    u_min = 0; u_max = 2 if antiwindup else 1; u_ind = sys.find_output('u')

    # Make sure the upper and lower bounds on v are OK
    while max(y[v_ind]) > v_max: v_max += 1
    while min(y[v_ind]) < v_min: v_min -= 1

    # Create arrays for return values
```

(continues on next page)

(continued from previous page)

```
    subplot_axes = list(subplots)

    # Velocity profile
    if subplot_axes[0] is None:
        subplot_axes[0] = plt.subplot(2, 1, 1)
    else:
        plt.sca(subplots[0])
    plt.plot(t, y[v_ind], linetype)
    plt.plot(t, vref*np.ones(t.shape), 'k-')
    plt.plot([t_hill, t_hill], [v_min, v_max], 'k--')
    plt.axis([0, t[-1], v_min, v_max])
    plt.xlabel('Time $t$ [s]')
    plt.ylabel('Velocity $v$ [m/s]')

    # Commanded input profile
    if subplot_axes[1] is None:
        subplot_axes[1] = plt.subplot(2, 1, 2)
    else:
        plt.sca(subplots[1])
    plt.plot(t, y[u_ind], 'r--' if antiwindup else linetype)
    plt.plot([t_hill, t_hill], [u_min, u_max], 'k--')
    plt.axis([0, t[-1], u_min, u_max])
    plt.xlabel('Time $t$ [s]')
    plt.ylabel('Throttle $u$')

    # Applied input profile
    if antiwindup:
        # TODO: plot the actual signal from the process?
        plt.plot(t, np.clip(y[u_ind], 0, 1), linetype)
        plt.legend(['Commanded', 'Applied'], frameon=False)

    return subplot_axes

# Define the time and input vectors
T = np.linspace(0, 30, 101)
vref = 20 * np.ones(T.shape)
gear = 4 * np.ones(T.shape)
theta0 = np.zeros(T.shape)

# Compute the equilibrium throttle setting for the desired speed (solve for x
# and u given the gear, slope, and desired output velocity)
X0, U0, Y0 = ct.find_eqpt(
    cruise_pi, [vref[0], 0], [vref[0], gear[0], theta0[0]],
    y0=[0, vref[0]], iu=[1, 2], iy=[1], return_y=True)

# Now simulate the effect of a hill at t = 5 seconds
plt.figure()
plt.suptitle('Car with cruise control encountering sloping road')
theta_hill = [
    0 if t <= 5 else
    4./180. * pi * (t-5) if t <= 6 else
    4./180. * pi for t in T]
t, y = ct.input_output_response(cruise_pi, T, [vref, gear, theta_hill], X0)
cruise_plot(cruise_pi, t, y)

#
# Example 7.8: State space feedback with integral action
```

(continues on next page)

(continued from previous page)

```
#

# State space controller model: control_sf_ia()
#
# Construct a state space controller with integral action, linearized around
# an equilibrium point.  The controller is constructed around the equilibrium
# point (x_d, u_d) and includes both feedforward and feedback compensation.
#
# Controller inputs: (x, y, r)    system states, system output, reference
# Controller state:  z            integrated error (y - r)
# Controller output: u            state feedback control
#
# Note: to make the structure of the controller more clear, we implement this
# as a "nonlinear" input/output module, even though the actual input/output
# system is linear.  This also allows the use of parameters to set the
# operating point and gains for the controller.

def sf_update(t, z, u, params={}):
    y, r = u[1], u[2]
    return y - r

def sf_output(t, z, u, params={}):
    # Get the controller parameters that we need
    K = params.get('K', 0)
    ki = params.get('ki', 0)
    kf = params.get('kf', 0)
    xd = params.get('xd', 0)
    yd = params.get('yd', 0)
    ud = params.get('ud', 0)

    # Get the system state and reference input
    x, y, r = u[0], u[1], u[2]

    return ud - K * (x - xd) - ki * z + kf * (r - yd)

# Create the input/output system for the controller
control_sf = ct.NonlinearIOSystem(
    sf_update, sf_output, name='control',
    inputs=('x', 'y', 'r'),
    outputs=('u'),
    states=('z'))

# Create the closed loop system for the state space controller
cruise_sf = ct.InterconnectedSystem(
    (vehicle, control_sf), name='cruise',
    connections=(
        ('vehicle.u', 'control.u'),
        ('control.x', 'vehicle.v'),
        ('control.y', 'vehicle.v')),
    inplist=('control.r', 'vehicle.gear', 'vehicle.theta'),
    outlist=('control.u', 'vehicle.v'), outputs=['u', 'v'])

# Compute the linearization of the dynamics around the equilibrium point

# Y0 represents the steady state with PI control => we can use it to
# identify the steady state velocity and required throttle setting.
xd = Y0[1]
```

(continues on next page)

(continued from previous page)

```
ud = Y0[0]
yd = Y0[1]

# Compute the linearized system at the eq pt
cruise_linearized = ct.linearize(vehicle, xd, [ud, gear[0], 0])

# Construct the gain matrices for the system
A, B, C = cruise_linearized.A, cruise_linearized.B[0, 0], cruise_linearized.C
K = 0.5
kf = -1 / (C * np.linalg.inv(A - B * K) * B)

# Response of the system with no integral feedback term
plt.figure()
plt.suptitle('Cruise control with proportional and PI control')
theta_hill = [
    0 if t <= 8 else
    4./180. * pi * (t-8) if t <= 9 else
    4./180. * pi for t in T]
t, y = ct.input_output_response(
    cruise_sf, T, [vref, gear, theta_hill], [X0[0], 0],
    params={'K':K, 'kf':kf, 'ki':0.0, 'kf':kf, 'xd':xd, 'ud':ud, 'yd':yd})
subplots = cruise_plot(cruise_sf, t, y, t_hill=8, linetype='b--')

# Response of the system with state feedback + integral action
t, y = ct.input_output_response(
    cruise_sf, T, [vref, gear, theta_hill], [X0[0], 0],
    params={'K':K, 'kf':kf, 'ki':0.1, 'kf':kf, 'xd':xd, 'ud':ud, 'yd':yd})
cruise_plot(cruise_sf, t, y, t_hill=8, linetype='b-', subplots=subplots)

# Add a legend
plt.legend(['Proportional', 'PI control'], frameon=False)

# Example 11.5: simulate the effect of a (steeper) hill at t = 5 seconds
#
# The windup effect occurs when a car encounters a hill that is so steep (6
# deg) that the throttle saturates when the cruise controller attempts to
# maintain speed.

plt.figure()
plt.suptitle('Cruise control with integrator windup')
T = np.linspace(0, 70, 101)
vref = 20 * np.ones(T.shape)
theta_hill = [
    0 if t <= 5 else
    6./180. * pi * (t-5) if t <= 6 else
    6./180. * pi for t in T]
t, y = ct.input_output_response(
    cruise_pi, T, [vref, gear, theta_hill], X0,
    params={'kaw':0})
cruise_plot(cruise_pi, t, y, antiwindup=True)

# Example 11.6: add anti-windup compensation
#
# Anti-windup can be applied to the system to improve the response. Because of
# the feedback from the actuator model, the output of the integrator is
# quickly reset to a value such that the controller output is at the
# saturation limit.
```

(continues on next page)

(continued from previous page)

```python

plt.figure()
plt.suptitle('Cruise control with integrator anti-windup protection')
t, y = ct.input_output_response(
    cruise_pi, T, [vref, gear, theta_hill], X0,
    params={'kaw':2.})
cruise_plot(cruise_pi, t, y, antiwindup=True)

# If running as a standalone program, show plots and wait before closing
import os
if __name__ == '__main__' and 'PYCONTROL_TEST_EXAMPLES' not in os.environ:
    plt.show()
else:
    plt.show(block=False)
```

Notes

1. The environment variable *PYCONTROL_TEST_EXAMPLES* is used for testing to turn off plotting of the outputs.

8.1.9 Gain scheduled control for vehicle steeering (I/O system)

Code

```python
# steering-gainsched.py - gain scheduled control for vehicle steering
# RMM, 8 May 2019
#
# This file works through Example 1.1 in the "Optimization-Based Control"
# course notes by Richard Murray (avaliable at http://fbsbook.org, in the
# optimization-based control supplement).  It is intended to demonstrate the
# functionality for nonlinear input/output systems in the python-control
# package.

import numpy as np
import control as ct
from cmath import sqrt
import matplotlib.pyplot as mpl

#
# Vehicle steering dynamics
#
# The vehicle dynamics are given by a simple bicycle model.  We take the state
# of the system as (x, y, theta) where (x, y) is the position of the vehicle
# in the plane and theta is the angle of the vehicle with respect to
# horizontal.  The vehicle input is given by (v, phi) where v is the forward
# velocity of the vehicle and phi is the angle of the steering wheel.  The
# model includes saturation of the vehicle steering angle.
#
# System state: x, y, theta
# System input: v, phi
# System output: x, y
# System parameters: wheelbase, maxsteer
#
def vehicle_update(t, x, u, params):
```

(continues on next page)

(continued from previous page)

```
    # Get the parameters for the model
    l = params.get('wheelbase', 3.)         # vehicle wheelbase
    phimax = params.get('maxsteer', 0.5)    # max steering angle (rad)

    # Saturate the steering input
    phi = np.clip(u[1], -phimax, phimax)

    # Return the derivative of the state
    return np.array([
        np.cos(x[2]) * u[0],            # xdot = cos(theta) v
        np.sin(x[2]) * u[0],            # ydot = sin(theta) v
        (u[0] / l) * np.tan(phi)        # thdot = v/l tan(phi)
    ])

def vehicle_output(t, x, u, params):
    return x                            # return x, y, theta (full state)

# Define the vehicle steering dynamics as an input/output system
vehicle = ct.NonlinearIOSystem(
    vehicle_update, vehicle_output, states=3, name='vehicle',
    inputs=('v', 'phi'),
    outputs=('x', 'y', 'theta'))

#
# Gain scheduled controller
#
# For this system we use a simple schedule on the forward vehicle velocity and
# place the poles of the system at fixed values.  The controller takes the
# current vehicle position and orientation plus the velocity velocity as
# inputs, and returns the velocity and steering commands.
#
# System state: none
# System input: ex, ey, etheta, vd, phid
# System output: v, phi
# System parameters: longpole, latpole1, latpole2
#
def control_output(t, x, u, params):
    # Get the controller parameters
    longpole = params.get('longpole', -2.)
    latpole1 = params.get('latpole1', -1/2 + sqrt(-7)/2)
    latpole2 = params.get('latpole2', -1/2 - sqrt(-7)/2)
    l = params.get('wheelbase', 3)

    # Extract the system inputs
    ex, ey, etheta, vd, phid = u

    # Determine the controller gains
    alpha1 = -np.real(latpole1 + latpole2)
    alpha2 = np.real(latpole1 * latpole2)

    # Compute and return the control law
    v = -longpole * ex          # Note: no feedfwd (to make plot interesting)
    if vd != 0:
        phi = phid + (alpha1 * l) / vd * ey + (alpha2 * l) / vd * etheta
    else:
        # We aren't moving, so don't turn the steering wheel
        phi = phid
```

(continues on next page)

(continued from previous page)

```
    return  np.array([v, phi])

# Define the controller as an input/output system
controller = ct.NonlinearIOSystem(
    None, control_output, name='controller',        # static system
    inputs=('ex', 'ey', 'etheta', 'vd', 'phid'),    # system inputs
    outputs=('v', 'phi')                            # system outputs
)

#
# Reference trajectory subsystem
#
# The reference trajectory block generates a simple trajectory for the system
# given the desired speed (vref) and lateral position (yref).  The trajectory
# consists of a straight line of the form (vref * t, yref, 0) with nominal
# input (vref, 0).
#
# System state: none
# System input: vref, yref
# System output: xd, yd, thetad, vd, phid
# System parameters: none
#
def trajgen_output(t, x, u, params):
    vref, yref = u
    return np.array([vref * t, yref, 0, vref, 0])

# Define the trajectory generator as an input/output system
trajgen = ct.NonlinearIOSystem(
    None, trajgen_output, name='trajgen',
    inputs=('vref', 'yref'),
    outputs=('xd', 'yd', 'thetad', 'vd', 'phid'))

#
# System construction
#
# The input to the full closed loop system is the desired lateral position and
# the desired forward velocity.  The output for the system is taken as the
# full vehicle state plus the velocity of the vehicle.  The following diagram
# summarizes the interconnections:
#
#                        +---------+       +---------------> v
#                        |         |       |
# [ yref ]               |         v       |
# [      ] ---> trajgen -+-+-> controller -+-> vehicle -+-> [x, y, theta]
# [ vref ]                 ^                            |
#                          |                            |
#                          +----------- [-1] -----------+
#
# We construct the system using the InterconnectedSystem constructor and using
# signal labels to keep track of everything.

steering = ct.InterconnectedSystem(
    # List of subsystems
    (trajgen, controller, vehicle), name='steering',

    # Interconnections between  subsystems
```

(continues on next page)

(continued from previous page)

```
    connections=(
        ('controller.ex', 'trajgen.xd', '-vehicle.x'),
        ('controller.ey', 'trajgen.yd', '-vehicle.y'),
        ('controller.etheta', 'trajgen.thetad', '-vehicle.theta'),
        ('controller.vd', 'trajgen.vd'),
        ('controller.phid', 'trajgen.phid'),
        ('vehicle.v', 'controller.v'),
        ('vehicle.phi', 'controller.phi')
    ),

    # System inputs
    inplist=['trajgen.vref', 'trajgen.yref'],
    inputs=['yref', 'vref'],

    #  System outputs
    outlist=['vehicle.x', 'vehicle.y', 'vehicle.theta', 'controller.v',
             'controller.phi'],
    outputs=['x', 'y', 'theta', 'v', 'phi']
)

# Set up the simulation conditions
yref = 1
T = np.linspace(0, 5, 100)

# Set up a figure for plotting the results
mpl.figure();

# Plot the reference trajectory for the y position
mpl.plot([0, 5], [yref, yref], 'k--')

# Find the signals we want to plot
y_index = steering.find_output('y')
v_index = steering.find_output('v')

# Do an iteration through different speeds
for vref in [8, 10, 12]:
    # Simulate the closed loop controller response
    tout, yout = ct.input_output_response(
        steering, T, [vref * np.ones(len(T)), yref * np.ones(len(T))])

    # Plot the reference speed
    mpl.plot([0, 5], [vref, vref], 'k--')

    # Plot the system output
    y_line, = mpl.plot(tout, yout[y_index, :], 'r')  # lateral position
    v_line, = mpl.plot(tout, yout[v_index, :], 'b')  # vehicle velocity

# Add axis labels
mpl.xlabel('Time (s)')
mpl.ylabel('x vel (m/s), y pos (m)')
mpl.legend((v_line, y_line), ('v', 'y'), loc='center right', frameon=False)
```

Notes

8.1.10 Differentially flat system - kinematic car

This example demonstrates the use of the *flatsys* module for generating trajectories for differentially flat systems. The example is drawn from Chapter 8 of FBS2e.

Code

```
# kincar-flatsys.py - differentially flat systems example
# RMM, 3 Jul 2019
#
# This example demonstrates the use of the `flatsys` module for generating
# trajectories for differnetially flat systems by computing a trajectory for a
# kinematic (bicycle) model of a car changing lanes.

import os
import numpy as np
import matplotlib.pyplot as plt
import control as ct
import control.flatsys as fs


# Function to take states, inputs and return the flat flag
def vehicle_flat_forward(x, u, params={}):
    # Get the parameter values
    b = params.get('wheelbase', 3.)

    # Create a list of arrays to store the flat output and its derivatives
    zflag = [np.zeros(3), np.zeros(3)]

    # Flat output is the x, y position of the rear wheels
    zflag[0][0] = x[0]
    zflag[1][0] = x[1]

    # First derivatives of the flat output
    zflag[0][1] = u[0] * np.cos(x[2])  # dx/dt
    zflag[1][1] = u[0] * np.sin(x[2])  # dy/dt

    # First derivative of the angle
    thdot = (u[0]/b) * np.tan(u[1])

    # Second derivatives of the flat output (setting vdot = 0)
    zflag[0][2] = -u[0] * thdot * np.sin(x[2])
    zflag[1][2] =  u[0] * thdot * np.cos(x[2])

    return zflag


# Function to take the flat flag and return states, inputs
def vehicle_flat_reverse(zflag, params={}):
    # Get the parameter values
    b = params.get('wheelbase', 3.)

    # Create a vector to store the state and inputs
    x = np.zeros(3)
```

(continues on next page)

(continued from previous page)

```
    u = np.zeros(2)

    # Given the flat variables, solve for the state
    x[0] = zflag[0][0]  # x position
    x[1] = zflag[1][0]  # y position
    x[2] = np.arctan2(zflag[1][1], zflag[0][1])  # tan(theta) = ydot/xdot

    # And next solve for the inputs
    u[0] = zflag[0][1] * np.cos(x[2]) + zflag[1][1] * np.sin(x[2])
    thdot_v = zflag[1][2] * np.cos(x[2]) - zflag[0][2] * np.sin(x[2])
    u[1] = np.arctan2(thdot_v, u[0]**2 / b)

    return x, u


# Function to compute the RHS of the system dynamics
def vehicle_update(t, x, u, params):
    b = params.get('wheelbase', 3.)             # get parameter values
    dx = np.array([
        np.cos(x[2]) * u[0],
        np.sin(x[2]) * u[0],
        (u[0]/b) * np.tan(u[1])
    ])
    return dx


# Create differentially flat input/output system
vehicle_flat = fs.FlatSystem(
    vehicle_flat_forward, vehicle_flat_reverse, vehicle_update,
    inputs=('v', 'delta'), outputs=('x', 'y', 'theta'),
    states=('x', 'y', 'theta'))

# Define the endpoints of the trajectory
x0 = [0., -2., 0.]; u0 = [10., 0.]
xf = [40., 2., 0.]; uf = [10., 0.]
Tf = 4

# Define a set of basis functions to use for the trajectories
poly = fs.PolyFamily(6)

# Find a trajectory between the initial condition and the final condition
traj = fs.point_to_point(vehicle_flat, x0, u0, xf, uf, Tf, basis=poly)

# Create the desired trajectory between the initial and final condition
T = np.linspace(0, Tf, 500)
xd, ud = traj.eval(T)

# Simulation the open system dynamics with the full input
t, y, x = ct.input_output_response(
    vehicle_flat, T, ud, x0, return_x=True)

# Plot the open loop system dynamics
plt.figure()
plt.suptitle("Open loop trajectory for kinematic car lane change")

# Plot the trajectory in xy coordinates
plt.subplot(4, 1, 2)
```

(continues on next page)

(continued from previous page)

```
plt.plot(x[0], x[1])
plt.xlabel('x [m]')
plt.ylabel('y [m]')
plt.axis([x0[0], xf[0], x0[1]-1, xf[1]+1])

# Time traces of the state and input
plt.subplot(2, 4, 5)
plt.plot(t, x[1])
plt.ylabel('y [m]')

plt.subplot(2, 4, 6)
plt.plot(t, x[2])
plt.ylabel('theta [rad]')

plt.subplot(2, 4, 7)
plt.plot(t, ud[0])
plt.xlabel('Time t [sec]')
plt.ylabel('v [m/s]')
plt.axis([0, Tf, u0[0] - 1, uf[0] + 1])

plt.subplot(2, 4, 8)
plt.plot(t, ud[1])
plt.xlabel('Ttime t [sec]')
plt.ylabel('$\delta$ [rad]')
plt.tight_layout()

# Show the results unless we are running in batch mode
if 'PYCONTROL_TEST_EXAMPLES' not in os.environ:
    plt.show()
```

Notes

1. The environment variable *PYCONTROL_TEST_EXAMPLES* is used for testing to turn off plotting of the outputs.

8.2 Jupyter notebooks

The examples below use *python-control* in a Jupyter notebook environment. These notebooks demonstrate the use of modeling, anaylsis, and design tools using running examples in FBS2e.

8.2.1 Cruise control

Richard M. Murray and Karl J. Åström
17 Jun 2019

The cruise control system of a car is a common feedback system encountered in everyday life. The system attempts to maintain a constant velocity in the presence of disturbances primarily caused by changes in the slope of a road. The controller compensates for these unknowns by measuring the speed of the car and adjusting the throttle appropriately.

This notebook explores the dynamics and control of the cruise control system, following the material presenting in Feedback Systems by Astrom and Murray. A nonlinear model of the vehicle dynamics is used, with both state space and frequency domain control laws. The process model is presented in Section 1, and a controller based on state feedback is discussed in Section 2, where we also add integral action to the controller. In Section 3 we explore the behavior with PI control including the effect of actuator saturation and how it is avoided by windup protection. Different methods of constructing control systems are shown, all using the InputOutputSystem class (and subclasses).

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
from math import pi
import control as ct
```

Process Model

Vehicle Dynamics

To develop a mathematical model we start with a force balance for the car body. Let v be the speed of the car, m the total mass (including passengers), F the force generated by the contact of the wheels with the road, and F_d the disturbance force due to gravity, friction, and aerodynamic drag.

[2]:
```
def vehicle_update(t, x, u, params={}):
    """Vehicle dynamics for cruise control system.

    Parameters
    ----------
    x : array
         System state: car velocity in m/s
    u : array
         System input: [throttle, gear, road_slope], where throttle is
         a float between 0 and 1, gear is an integer between 1 and 5,
         and road_slope is in rad.

    Returns
    -------
    float
        Vehicle acceleration

    """
    from math import copysign, sin
    sign = lambda x: copysign(1, x)         # define the sign() function

    # Set up the system parameters
    m = params.get('m', 1600.)              # vehicle mass, kg
    g = params.get('g', 9.8)                # gravitational constant, m/s^2
    Cr = params.get('Cr', 0.01)             # coefficient of rolling friction
    Cd = params.get('Cd', 0.32)             # drag coefficient
    rho = params.get('rho', 1.3)            # density of air, kg/m^3
    A = params.get('A', 2.4)                # car area, m^2
    alpha = params.get(
        'alpha', [40, 25, 16, 12, 10])      # gear ratio / wheel radius

    # Define variables for vehicle state and inputs
    v = x[0]                           # vehicle velocity
    throttle = np.clip(u[0], 0, 1)     # vehicle throttle
    gear = u[1]                        # vehicle gear
    theta = u[2]                       # road slope

    # Force generated by the engine
```

(continues on next page)

(continued from previous page)

```
    omega = alpha[int(gear)-1] * v      # engine angular speed
    F = alpha[int(gear)-1] * motor_torque(omega, params) * throttle

    # Disturbance forces
    #
    # The disturbance force Fd has three major components: Fg, the forces due
    # to gravity; Fr, the forces due to rolling friction; and Fa, the
    # aerodynamic drag.

    # Letting the slope of the road be \theta (theta), gravity gives the
    # force Fg = m g sin \theta.

    Fg = m * g * sin(theta)

    # A simple model of rolling friction is Fr = m g Cr sgn(v), where Cr is
    # the coefficient of rolling friction and sgn(v) is the sign of v (±1) or
    # zero if v = 0.

    Fr  = m * g * Cr * sign(v)

    # The aerodynamic drag is proportional to the square of the speed: Fa =
    # 1/2 \rho Cd A |v| v, where \rho is the density of air, Cd is the
    # shape-dependent aerodynamic drag coefficient, and A is the frontal area
    # of the car.

    Fa = 1/2 * rho * Cd * A * abs(v) * v

    # Final acceleration on the car
    Fd = Fg + Fr + Fa
    dv = (F - Fd) / m

    return dv
```

Engine model

The force F is generated by the engine, whose torque is proportional to the rate of fuel injection, which is itself proportional to a control signal 0 <= u <= 1 that controls the throttle position. The torque also depends on engine speed omega.

[3]:
```
def motor_torque(omega, params={}):
    # Set up the system parameters
    Tm = params.get('Tm', 190.)             # engine torque constant
    omega_m = params.get('omega_m', 420.)   # peak engine angular speed
    beta = params.get('beta', 0.4)          # peak engine rolloff

    return np.clip(Tm * (1 - beta * (omega/omega_m - 1)**2), 0, None)
```

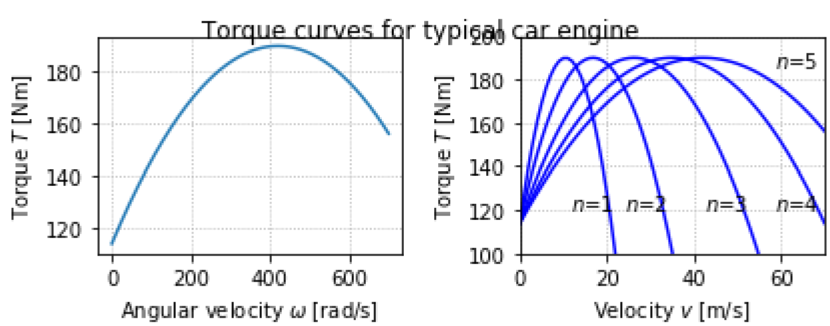
Torque curves for a typical car engine. The graph on the left shows the torque generated by the engine as a function of the angular velocity of the engine, while the curve on the right shows torque as a function of car speed for different gears.

[4]:
```
# Figure 4.2a - single torque curve as function of omega
omega_range = np.linspace(0, 700, 701)
plt.subplot(2, 2, 1)
```

(continues on next page)

(continued from previous page)

```
plt.plot(omega_range, [motor_torque(w) for w in omega_range])
plt.xlabel('Angular velocity $\omega$ [rad/s]')
plt.ylabel('Torque $T$ [Nm]')
plt.grid(True, linestyle='dotted')

# Figure 4.2b - torque curves in different gears, as function of velocity
plt.subplot(2, 2, 2)
v_range = np.linspace(0, 70, 71)
alpha = [40, 25, 16, 12, 10]
for gear in range(5):
    omega_range = alpha[gear] * v_range
    plt.plot(v_range, [motor_torque(w) for w in omega_range],
             color='blue', linestyle='solid')

# Set up the axes and style
plt.axis([0, 70, 100, 200])
plt.grid(True, linestyle='dotted')

# Add labels
plt.text(11.5, 120, '$n$=1')
plt.text(24, 120, '$n$=2')
plt.text(42.5, 120, '$n$=3')
plt.text(58.5, 120, '$n$=4')
plt.text(58.5, 185, '$n$=5')
plt.xlabel('Velocity $v$ [m/s]')
plt.ylabel('Torque $T$ [Nm]')

plt.tight_layout()
plt.suptitle('Torque curves for typical car engine');
```



Input/ouput model for the vehicle system

We now create an input/output model for the vehicle system that takes the throttle input u, the gear and the angle of the road θ as input. The output of this model is the current vehicle velocity v.

[5]:
```
vehicle = ct.NonlinearIOSystem(
    vehicle_update, None, name='vehicle',
    inputs = ('u', 'gear', 'theta'), outputs = ('v'), states=('v'))

# Define a generator for creating a "standard" cruise control plot
def cruise_plot(sys, t, y, t_hill=5, vref=20, antiwindup=False, linetype='b-',
                subplots=[None, None]):
    # Figure out the plot bounds and indices
```

(continues on next page)

(continued from previous page)

```
    v_min = vref-1.2; v_max = vref+0.5; v_ind = sys.find_output('v')
    u_min = 0; u_max = 2 if antiwindup else 1; u_ind = sys.find_output('u')

    # Make sure the upper and lower bounds on v are OK
    while max(y[v_ind]) > v_max: v_max += 1
    while min(y[v_ind]) < v_min: v_min -= 1

    # Create arrays for return values
    subplot_axes = subplots.copy()

    # Velocity profile
    if subplot_axes[0] is None:
        subplot_axes[0] = plt.subplot(2, 1, 1)
    else:
        plt.sca(subplots[0])
    plt.plot(t, y[v_ind], linetype)
    plt.plot(t, vref*np.ones(t.shape), 'k-')
    plt.plot([t_hill, t_hill], [v_min, v_max], 'k--')
    plt.axis([0, t[-1], v_min, v_max])
    plt.xlabel('Time $t$ [s]')
    plt.ylabel('Velocity $v$ [m/s]')

    # Commanded input profile
    if subplot_axes[1] is None:
        subplot_axes[1] = plt.subplot(2, 1, 2)
    else:
        plt.sca(subplots[1])
    plt.plot(t, y[u_ind], 'r--' if antiwindup else linetype)
    plt.plot([t_hill, t_hill], [u_min, u_max], 'k--')
    plt.axis([0, t[-1], u_min, u_max])
    plt.xlabel('Time $t$ [s]')
    plt.ylabel('Throttle $u$')

    # Applied input profile
    if antiwindup:
        plt.plot(t, np.clip(y[u_ind], 0, 1), linetype)
        plt.legend(['Commanded', 'Applied'], frameon=False)

    return subplot_axes
```

State space controller

Construct a state space controller with integral action, linearized around an equilibrium point. The controller is constructed around the equilibrium point (x_d, u_d) and includes both feedforward and feedback compensation.

- Controller inputs - (x, y, r): system states, system output, reference
- Controller state - z: integrated error $(y - r)$
- Controller output - u: state feedback control

Note: to make the structure of the controller more clear, we implement this as a "nonlinear" input/output module, even though the actual input/output system is linear. This also allows the use of parameters to set the operating point and gains for the controller.

[6]:
```
def sf_update(t, z, u, params={}):
    y, r = u[1], u[2]
    return y - r

def sf_output(t, z, u, params={}):
    # Get the controller parameters that we need
    K = params.get('K', 0)
    ki = params.get('ki', 0)
    kf = params.get('kf', 0)
    xd = params.get('xd', 0)
    yd = params.get('yd', 0)
    ud = params.get('ud', 0)

    # Get the system state and reference input
    x, y, r = u[0], u[1], u[2]

    return ud - K * (x - xd) - ki * z + kf * (r - yd)

# Create the input/output system for the controller
control_sf = ct.NonlinearIOSystem(
    sf_update, sf_output, name='control',
    inputs=('x', 'y', 'r'),
    outputs=('u'),
    states=('z'))

# Create the closed loop system for the state space controller
cruise_sf = ct.InterconnectedSystem(
    (vehicle, control_sf), name='cruise',
    connections=(
        ('vehicle.u', 'control.u'),
        ('control.x', 'vehicle.v'),
        ('control.y', 'vehicle.v')),
    inplist=('control.r', 'vehicle.gear', 'vehicle.theta'),
    outlist=('control.u', 'vehicle.v'), outputs=['u', 'v'])

# Define the time and input vectors
T = np.linspace(0, 25, 501)
vref = 20 * np.ones(T.shape)
gear = 4 * np.ones(T.shape)
theta0 = np.zeros(T.shape)

# Find the equilibrium point for the system
Xeq, Ueq = ct.find_eqpt(
    vehicle, [vref[0]], [0, gear[0], theta0[0]], y0=[vref[0]], iu=[1, 2])
print("Xeq = ", Xeq)
print("Ueq = ", Ueq)

# Compute the linearized system at the eq pt
cruise_linearized = ct.linearize(vehicle, Xeq, [Ueq[0], gear[0], 0])
```

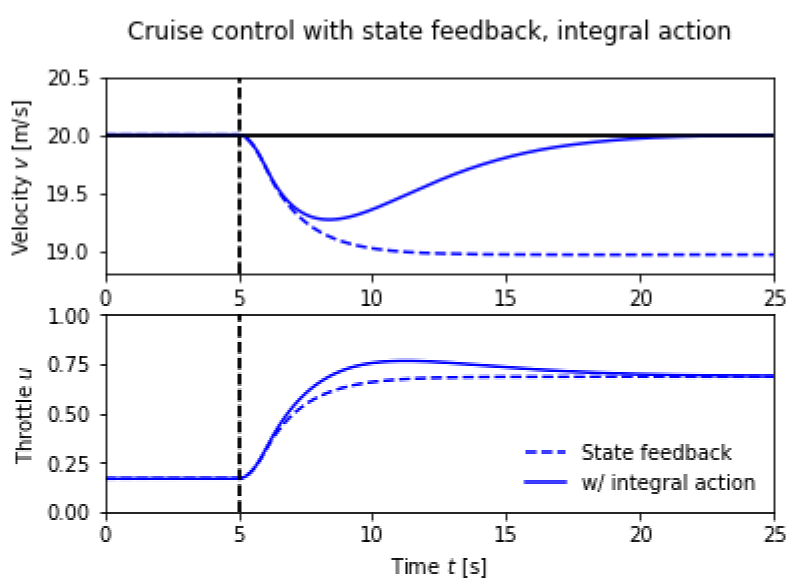
```
Xeq =  [20.]
Ueq =  [0.16874874 4.         0.        ]
```

[7]:
```
# Construct the gain matrices for the system
A, B, C = cruise_linearized.A, cruise_linearized.B[0, 0], cruise_linearized.C
K = 0.5
kf = -1 / (C * np.linalg.inv(A - B * K) * B)
```

(continues on next page)

(continued from previous page)

```
# Compute the steady state velocity and throttle setting
xd = Xeq[0]
ud = Ueq[0]
yd = vref[-1]

# Response of the system with no integral feedback term
plt.figure()
theta_hill = [
    0 if t <= 5 else
    4./180. * pi * (t-5) if t <= 6 else
    4./180. * pi for t in T]
t, y_sfb = ct.input_output_response(
    cruise_sf, T, [vref, gear, theta_hill], [Xeq[0], 0],
    params={'K':K, 'ki':0.0, 'kf':kf, 'xd':xd, 'ud':ud, 'yd':yd})
subplots = cruise_plot(cruise_sf, t, y_sfb, t_hill=5, linetype='b--')

# Response of the system with state feedback + integral action
t, y_sfb_int = ct.input_output_response(
    cruise_sf, T, [vref, gear, theta_hill], [Xeq[0], 0],
    params={'K':K, 'ki':0.1, 'kf':kf, 'xd':xd, 'ud':ud, 'yd':yd})
cruise_plot(cruise_sf, t, y_sfb_int, t_hill=5, linetype='b-', subplots=subplots)

# Add title and legend
plt.suptitle('Cruise control with state feedback, integral action')
import matplotlib.lines as mlines
p_line = mlines.Line2D([], [], color='blue', linestyle='--', label='State feedback')
pi_line = mlines.Line2D([], [], color='blue', linestyle='-', label='w/ integral action
↪')
plt.legend(handles=[p_line, pi_line], frameon=False, loc='lower right');
```

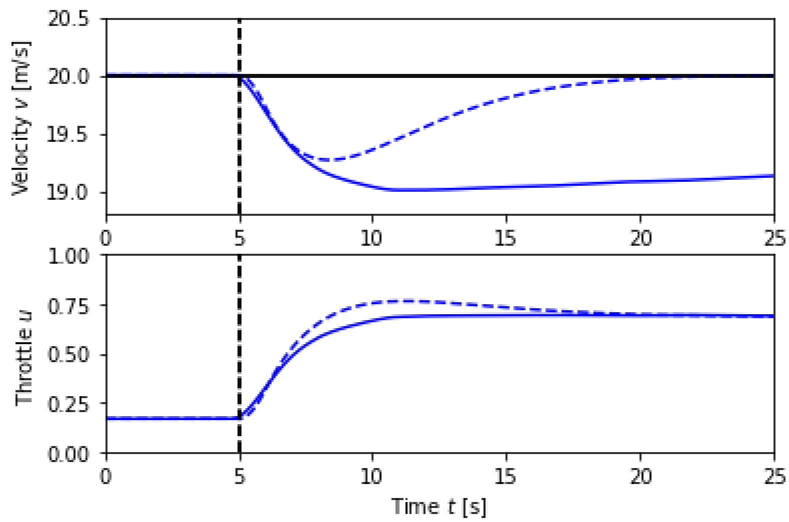


Pole/zero cancellation

The transfer function for the linearized dynamics of the cruise control system is given by $P(s) = b/(s+a)$. A simple (but not necessarily good) way to design a PI controller is to choose the parameters of the PI controller as $k_\mathrm{i} = ak_\mathrm{p}$. The controller transfer function is then $C(s) = k_\mathrm{p} + k_\mathrm{i}/s = k_\mathrm{i}(s+a)/s$. It has a zero at $s = -k_\mathrm{i}/k_\mathrm{p} = -a$ that cancels the process pole at $s = -a$. We have $P(s)C(s) = k_\mathrm{i}/s$ giving the transfer function from reference to vehicle velocity as $G_{yr}(s) = bk_\mathrm{p}/(s + bk_\mathrm{p})$, and control design is then simply a matter of choosing the gain k_p. The closed loop system dynamics are of first order with the time constant $1/(bk_\mathrm{p})$.

[8]:
```
# Get the transfer function from throttle input + hill to vehicle speed
P = ct.ss2tf(cruise_linearized[0, 0])

# Construction a controller that cancels the pole
kp = 0.5
a = -P.pole()[0]
b = np.real(P(0)) * a
ki = a * kp
C = ct.tf2ss(ct.TransferFunction([kp, ki], [1, 0]))
control_pz = ct.LinearIOSystem(C, name='control', inputs='u', outputs='y')
print("system: a = ", a, ", b = ", b)
print("pzcancel: kp =", kp, ", ki =", ki, ", 1/(kp b) = ", 1/(kp * b))
print("sfb_int: K = ", K, ", ki = 0.1")

# Construct the closed loop system and plot the response
# Create the closed loop system for the state space controller
cruise_pz = ct.InterconnectedSystem(
    (vehicle, control_pz), name='cruise_pz',
    connections = (
        ('control.u', '-vehicle.v'),
        ('vehicle.u', 'control.y')),
    inplist = ('control.u', 'vehicle.gear', 'vehicle.theta'),
    inputs = ('vref', 'gear', 'theta'),
    outlist = ('vehicle.v', 'vehicle.u'),
    outputs = ('v', 'u'))

# Find the equilibrium point
X0, U0 = ct.find_eqpt(
    cruise_pz, [vref[0], 0], [vref[0], gear[0], theta0[0]],
    iu=[1, 2], y0=[vref[0], 0], iy=[0])

# Response of the system with PI controller canceling process pole
t, y_pzcancel = ct.input_output_response(
    cruise_pz, T, [vref, gear, theta_hill], X0)
subplots = cruise_plot(cruise_pz, t, y_pzcancel, t_hill=5, linetype='b-')
cruise_plot(cruise_sf, t, y_sfb_int, t_hill=5, linetype='b--', subplots=subplots);
```

```
system: a =  0.010124405669387215 , b =  1.3203061238159202
pzcancel: kp = 0.5 , ki = 0.005062202834693608 , 1/(kp b) =  1.5148002148317266
sfb_int: K =  0.5 , ki = 0.1
```



PI Controller

In this example, the speed of the vehicle is measured and compared to the desired speed. The controller is a PI controller represented as a transfer function. In the textbook, the simulations are done for LTI systems, but here we simulate the full nonlinear system.

Parameter design through pole placement

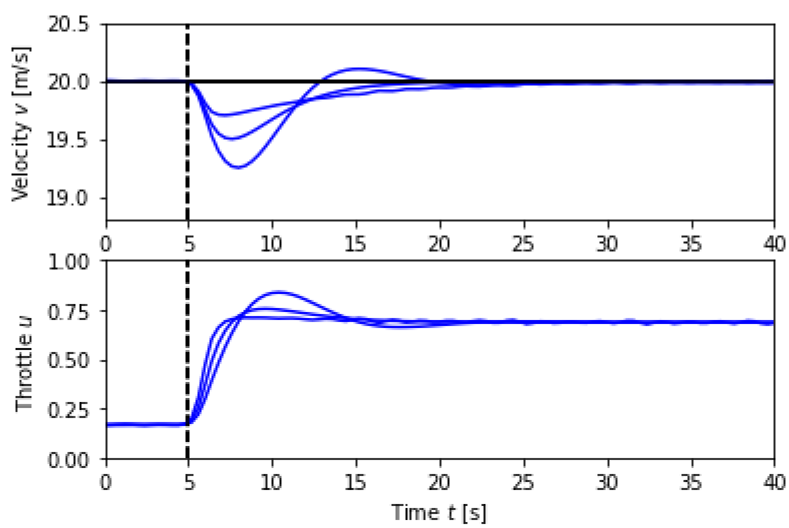
To illustrate the design of a PI controller, we choose the gains k_p and k_i so that the characteristic polynomial has the form

$$s^2 + 2\zeta\omega_0 s + \omega_0^2$$

[9]:
```
# Values of the first order transfer function P(s) = b/(s + a) are set above

# Define the input that we want to track
T = np.linspace(0, 40, 101)
vref = 20 * np.ones(T.shape)
gear = 4 * np.ones(T.shape)
theta_hill = np.array([
    0 if t <= 5 else
    4./180. * pi * (t-5) if t <= 6 else
    4./180. * pi for t in T])

# Fix \omega_0 and vary \zeta
w0 = 0.5
subplots = [None, None]
for zeta in [0.5, 1, 2]:
    # Create the controller transfer function (as an I/O system)
    kp = (2*zeta*w0 - a)/b
    ki = w0**2 / b
    control_tf = ct.tf2io(
        ct.TransferFunction([kp, ki], [1, 0.01*ki/kp]),
```
(continues on next page)

(continued from previous page)

```
        name='control', inputs='u', outputs='y')

    # Construct the closed loop system by interconnecting process and controller
    cruise_tf = ct.InterconnectedSystem(
    (vehicle, control_tf), name='cruise',
    connections = [('control.u', '-vehicle.v'), ('vehicle.u', 'control.y')],
    inplist = ('control.u', 'vehicle.gear', 'vehicle.theta'),
        inputs = ('vref', 'gear', 'theta'),
    outlist = ('vehicle.v', 'vehicle.u'), outputs = ('v', 'u'))

    # Plot the velocity response
    X0, U0 = ct.find_eqpt(
        cruise_tf, [vref[0], 0], [vref[0], gear[0], theta_hill[0]],
        iu=[1, 2], y0=[vref[0], 0], iy=[0])

    t, y = ct.input_output_response(cruise_tf, T, [vref, gear, theta_hill], X0)
    subplots = cruise_plot(cruise_tf, t, y, t_hill=5, subplots=subplots)
```

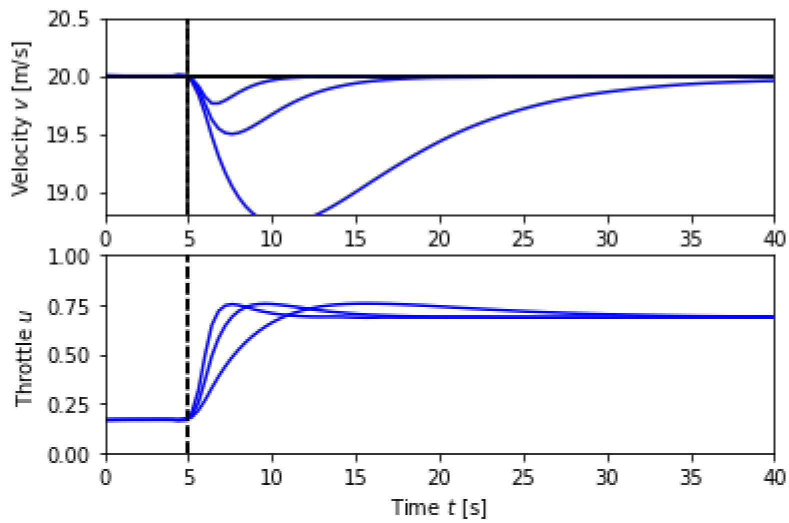


[10]:
```
# Fix \zeta and vary \omega_0
zeta = 1
subplots = [None, None]
for w0 in [0.2, 0.5, 1]:
    # Create the controller transfer function (as an I/O system)
    kp = (2*zeta*w0 - a)/b
    ki = w0**2 / b
    control_tf = ct.tf2io(
        ct.TransferFunction([kp, ki], [1, 0.01*ki/kp]),
        name='control', inputs='u', outputs='y')

    # Construct the closed loop system by interconnecting process and controller
    cruise_tf = ct.InterconnectedSystem(
    (vehicle, control_tf), name='cruise',
    connections = [('control.u', '-vehicle.v'), ('vehicle.u', 'control.y')],
    inplist = ('control.u', 'vehicle.gear', 'vehicle.theta'),
        inputs = ('vref', 'gear', 'theta'),
    outlist = ('vehicle.v', 'vehicle.u'), outputs = ('v', 'u'))
```

(continues on next page)

(continued from previous page)

```
    # Plot the velocity response
    X0, U0 = ct.find_eqpt(
        cruise_tf, [vref[0], 0], [vref[0], gear[0], theta_hill[0]],
        iu=[1, 2], y0=[vref[0], 0], iy=[0])

    t, y = ct.input_output_response(cruise_tf, T, [vref, gear, theta_hill], X0)
    subplots = cruise_plot(cruise_tf, t, y, t_hill=5, subplots=subplots)
```



Robustness to change in mass

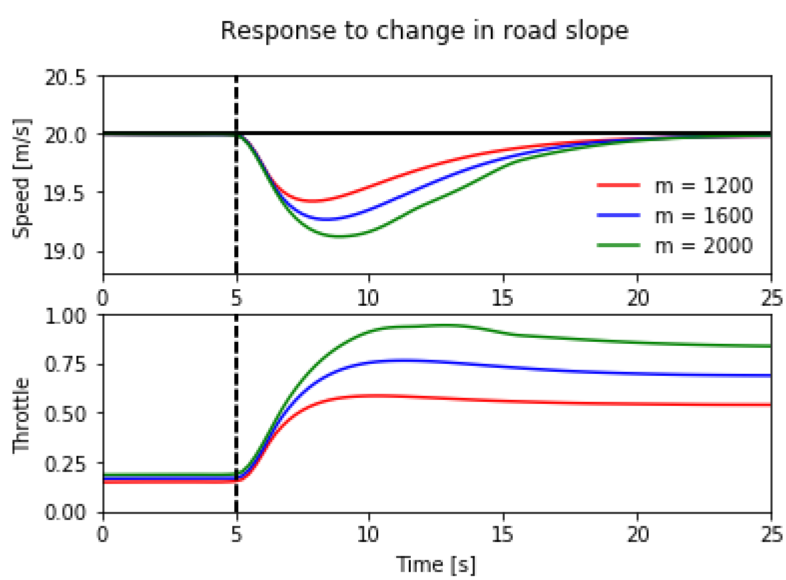
[11]:
```
# Nominal controller design for remaining analyses
# Construct a PI controller with rolloff, as a transfer function
Kp = 0.5                        # proportional gain
Ki = 0.1                        # integral gain
control_tf = ct.tf2io(
    ct.TransferFunction([Kp, Ki], [1, 0.01*Ki/Kp]),
    name='control', inputs='u', outputs='y')

cruise_tf = ct.InterconnectedSystem(
    (vehicle, control_tf), name='cruise',
    connections = [('control.u', '-vehicle.v'), ('vehicle.u', 'control.y')],
    inplist = ('control.u', 'vehicle.gear', 'vehicle.theta'), inputs = ('vref', 'gear
↪', 'theta'),
    outlist = ('vehicle.v', 'vehicle.u'), outputs = ('v', 'u'))
```

[12]:
```
# Define the time and input vectors
T = np.linspace(0, 25, 101)
vref = 20 * np.ones(T.shape)
gear = 4 * np.ones(T.shape)
theta0 = np.zeros(T.shape)

# Now simulate the effect of a hill at t = 5 seconds
plt.figure()
plt.suptitle('Response to change in road slope')
theta_hill = np.array([
```

(continues on next page)

(continued from previous page)

```
    0 if t <= 5 else
    4./180. * pi * (t-5) if t <= 6 else
    4./180. * pi for t in T])

subplots = [None, None]
linecolor = ['red', 'blue', 'green']
handles = []
for i, m in enumerate([1200, 1600, 2000]):
    # Compute the equilibrium state for the system
    X0, U0 = ct.find_eqpt(
        cruise_tf, [vref[0], 0], [vref[0], gear[0], theta0[0]],
        iu=[1, 2], y0=[vref[0], 0], iy=[0], params={'m':m})

    t, y = ct.input_output_response(
        cruise_tf, T, [vref, gear, theta_hill], X0, params={'m':m})

    subplots = cruise_plot(cruise_tf, t, y, t_hill=5, subplots=subplots,
                           linetype=linecolor[i][0] + '-')
    handles.append(mlines.Line2D([], [], color=linecolor[i], linestyle='-',
                                 label="m = %d" % m))

# Add labels to the plots
plt.sca(subplots[0])
plt.ylabel('Speed [m/s]')
plt.legend(handles=handles, frameon=False, loc='lower right');

plt.sca(subplots[1])
plt.ylabel('Throttle')
plt.xlabel('Time [s]');
```



PI controller with antiwindup protection

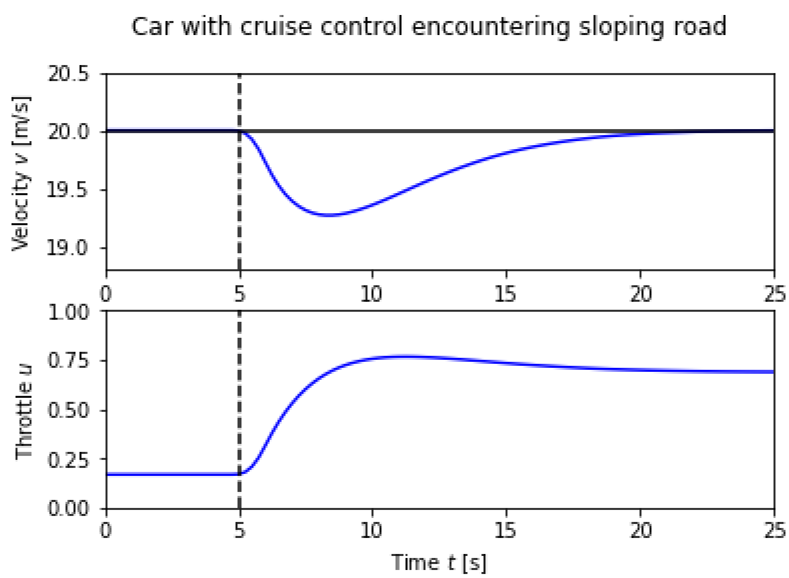
We now create a more complicated feedback controller that includes anti-windup protection.

[13]:
```
def pi_update(t, x, u, params={}):
    # Get the controller parameters that we need
    ki = params.get('ki', 0.1)
    kaw = params.get('kaw', 2)  # anti-windup gain

    # Assign variables for inputs and states (for readability)
    v = u[0]                    # current velocity
    vref = u[1]                 # reference velocity
    z = x[0]                    # integrated error

    # Compute the nominal controller output (needed for anti-windup)
    u_a = pi_output(t, x, u, params)

    # Compute anti-windup compensation (scale by ki to account for structure)
    u_aw = kaw/ki * (np.clip(u_a, 0, 1) - u_a) if ki != 0 else 0

    # State is the integrated error, minus anti-windup compensation
    return (vref - v) + u_aw

def pi_output(t, x, u, params={}):
    # Get the controller parameters that we need
    kp = params.get('kp', 0.5)
    ki = params.get('ki', 0.1)

    # Assign variables for inputs and states (for readability)
    v = u[0]                    # current velocity
    vref = u[1]                 # reference velocity
    z = x[0]                    # integrated error

    # PI controller
    return kp * (vref - v) + ki * z

control_pi = ct.NonlinearIOSystem(
    pi_update, pi_output, name='control',
    inputs = ['v', 'vref'], outputs = ['u'], states = ['z'],
    params = {'kp':0.5, 'ki':0.1})

# Create the closed loop system
cruise_pi = ct.InterconnectedSystem(
    (vehicle, control_pi), name='cruise',
    connections=(
        ('vehicle.u', 'control.u'),
        ('control.v', 'vehicle.v')),
    inplist=('control.vref', 'vehicle.gear', 'vehicle.theta'),
    outlist=('control.u', 'vehicle.v'), outputs=['u', 'v'])
```

Response to a small hill

Figure 4.3b shows the response of the closed loop system. The figure shows that even if the hill is so steep that the throttle changes from 0.17 to almost full throttle, the largest speed error is less than 1 m/s, and the desired velocity is recovered after 20 s.

[14]:
```
# Compute the equilibrium throttle setting for the desired speed
X0, U0, Y0 = ct.find_eqpt(
    cruise_pi, [vref[0], 0], [vref[0], gear[0], theta0[0]],
    y0=[0, vref[0]], iu=[1, 2], iy=[1], return_y=True)

# Now simulate the effect of a hill at t = 5 seconds
plt.figure()
plt.suptitle('Car with cruise control encountering sloping road')
theta_hill = [
    0 if t <= 5 else
    4./180. * pi * (t-5) if t <= 6 else
    4./180. * pi for t in T]
t, y = ct.input_output_response(
    cruise_pi, T, [vref, gear, theta_hill], X0)
cruise_plot(cruise_pi, t, y);
```



Effect of Windup

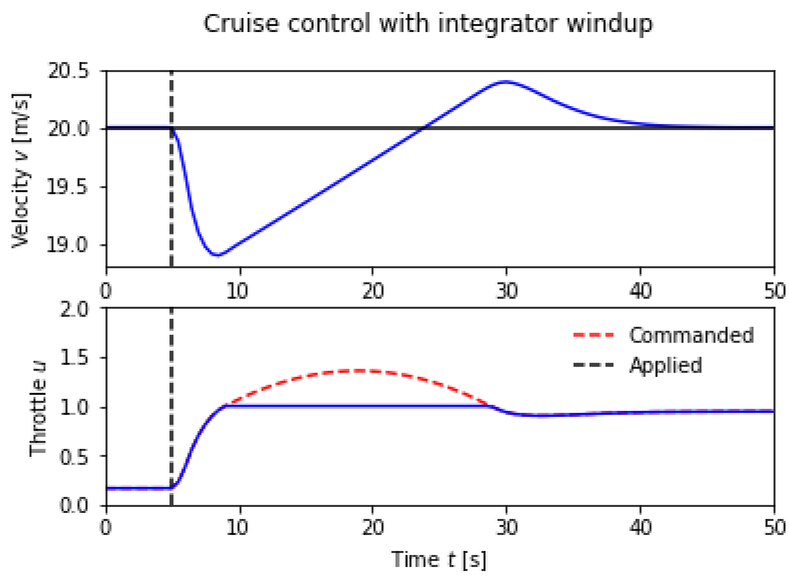
The windup effect occurs when a car encounters a hill that is so steep (6°) that the throttle saturates when the cruise controller attempts to maintain speed.

[15]:
```
plt.figure()
plt.suptitle('Cruise control with integrator windup')
T = np.linspace(0, 50, 101)
vref = 20 * np.ones(T.shape)
theta_hill = [
    0 if t <= 5 else
    6./180. * pi * (t-5) if t <= 6 else
    6./180. * pi for t in T]
```

(continues on next page)

(continued from previous page)

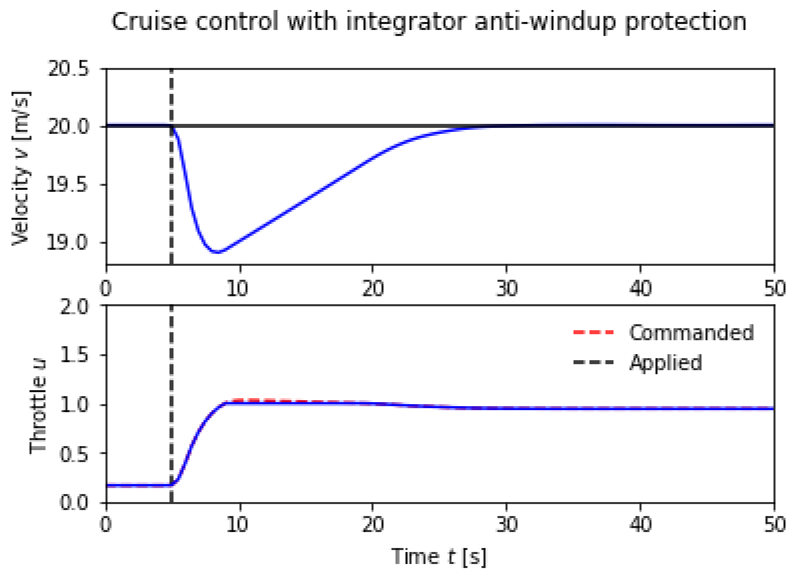
```
t, y = ct.input_output_response(
    cruise_pi, T, [vref, gear, theta_hill], X0,
    params={'kaw':0})
cruise_plot(cruise_pi, t, y, antiwindup=True);
```



PI controller with anti-windup compensation

Anti-windup can be applied to the system to improve the response. Because of the feedback from the actuator model, the output of the integrator is quickly reset to a value such that the controller output is at the saturation limit.

[16]:
```
plt.figure()
plt.suptitle('Cruise control with integrator anti-windup protection')
t, y = ct.input_output_response(
    cruise_pi, T, [vref, gear, theta_hill], X0,
    params={'kaw':2.})
cruise_plot(cruise_pi, t, y, antiwindup=True);
```




```
[ ]:
```

8.2.2 Vehicle steering

Karl J. Astrom and Richard M. Murray
23 Jul 2019

This notebook contains the computations for the vehicle steering running example in *Feedback Systems*.

RMM comments to Karl, 27 Jun 2019 * I'm using this notebook to walk through all of the vehicle steering examples and make sure that all of the parameters, conditions, and maximum steering angles are consitent and reasonable. * Please feel free to send me comments on the contents as well as the bulletted notes, in whatever form is most convenient. * Once we have sorted out all of the settings we want to use, I'll copy over the changes into the MATLAB files that we use for creating the figures in the book. * These notes will be removed from the notebook once we have finalized everything.

```
[1]: import numpy as np
import matplotlib.pyplot as plt
import control as ct
ct.use_fbs_defaults()
ct.use_numpy_matrix(False)
```

Vehicle steering dynamics (Example 3.11)

The vehicle dynamics are given by a simple bicycle model. We take the state of the system as (x, y, θ) where (x, y) is the position of the reference point of the vehicle in the plane and θ is the angle of the vehicle with respect to horizontal. The vehicle input is given by (v, δ) where v is the forward velocity of the vehicle and δ is the angle of the steering wheel. We take as parameters the wheelbase b and the offset a between the rear wheels and the reference point. The model includes saturation of the vehicle steering angle (`maxsteer`).

- System state: `x`, `y`, `theta`
- System input: `v`, `delta`
- System output: `x`, `y`
- System parameters: `wheelbase`, `refoffset`, `maxsteer`

Assuming no slipping of the wheels, the motion of the vehicle is given by a rotation around a point O that depends on the steering angle δ. To compute the angle α of the velocity of the reference point with respect to the axis of the vehicle, we let the distance from the center of rotation O to the contact point of the rear wheel be r_r and it the follows from Figure 3.17 in FBS that $b = r_\mathrm{r} \tan \delta$ and $a = r_\mathrm{r} \tan \alpha$, which implies that $\tan \alpha = (a/b) \tan \delta$.

Reasonable limits for the steering angle depend on the speed. The physical limit is given in our model as 0.5 radians (about 30 degrees). However, this limit is rarely possible when the car is driving since it would cause the tires to slide on the pavement. We us a limit of 0.1 radians (about 6 degrees) at 10 m/s ($\approx$ 35 kph) and 0.05 radians (about 3 degrees) at 30 m/s ($\approx$ 110 kph). Note that a steering angle of 0.05 rad gives a cross acceleration of $(v^2/b) \tan \delta \approx (100/3) 0.05 = 1.7$ m/s^2 at 10 m/s and 15 m/s^2 at 30 m/s ($\approx$ 1.5 times the force of gravity).

[2]:
```python
def vehicle_update(t, x, u, params):
    # Get the parameters for the model
    a = params.get('refoffset', 1.5)         # offset to vehicle reference point
    b = params.get('wheelbase', 3.)          # vehicle wheelbase
    maxsteer = params.get('maxsteer', 0.5)   # max steering angle (rad)

    # Saturate the steering input
    delta = np.clip(u[1], -maxsteer, maxsteer)
    alpha = np.arctan2(a * np.tan(delta), b)

    # Return the derivative of the state
    return np.array([
        u[0] * np.cos(x[2] + alpha),     # xdot = cos(theta + alpha) v
        u[0] * np.sin(x[2] + alpha),     # ydot = sin(theta + alpha) v
        (u[0] / b) * np.tan(delta)       # thdot = v/l tan(phi)
    ])

def vehicle_output(t, x, u, params):
    return x[0:2]

# Default vehicle parameters (including nominal velocity)
vehicle_params={'refoffset': 1.5, 'wheelbase': 3, 'velocity': 15,
                'maxsteer': 0.5}

# Define the vehicle steering dynamics as an input/output system
vehicle = ct.NonlinearIOSystem(
    vehicle_update, vehicle_output, states=3, name='vehicle',
    inputs=('v', 'delta'), outputs=('x', 'y'), params=vehicle_params)
```

Vehicle driving on a curvy road (Figure 8.6a)

To illustrate the dynamics of the system, we create an input that correspond to driving down a curvy road. This trajectory will be used in future simulations as a reference trajectory for estimation and control.

RMM notes, 27 Jun 2019: * The figure below appears in Chapter 8 (output feedback) as Example 8.3, but I've put it here in the notebook since it is a good way to demonstrate the dynamics of the vehicle. * In the book, this figure is created for the linear model and in a manner that I can't quite understand, since the linear model that is used is only for the lateral dynamics. The original file is `OutputFeedback/figures/steering_obs.m`. * To create the figure here, I set the initial vehicle angle to be $\theta(0) = 0.75$ rad and then used an input that gives a figure approximating Example 8.3 To create the lateral offset, I think subtracted the trajectory from the averaged straight line trajectory, shown as a dashed line in the xy figure below. * I find the approach that we used in the MATLAB version to be confusing, but I also think the method of creating the lateral error here is a hart to follow. We might instead consider choosing a trajectory that goes mainly vertically, with the 2D dynamics being the x, θ dynamics instead of the y, θ dynamics.

KJA comments, 1 Jul 2019:

0. I think we should point out that the reference point is typically the projection of the center of mass of the whole vehicle.
1. The heading angle θ must be marked in Figure 3.17b.
2. I think it is useful to start with a curvy road that you have done here but then to specialized to a trajectory that is essentially horizontal, where y is the deviation from the nominal horizontal x axis. Assuming that α and θ are small we get the natural linearization of (3.26) $\dot{x} = v$ and $\dot{y} = v(\alpha + \theta)$

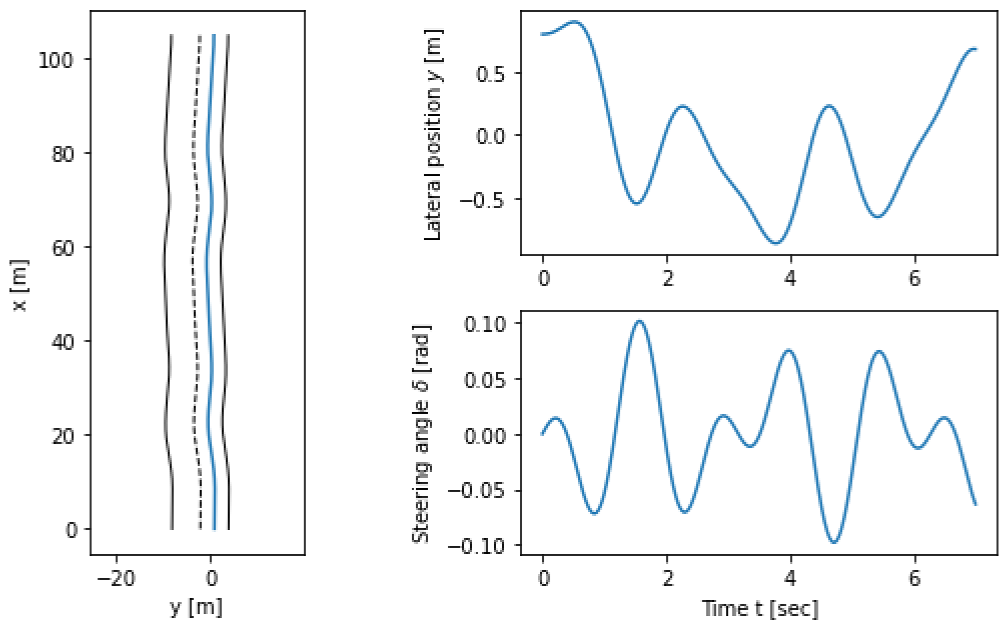
RMM response, 16 Jul 2019: * I've changed the trajectory to be about the horizontal axis, but I am ploting things vertically for better figure layout. This corresponds to what is done in Example 9.10 in the text, which I think looks OK.

KJA response, 20 Jul 2019: Fig 8.6a is fine

[3]:
```
# System parameters
wheelbase = vehicle_params['wheelbase']
v0 = vehicle_params['velocity']

# Control inputs
T_curvy = np.linspace(0, 7, 500)
v_curvy = v0*np.ones(T_curvy.shape)
delta_curvy = 0.1*np.sin(T_curvy)*np.cos(4*T_curvy) + 0.0025*np.sin(T_curvy*np.pi/7)
u_curvy = [v_curvy, delta_curvy]
X0_curvy = [0, 0.8, 0]

# Simulate the system + estimator
t_curvy, y_curvy, x_curvy = ct.input_output_response(
    vehicle, T_curvy, u_curvy, X0_curvy, params=vehicle_params, return_x=True)

# Configure matplotlib plots to be a bit bigger and optimize layout
plt.figure(figsize=[9, 4.5])

# Plot the resulting trajectory (and some road boundaries)
plt.subplot(1, 4, 2)
plt.plot(y_curvy[1], y_curvy[0])
plt.plot(y_curvy[1] - 9/np.cos(x_curvy[2]), y_curvy[0], 'k-', linewidth=1)
plt.plot(y_curvy[1] - 3/np.cos(x_curvy[2]), y_curvy[0], 'k--', linewidth=1)
plt.plot(y_curvy[1] + 3/np.cos(x_curvy[2]), y_curvy[0], 'k-', linewidth=1)

plt.xlabel('y [m]')
```
(continues on next page)

(continued from previous page)

```
plt.ylabel('x [m]');
plt.axis('Equal')

# Plot the lateral position
plt.subplot(2, 2, 2)
plt.plot(t_curvy, y_curvy[1])
plt.ylabel('Lateral position $y$ [m]')

# Plot the steering angle
plt.subplot(2, 2, 4)
plt.plot(t_curvy, delta_curvy)
plt.ylabel('Steering angle $\\delta$ [rad]')
plt.xlabel('Time t [sec]')
plt.tight_layout()
```



Linearization of lateral steering dynamics (Example 6.13)

We are interested in the motion of the vehicle about a straight-line path ($\theta = \theta_0$) with constant velocity $v_0 \neq 0$. To find the relevant equilibrium point, we first set $\dot{\theta} = 0$ and we see that we must have $\delta = 0$, corresponding to the steering wheel being straight. The motion in the xy plane is by definition not at equilibrium and so we focus on lateral deviation of the vehicle from a straight line. For simplicity, we let $\theta_e = 0$, which corresponds to driving along the x axis. We can then focus on the equations of motion in the y and θ directions with input $u = \delta$.

[4]:
```
# Define the lateral dynamics as a subset of the full vehicle steering dynamics
lateral = ct.NonlinearIOSystem(
    lambda t, x, u, params: vehicle_update(
        t, [0., x[0], x[1]], [params.get('velocity', 1), u[0]], params)[1:],
    lambda t, x, u, params: vehicle_output(
        t, [0., x[0], x[1]], [params.get('velocity', 1), u[0]], params)[1:],
    states=2, name='lateral', inputs=('phi'), outputs=('y')
)

# Compute the linearization at velocity v0 = 15 m/sec
```

(continues on next page)

(continued from previous page)

```
lateral_linearized = ct.linearize(lateral, [0, 0], [0], params=vehicle_params)

# Normalize dynamics using state [x1/b, x2] and timescale v0 t / b
b = vehicle_params['wheelbase']
v0 = vehicle_params['velocity']
lateral_transformed = ct.similarity_transform(
    lateral_linearized, [[1/b, 0], [0, 1]], timescale=v0/b)

# Set the output to be the normalized state x1/b
lateral_normalized = lateral_transformed * (1/b)
print("Linearized system dynamics:\n")
print(lateral_normalized)

# Save the system matrices for later use
A = lateral_normalized.A
B = lateral_normalized.B
C = lateral_normalized.C
```

```
Linearized system dynamics:

A = [[0. 1.]
 [0. 0.]]

B = [[0.5]
 [1. ]]

C = [[1. 0.]]

D = [[0.]]
```

Eigenvalue placement controller design (Example 7.4)

We want to design a controller that stabilizes the dynamics of the vehicle and tracks a given reference value r of the lateral position of the vehicle. We use feedback to design the dynamics of the system to have the characteristic polynomial $p(s) = s^2 + 2\zeta_c\omega_c + \omega_c^2$.

To find reasonable values of ω_c we observe that the initial response of the steering angle to a unit step change in the steering command is $\omega_c^2 r$, where r is the commanded lateral transition. Recall that the model is normalized so that the length unit is the wheelbase b and the time unit is the time b/v_0 to travel one wheelbase. A typical car has a wheelbase of about 3 m and, assuming a speed of 30 m/s, a normalized time unit corresponds to 0.1 s. To determine a reasonable steering angle when making a gentle lane change, we assume that the turning radius is R = 600 m. For a wheelbase of 3 m this corresponds to a steering angle $\delta \approx 3/600 = 0.005$ rad and a lateral acceleration of v^2/R = 302/600 = 1.5 m/s^2. Assuming that a lane change corresponds to a translation of one wheelbase we find $\omega_c = \sqrt{0.005}$ = 0.07 rad/s.

The unit step responses for the closed loop system for different values of the design parameters are shown below. The effect of ω_c is shown on the left, which shows that the response speed increases with increasing ω_c. All responses have overshoot less than 5% (15 cm), as indicated by the dashed lines. The settling times range from 30 to 60 normalized time units, which corresponds to about 3–6 s, and are limited by the acceptable lateral acceleration of the vehicle. The effect of ζ_c is shown on the right. The response speed and the overshoot increase with decreasing damping. Using these plots, we conclude that a reasonable design choice is $\omega_c = 0.07$ and $\zeta_c = 0.7$.

RMM note, 27 Jun 2019: * The design guidelines are for v_0 = 30 m/s (highway speeds) but most of the examples below are done at lower speed (typically 10 m/s). Also, the eigenvalue locations above are not the same ones that we use in the output feedback example below. We should probably make things more consistent.

KJA comment, 1 Jul 2019: * I am all for maikng it consist and choosing e.g. v0 = 30 m/s

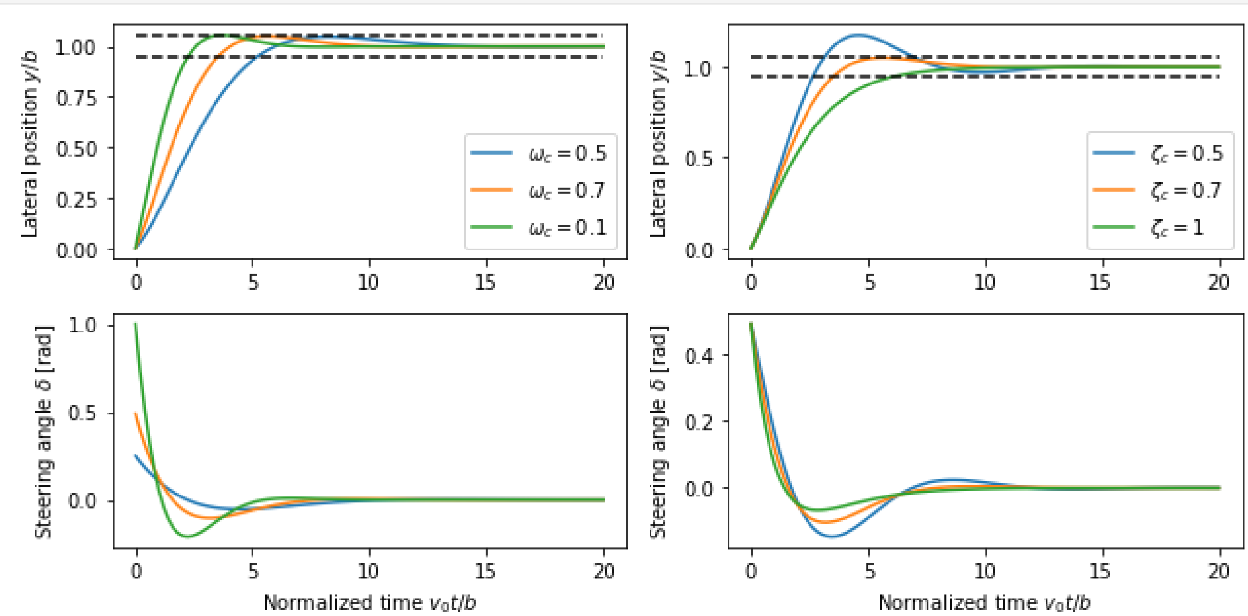
RMM comment, 17 Jul 2019: * I've updated the examples below to use v0 = 30 m/s for everything except the forward/reverse example. This corresponds to ~105 kph (freeway speeds) and a reasonable bound for the steering angle to avoid slipping is 0.05 rad.

[5]:
```
# Utility function to place poles for the normalized vehicle steering system
def normalized_place(wc, zc):
    # Get the dynamics and input matrices, for later use
    A, B = lateral_normalized.A, lateral_normalized.B

    # Compute the eigenvalues from the characteristic polynomial
    eigs = np.roots([1, 2*zc*wc, wc**2])

    # Compute the feedback gain using eigenvalue placement
    K = ct.place_varga(A, B, eigs)

    # Create a new system representing the closed loop response
    clsys = ct.StateSpace(A - B @ K, B, lateral_normalized.C, 0)

    # Compute the feedforward gain based on the zero frequency gain of the closed loop
    kf = np.real(1/clsys.evalfr(0))

    # Scale the input by the feedforward gain
    clsys *= kf

    # Return gains and closed loop system dynamics
    return K, kf, clsys

# Utility function to plot simulation results for normalized vehicle steering system
def normalized_plot(t, y, u, inpfig, outfig):
    plt.sca(outfig)
    plt.plot(t, y)
    plt.sca(inpfig)
    plt.plot(t, u[0])

# Utility function to label plots of normalized vehicle steering system
def normalized_label(inpfig, outfig):
    plt.sca(inpfig)
    plt.xlabel('Normalized time $v_0 t / b$')
    plt.ylabel('Steering angle $\delta$ [rad]')

    plt.sca(outfig)
    plt.ylabel('Lateral position $y/b$')
    plt.plot([0, 20], [0.95, 0.95], 'k--')
    plt.plot([0, 20], [1.05, 1.05], 'k--')

# Configure matplotlib plots to be a bit bigger and optimize layout
plt.figure(figsize=[9, 4.5])

# Explore range of values for omega_c, with zeta_c = 0.7
outfig = plt.subplot(2, 2, 1)
inpfig = plt.subplot(2, 2, 3)
zc = 0.7
for wc in [0.5, 0.7, 1]:
    # Place the poles of the system
    K, kf, clsys = normalized_place(wc, zc)
```

(continues on next page)

(continued from previous page)

```
    # Compute the step response
    t, y, x = ct.step_response(clsys, np.linspace(0, 20, 100), return_x=True)

    # Compute the input used to generate the control response
    u = -K @ x + kf * 1

    # Plot the results
    normalized_plot(t, y, u, inpfig, outfig)

# Add labels to the figure
normalized_label(inpfig, outfig)
plt.legend(('$\omega_c = 0.5$', '$\omega_c = 0.7$', '$\omega_c = 0.1$'))

# Explore range of values for zeta_c, with omega_c = 0.07
outfig = plt.subplot(2, 2, 2)
inpfig = plt.subplot(2, 2, 4)
wc = 0.7
for zc in [0.5, 0.7, 1]:
    # Place the poles of the system
    K, kf, clsys = normalized_place(wc, zc)

    # Compute the step response
    t, y, x = ct.step_response(clsys, np.linspace(0, 20, 100), return_x=True)

    # Compute the input used to generate the control response
    u = -K @ x + kf * 1

    # Plot the results
    normalized_plot(t, y, u, inpfig, outfig)

# Add labels to the figure
normalized_label(inpfig, outfig)
plt.legend(('$\zeta_c = 0.5$', '$\zeta_c = 0.7$', '$\zeta_c = 1$'))
plt.tight_layout()
```



RMM notes, 17 Jul 2019 * These step responses are *very* slow. Note that the steering wheel angles are about 10X less than a resonable bound (0.05 rad at 30 m/s). A consequence of these low gains is that the tracking controller in Example 8.4 has to use a different set of gains. We could update, but the gains listed here have a rationale that we would have to update as well. * Based on the discussion below, I think we should make ω_c range from 0.5 to 1 (10X faster).

KJA response, 20 Jul 2019: Makes a lot of sense to make ω_c range from 0.5 to 1 (10X faster). The plots were still in the range 0.05 to 0.1 in the note you sent me.

RMM response: 23 Jul 2019: Updated ω_c to 10X faster. Note that this makes size of the inputs for the step response quite large, but that is in part because a unit step in the desired position produces an (instantaneous) error of $b = 3$ m $\implies$ quite a large error. A lateral error of 10 cm with $\omega_c = 0.7$ would produce an (initial) input of 0.015 rad.

Eigenvalue placement observer design (Example 8.3)

We construct an estimator for the (normalized) lateral dynamics by assigning the eigenvalues of the estimator dynamics to desired value, specified in terms of the second order characteristic equation for the estimator dynamics.

[6]:
```
# Find the eigenvalue from the characteristic polynomial
wo = 1          # bandwidth for the observer
zo = 0.7        # damping ratio for the observer
eigs = np.roots([1, 2*zo*wo, wo**2])

# Compute the estimator gain using eigenvalue placement
L = np.transpose(
    ct.place(np.transpose(A), np.transpose(C), eigs))
print("L = ", L)

# Create a linear model of the lateral dynamics driving the estimator
est = ct.StateSpace(A - L @ C, np.block([[B, L]]), np.eye(2), np.zeros((2,2)))
```

```
L =  [[1.4]
 [1. ]]
```

Linear observer applied to nonlinear system output

A simulation of the observer for a vehicle driving on a curvy road is shown below. The first figure shows the trajectory of the vehicle on the road, as viewed from above. The response of the observer is shown on the right, where time is normalized to the vehicle length. We see that the observer error settles in about 4 vehicle lengths.

RMM note, 27 Jun 2019: * As an alternative, we can attempt to estimate the state of the full nonlinear system using a linear estimator. This system does not necessarily converge to zero since there will be errors in the nominal dynamics of the system for the linear estimator. * The limits on the x axis for the time plots are different to show the error over the entire trajectory. * We should decide whether we want to keep the figure above or the one below for the text.

KJA comment, 1 Jul 2019: * I very much like your observation about the nonlinear system. I think it is a very good idea to use your new simulation

RMM comment, 17 Jul 2019: plan to use this version in the text.

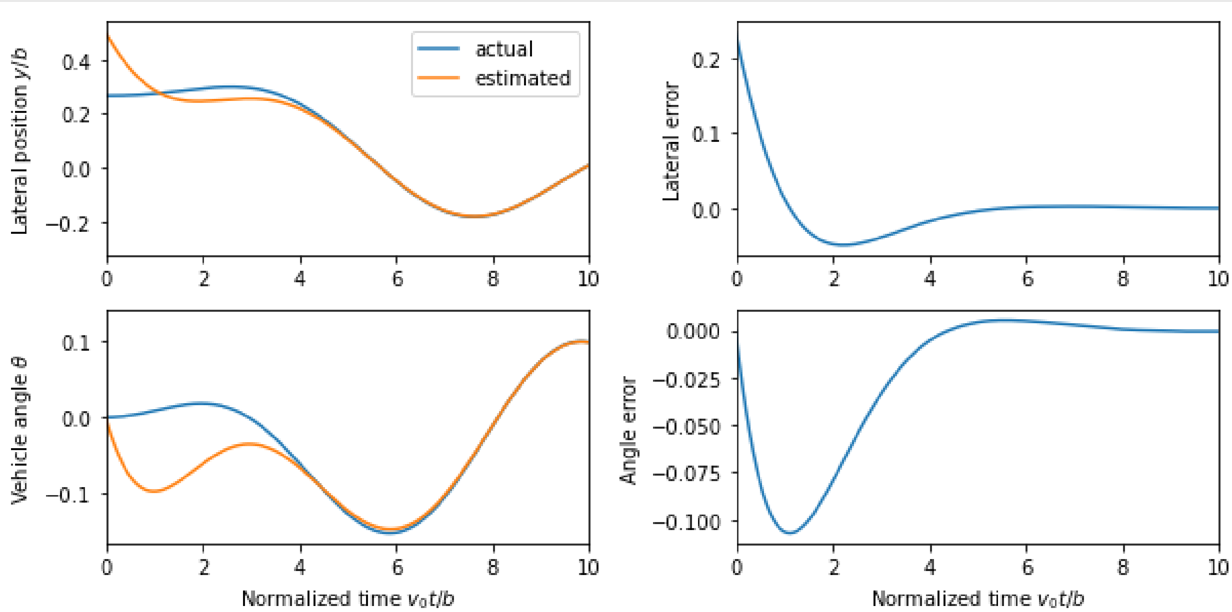
KJA comment, 20 Jul 2019: I think this is a big improvement we show that an observer based on a linearized model works on a nonlinear simulation, If possible we could add a line telling why the linear model works and that this is standard procedure in control engineering.

[7]:
```
# Convert the curvy trajectory into normalized coordinates
x_ref = x_curvy[0] / wheelbase
y_ref = x_curvy[1] / wheelbase
```

(continues on next page)

(continued from previous page)

```
theta_ref = x_curvy[2]
tau = v0 * T_curvy / b

# Simulate the estimator, with a small initial error in y position
t, y_est, x_est = ct.forced_response(est, tau, [delta_curvy, y_ref], [0.5, 0])

# Configure matplotlib plots to be a bit bigger and optimize layout
plt.figure(figsize=[9, 4.5])

# Plot the actual and estimated states
ax = plt.subplot(2, 2, 1)
plt.plot(t, y_ref)
plt.plot(t, x_est[0])
ax.set(xlim=[0, 10])
plt.legend(['actual', 'estimated'])
plt.ylabel('Lateral position $y/b$')

ax = plt.subplot(2, 2, 2)
plt.plot(t, x_est[0] - y_ref)
ax.set(xlim=[0, 10])
plt.ylabel('Lateral error')

ax = plt.subplot(2, 2, 3)
plt.plot(t, theta_ref)
plt.plot(t, x_est[1])
ax.set(xlim=[0, 10])
plt.xlabel('Normalized time $v_0 t / b$')
plt.ylabel('Vehicle angle $\\theta$')

ax = plt.subplot(2, 2, 4)
plt.plot(t, x_est[1] - theta_ref)
ax.set(xlim=[0, 10])
plt.xlabel('Normalized time $v_0 t / b$')
plt.ylabel('Angle error')
plt.tight_layout()
```



Output Feedback Controller (Example 8.4)

RMM note, 27 Jun 2019 * The feedback gains for the controller below are different that those computed in the eigenvalue placement example (from Ch 7), where an argument was given for the choice of the closed loop eigenvalues. Should we choose a single, consistent set of gains in both places? * This plot does not quite match Example 8.4 because a different reference is being used for the laterial position. * The transient in δ is quiet large. This appears to be due to the error in $\theta(0)$, which is initialized to zero intead of to `theta_curvy`.

KJA comment, 1 Jul 2019: 1. The large initial errors dominate the plots.

2. There is somehing funny happening at the end of the simulation, may be due to the small curvature at the end of the path?

RMM comment, 17 Jul 2019: * Updated to use the new trajectory * We will have the issue that the gains here are different than the gains that we used in Chapter 7. I think that what we need to do is update the gains in Ch 7 (they are too sluggish, as noted above). * Note that unlike the original example in the book, the errors do not converge to zero. This is because we are using pure state feedback (no feedforward) => the controller doesn't apply any input until there is an error.

KJA comment, 20 Jul 2019: We may add that state feedback is a proportional controller which does not guarantee that the error goes to zero for example by changing the line "The tracking error …" to "The tracking error can be improved by adding integral action (Section7.4), later in this chapter "Disturbance Modeling" or feedforward (Section 8,5). Should we do an exercises?

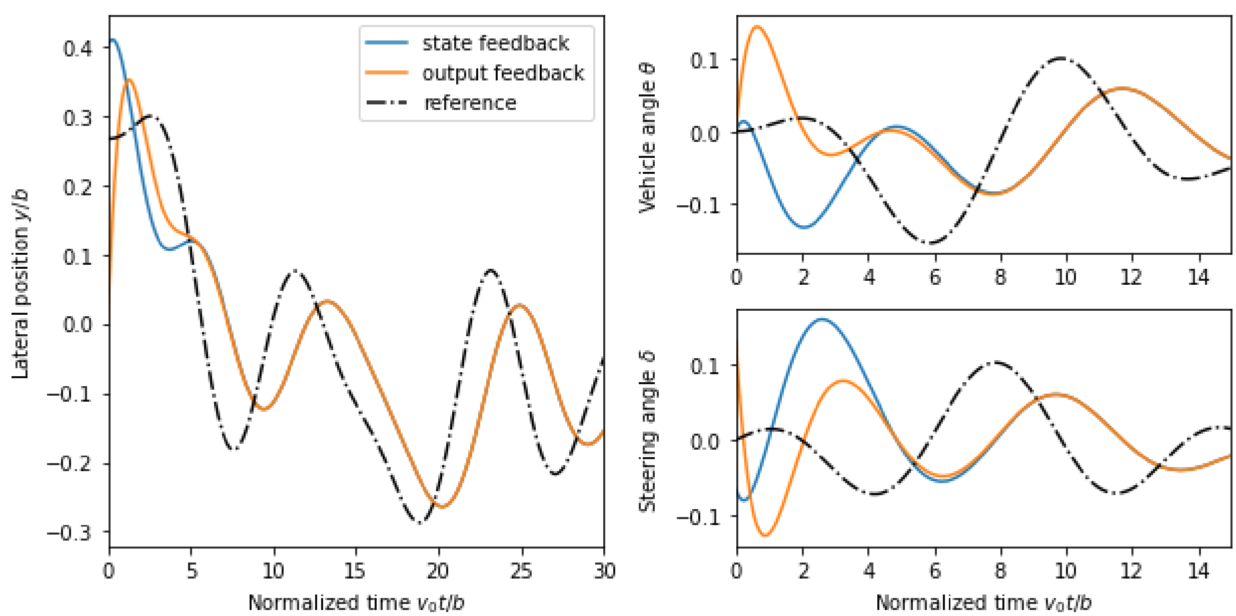
[8]:
```
# Compute the feedback gains
# K, kf, clsys = normalized_place(1, 0.707)     # Gains from MATLAB
# K, kf, clsys = normalized_place(0.07, 0.707)  # Original gains
K, kf, clsys = normalized_place(0.7, 0.707)       # Final gains

# Print out the gains
print("K = ", K)
print("kf = ", kf)

# Construct an output-based controller for the system
clsys = ct.StateSpace(
    np.block([[A, -B@K], [L@C, A - B@K - L@C]]),
    np.block([[B], [B]]) * kf,
    np.block([[C, np.zeros(C.shape)], [np.zeros(C.shape), C]]),
    np.zeros((2,1)))

# Simulate the system
t, y, x = ct.forced_response(clsys, tau, y_ref, [0.4, 0, 0.0, 0])

# Calcaluate the input used to generate the control response
u_sfb = kf * y_ref - K @ x[0:2]
u_ofb = kf * y_ref - K @ x[2:4]

# Configure matplotlib plots to be a bit bigger and optimize layout
plt.figure(figsize=[9, 4.5])

# Plot the actual and estimated states
ax = plt.subplot(1, 2, 1)
plt.plot(t, x[0])
plt.plot(t, x[2])
plt.plot(t, y_ref, 'k-.')
ax.set(xlim=[0, 30])
plt.legend(['state feedback', 'output feedback', 'reference'])
plt.xlabel('Normalized time $v_0 t / b$')
```

(continues on next page)

(continued from previous page)

```
plt.ylabel('Lateral position $y/b$')

ax = plt.subplot(2, 2, 2)
plt.plot(t, x[1])
plt.plot(t, x[3])
plt.plot(t, theta_ref, 'k-.')
ax.set(xlim=[0, 15])
plt.ylabel('Vehicle angle $\\theta$')

ax = plt.subplot(2, 2, 4)
plt.plot(t, u_sfb[0])
plt.plot(t, u_ofb[0])
plt.plot(t, delta_curvy, 'k-.')
ax.set(xlim=[0, 15])
plt.xlabel('Normalized time $v_0 t / b$')
plt.ylabel('Steering angle $\\delta$')
plt.tight_layout()
```

```
K =  [[0.49   0.7448]]
kf =  [[0.49]]
```



Trajectory Generation (Example 8.8)

To illustrate how we can use a two degree-of-freedom design to improve the performance of the system, consider the problem of steering a car to change lanes on a road. We use the non-normalized form of the dynamics, which were derived in Example 3.11.

KJA comment, 1 Jul 2019: 1. I think the reference trajectory is too much curved in the end compare with Example 3.11

In summary I think it is OK to change the reference trajectories but we should make sure that the curvature is less than $\rho = 600m$ not to have too high acceleratarion.

RMM response, 16 Jul 2019: * Not sure if the comment about the trajectory being too curved is referring to this example. The steering angles (and hence radius of curvature/acceleration) are quite low. ??

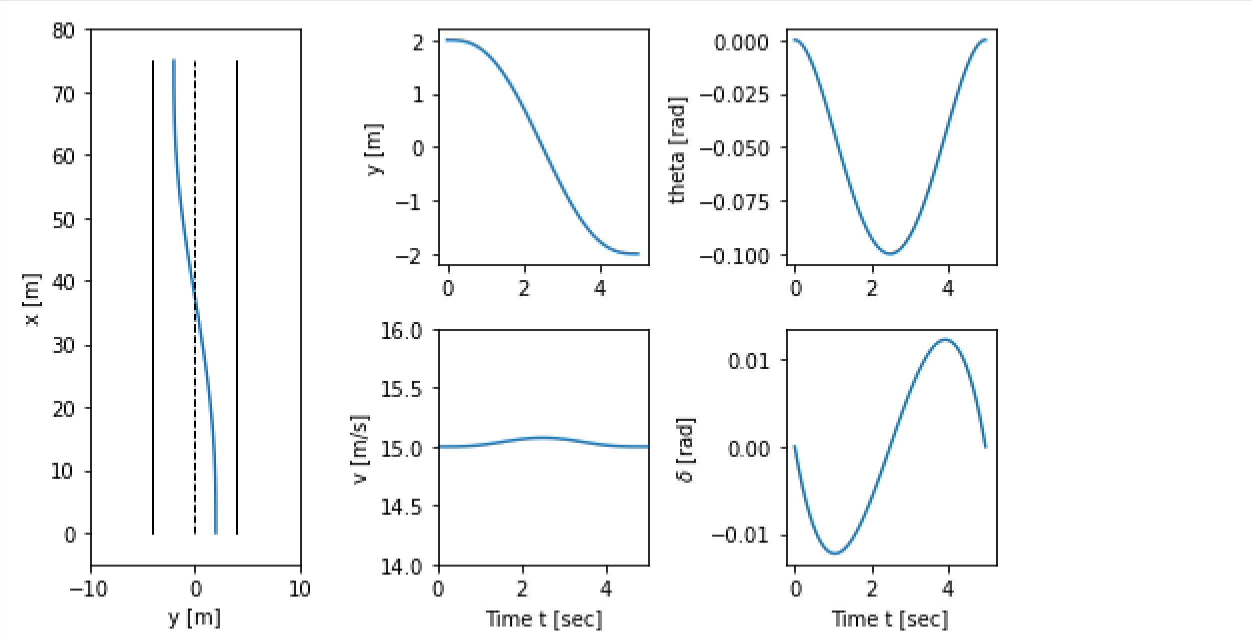
KJA response, 20 Jul 2019: You are right the curvature is not too small. We could add the sentence "The small deviations can be eliminated by adding feedback."

RMM response, 23 Jul 2019: I think the small deviation you are referring to is in the velocity trace. This occurs because I gave a fixed endpoint in time and so the velocity had to be adjusted to hit that exact point at that time. This doesn't show up in the book, so it won't be a problem ($\implies$ no additional explanation required).

[9]:
```python
import control.flatsys as fs

# Function to take states, inputs and return the flat flag
def vehicle_flat_forward(x, u, params={}):
    # Get the parameter values
    b = params.get('wheelbase', 3.)

    # Create a list of arrays to store the flat output and its derivatives
    zflag = [np.zeros(3), np.zeros(3)]

    # Flat output is the x, y position of the rear wheels
    zflag[0][0] = x[0]
    zflag[1][0] = x[1]

    # First derivatives of the flat output
    zflag[0][1] = u[0] * np.cos(x[2])   # dx/dt
    zflag[1][1] = u[0] * np.sin(x[2])   # dy/dt

    # First derivative of the angle
    thdot = (u[0]/b) * np.tan(u[1])

    # Second derivatives of the flat output (setting vdot = 0)
    zflag[0][2] = -u[0] * thdot * np.sin(x[2])
    zflag[1][2] =  u[0] * thdot * np.cos(x[2])

    return zflag

# Function to take the flat flag and return states, inputs
def vehicle_flat_reverse(zflag, params={}):
    # Get the parameter values
    b = params.get('wheelbase', 3.)

    # Create a vector to store the state and inputs
    x = np.zeros(3)
    u = np.zeros(2)

    # Given the flat variables, solve for the state
    x[0] = zflag[0][0]  # x position
    x[1] = zflag[1][0]  # y position
    x[2] = np.arctan2(zflag[1][1], zflag[0][1])  # tan(theta) = ydot/xdot

    # And next solve for the inputs
    u[0] = zflag[0][1] * np.cos(x[2]) + zflag[1][1] * np.sin(x[2])
    thdot_v = zflag[1][2] * np.cos(x[2]) - zflag[0][2] * np.sin(x[2])
    u[1] = np.arctan2(thdot_v, u[0]**2 / b)

    return x, u

vehicle_flat = fs.FlatSystem(vehicle_flat_forward, vehicle_flat_reverse, inputs=2,␣
↪states=3)
```

To find a trajectory from an initial state x_0 to a final state x_f in time T_f we solve a point-to-point trajectory generation problem. We also set the initial and final inputs, which sets the vehicle velocity v and steering wheel angle δ at the endpoints.

[10]:
```
# Define the endpoints of the trajectory
x0 = [0., 2., 0.]; u0 = [15, 0.]
xf = [75, -2., 0.]; uf = [15, 0.]
Tf = xf[0] / uf[0]

# Define a set of basis functions to use for the trajectories
poly = fs.PolyFamily(6)

# Find a trajectory between the initial condition and the final condition
traj = fs.point_to_point(vehicle_flat, x0, u0, xf, uf, Tf, basis=poly)

# Create the trajectory
t = np.linspace(0, Tf, 100)
x, u = traj.eval(t)

# Configure matplotlib plots to be a bit bigger and optimize layout
plt.figure(figsize=[9, 4.5])

# Plot the trajectory in xy coordinate
plt.subplot(1, 4, 2)
plt.plot(x[1], x[0])
plt.xlabel('y [m]')
plt.ylabel('x [m]')

# Add lane lines and scale the axis
plt.plot([-4, -4], [0, x[0, -1]], 'k-', linewidth=1)
plt.plot([0, 0], [0, x[0, -1]], 'k--', linewidth=1)
plt.plot([4, 4], [0, x[0, -1]], 'k-', linewidth=1)
plt.axis([-10, 10, -5, x[0, -1] + 5])

# Time traces of the state and input
plt.subplot(2, 4, 3)
plt.plot(t, x[1])
plt.ylabel('y [m]')

plt.subplot(2, 4, 4)
plt.plot(t, x[2])
plt.ylabel('theta [rad]')

plt.subplot(2, 4, 7)
plt.plot(t, u[0])
plt.xlabel('Time t [sec]')
plt.ylabel('v [m/s]')
plt.axis([0, Tf, u0[0] - 1, uf[0] +1])

plt.subplot(2, 4, 8)
plt.plot(t, u[1]);
plt.xlabel('Time t [sec]')
plt.ylabel('$\delta$ [rad]')
plt.tight_layout()
```



Vehicle transfer functions for forward and reverse driving (Example 10.11)

The vehicle steering model has different properties depending on whether we are driving forward or in reverse. The figures below show step responses from steering angle to lateral translation for a the linearized model when driving forward (dashed) and reverse (solid). In this simulation we have added an extra pole with the time constant $T = 0.1$ to approximately account for the dynamics in the steering system.

With rear-wheel steering the center of mass first moves in the wrong direction and the overall response with rear-wheel steering is significantly delayed compared with that for front-wheel steering. (b) Frequency response for driving forward (dashed) and reverse (solid). Notice that the gain curves are identical, but the phase curve for driving in reverse has non-minimum phase.

RMM note, 27 Jun 2019: * I cannot recreate the figures in Example 10.11. Since we are looking at the lateral *velocity*, there is a differentiator in the output and this takes the step function and creates an offset at $t = 0$ (intead of a smooth curve). * The transfer functions are also different, and I don't quite understand why. Need to spend a bit more time on this one.

KJA comment, 1 Jul 2019: The reason why you cannot recreate figures i Example 10.11 is because the caption in figure is wrong, sorry my fault, the y-axis should be lateral position not lateral velocity. The approximate expression for the transfer functions

$$G_{y\delta} = \frac{av_0s + v_0^2}{bs} = \frac{1.5s + 1}{3s^2} = \frac{0.5s + 0.33}{s}$$

are quite close to the values that you get numerically

In this case I think it is useful to have v=1 m/s because we do not drive to fast backwards.

RMM response, 17 Jul 2019 * Updated figures below use the same parameters as the running example (the current text uses different parameters) * Following the material in the text, a pole is added at s = -1 to approximate the dynamics of the steering system. This is not strictly needed, so we could decide to take it out (and update the text)

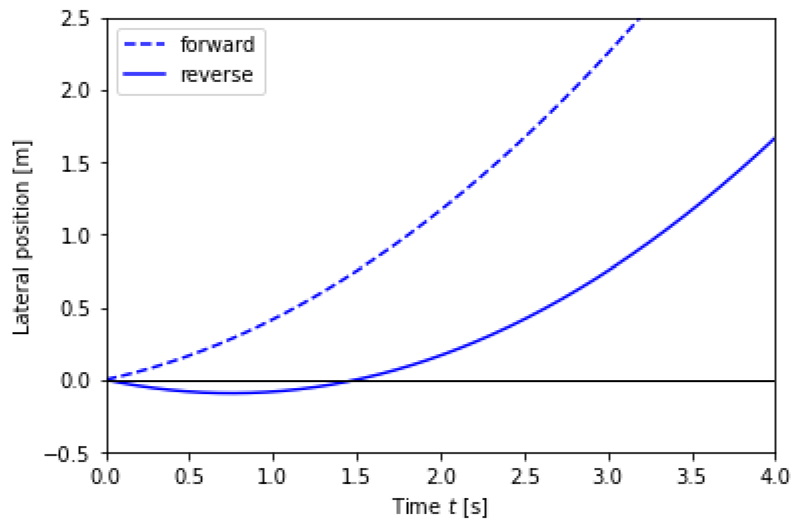
KJA comment, 20 Jul 2019: I have been oscillating a bit about this example. Of course it does not make sense to drive in reverse in 30 m/s but it seems a bit silly to change parameters just in this case (if we do we have to motivate it). On the other hand what we are doing is essentially based on transfer functions and a RHP zero. My current view which has changed a few times is to keep the standard parameters. In any case we should eliminate the extra time constant. A small detail, I could not see the time response in the file you sent, do not resend it!, I will look at the final version.

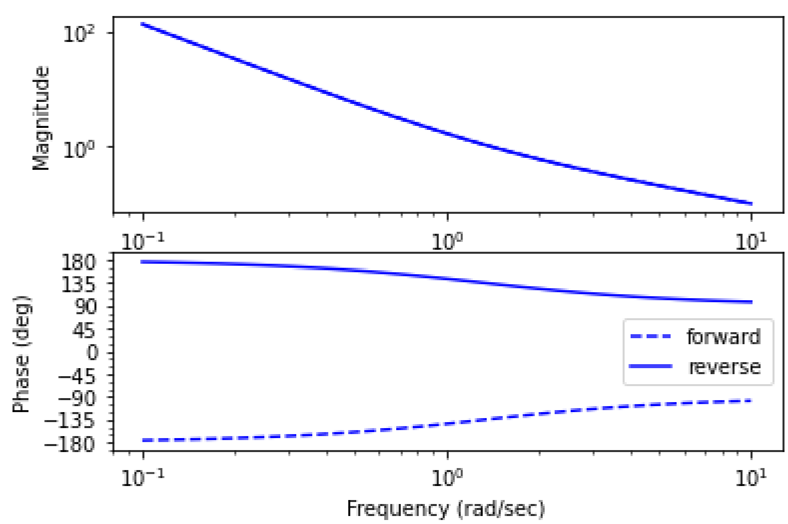
RMM comment, 23 Jul 2019: I think it is OK to have the speed be different and just talk about this in the text. I have removed the extra time constant in the current version.

[11]:
```
# Magnitude of the steering input (half maximum)
Msteer = vehicle_params['maxsteer'] / 2

# Create a linearized model of the system going forward at 2 m/s
forward_lateral = ct.linearize(lateral, [0, 0], [0], params={'velocity': 2})
forward_tf = ct.ss2tf(forward_lateral)[0, 0]
print("Forward TF = ", forward_tf)

# Create a linearized model of the system going in reverise at 1 m/s
reverse_lateral = ct.linearize(lateral, [0, 0], [0], params={'velocity': -2})
reverse_tf = ct.ss2tf(reverse_lateral)[0, 0]
print("Reverse TF = ", reverse_tf)
```

```
Forward TF =
          s + 1.333
-----------------------------
s^2 + 7.828e-16 s - 1.848e-16

Reverse TF =
          -s + 1.333
-----------------------------
s^2 - 7.828e-16 s - 1.848e-16
```

[12]:
```
# Configure matplotlib plots to be a bit bigger and optimize layout
plt.figure()

# Forward motion
t, y = ct.step_response(forward_tf * Msteer, np.linspace(0, 4, 500))
plt.plot(t, y, 'b--')

# Reverse motion
t, y = ct.step_response(reverse_tf * Msteer, np.linspace(0, 4, 500))
plt.plot(t, y, 'b-')

# Add labels and reference lines
plt.axis([0, 4, -0.5, 2.5])
plt.legend(['forward', 'reverse'], loc='upper left')
plt.xlabel('Time $t$ [s]')
plt.ylabel('Lateral position [m]')
plt.plot([0, 4], [0, 0], 'k-', linewidth=1)

# Plot the Bode plots
plt.figure()
plt.subplot(1, 2, 2)
ct.bode_plot(forward_tf[0, 0], np.logspace(-1, 1, 100), color='b', linestyle='--')
ct.bode_plot(reverse_tf[0, 0], np.logspace(-1, 1, 100), color='b', linestyle='-')
plt.legend(('forward', 'reverse'));
```





Feedforward Compensation (Example 12.6)

For a lane transfer system we would like to have a nice response without overshoot, and we therefore consider the use of feedforward compensation to provide a reference trajectory for the closed loop system. We choose the desired response as $F_{\mathrm{m}}(s) = a^22/(s+a)^2$, where the response speed or aggressiveness of the steering is governed by the parameter a.

RMM note, 27 Jun 2019: * a was used in the original description of the dynamics as the reference offset. Perhaps choose a different symbol here? * In current version of Ch 12, the y axis is labeled in absolute units, but it should actually be in normalized units, I think. * The steering angle input for this example is quite high. Compare to Example 8.8, above. Also, we should probably make the size of the "lane change" from this example match whatever we use in Example 8.8

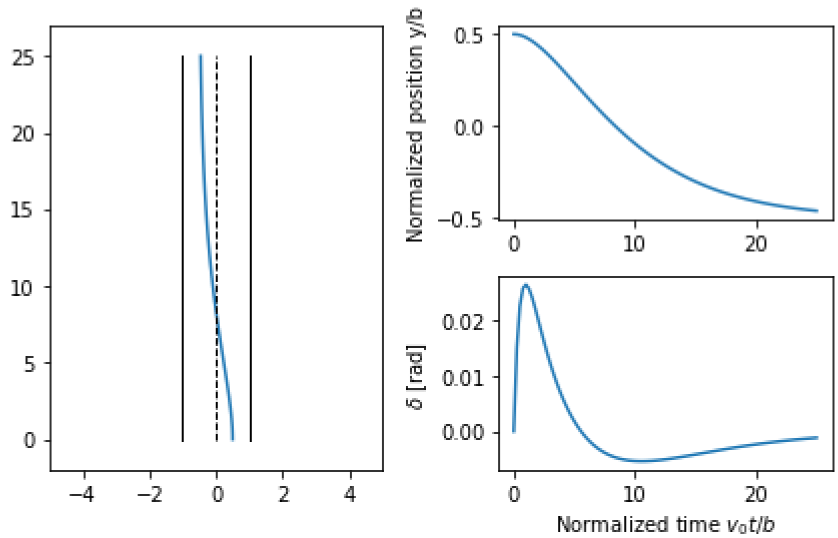
KJA comments, 1 Jul 2019: Chosen parameters look good to me

RMM response, 17 Jul 2019 * I changed the time constant for the feedforward model to give something that is more reasonable in terms of turning angle at the speed of $v_0 = 30$ m/s. Note that this takes about 30 body lengths to change lanes (= 9 seconds at 105 kph). * The time to change lanes is about 2X what it is using the differentially flat trajectory above. This is mainly because the feedback controller applies a large pulse at the beginning of the trajectory (based on the input error), whereas the differentially flat trajectory spreads the turn over a longer interval. Since are living the steering angle, we have to limit the size of the pulse => slow down the time constant for the reference model.

KJA response, 20 Jul 2019: I think the time for lane change is too long, which may depend on the small steering angles used. The largest steering angle is about 0.03 rad, but we have admitted larger values in previous examples. I suggest that we change the design so that the largest sterring angel is closer to 0.05, see the remark from Bjorn O a lane change could take about 5 s at 30m/s.

RMM response, 23 Jul 2019: I reset the time constant to 0.2, which gives something closer to what we had for trajectory generation. It is still slower, but this is to be expected since it is a linear controller. We now finish the trajectory in 20 body lengths, which is about 6 seconds.

[13]:
```
# Define the desired response of the system
a = 0.2
P = ct.ss2tf(lateral_normalized)
Fm = ct.TransferFunction([a**2], [1, 2*a, a**2])
Fr = Fm / P

# Compute the step response of the feedforward components
t, y_ffwd = ct.step_response(Fm, np.linspace(0, 25, 100))
t, delta_ffwd = ct.step_response(Fr, np.linspace(0, 25, 100))

# Scale and shift to correspond to lane change (-2 to +2)
y_ffwd = 0.5 - 1 * y_ffwd
delta_ffwd *= 1

# Overhead view
plt.subplot(1, 2, 1)
plt.plot(y_ffwd, t)
plt.plot(-1*np.ones(t.shape), t, 'k-', linewidth=1)
plt.plot(0*np.ones(t.shape), t, 'k--', linewidth=1)
plt.plot(1*np.ones(t.shape), t, 'k-', linewidth=1)
plt.axis([-5, 5, -2, 27])

# Plot the response
plt.subplot(2, 2, 2)
plt.plot(t, y_ffwd)
# plt.axis([0, 10, -5, 5])
plt.ylabel('Normalized position y/b')

plt.subplot(2, 2, 4)
plt.plot(t, delta_ffwd)
# plt.axis([0, 10, -1, 1])
plt.ylabel('$\\delta$ [rad]')
plt.xlabel('Normalized time $v_0 t / b$');

plt.tight_layout()
```



Fundamental Limits (Example 14.13)

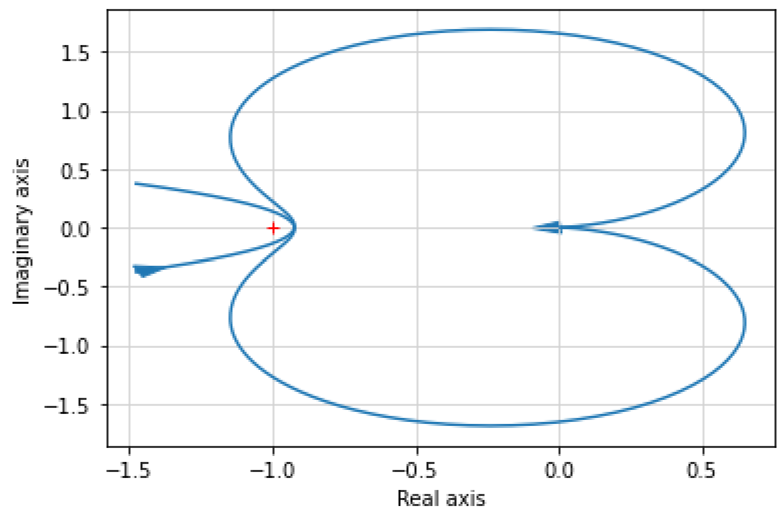
Consider a controller based on state feedback combined with an observer where we want a faster closed loop system and choose $\omega_c = 10$, $\zeta_c = 0.707$, $\omega_o = 20$, and $\zeta_o = 0.707$.

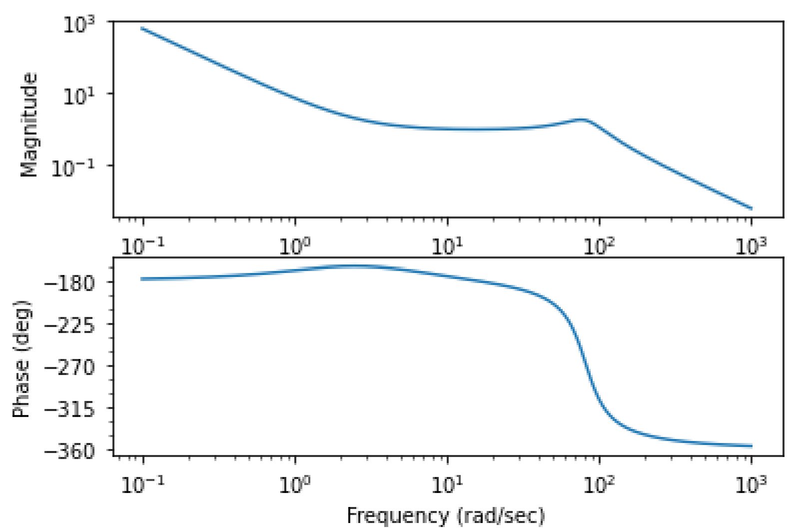
KJA comment, 20 Jul 2019: This is a really troublesome case. If we keep it as a vehicle steering problem we must have an order of magnitude lower valuer for ω_c and ω_o and then the zero will not be slow. My recommendation is to keep it as a general system with the transfer function. $P(s) = (s + 1)/s^2$. The text then has to be reworded.

RMM response, 23 Jul 2019: I think the way we have it is OK. Our current value for the controller and observer is $\omega_c = 0.7$ and $\omega_o = 1$. Here we way we want something faster and so we got to $\omega_c = 7$ (10X) and $\omega_o = 10$ (10X).

[14]:
```
# Compute the feedback gain using eigenvalue placement
wc = 10
zc = 0.707
eigs = np.roots([1, 2*zc*wc, wc**2])
K = ct.place(A, B, eigs)
kr = np.real(1/clsys.evalfr(0))
print("K = ", np.squeeze(K))

# Compute the estimator gain using eigenvalue placement
wo = 20
zo = 0.707
eigs = np.roots([1, 2*zo*wo, wo**2])
L = np.transpose(
    ct.place(np.transpose(A), np.transpose(C), eigs))
print("L = ", np.squeeze(L))

# Construct an output-based controller for the system
C1 = ct.ss2tf(ct.StateSpace(A - B@K - L@C, L, K, 0))
print("C(s) = ", C1)

# Compute the loop transfer function and plot Nyquist, Bode
L1 = P * C1
plt.figure(); ct.nyquist_plot(L1, np.logspace(0.5, 3, 500))
plt.figure(); ct.bode_plot(L1, np.logspace(-1, 3, 500));
```

```
K =  [100.   -35.86]
L =  [ 28.28 400.  ]
C(s) =
-1.152e+04 s + 4e+04
--------------------
s^2 + 42.42 s + 6658
```



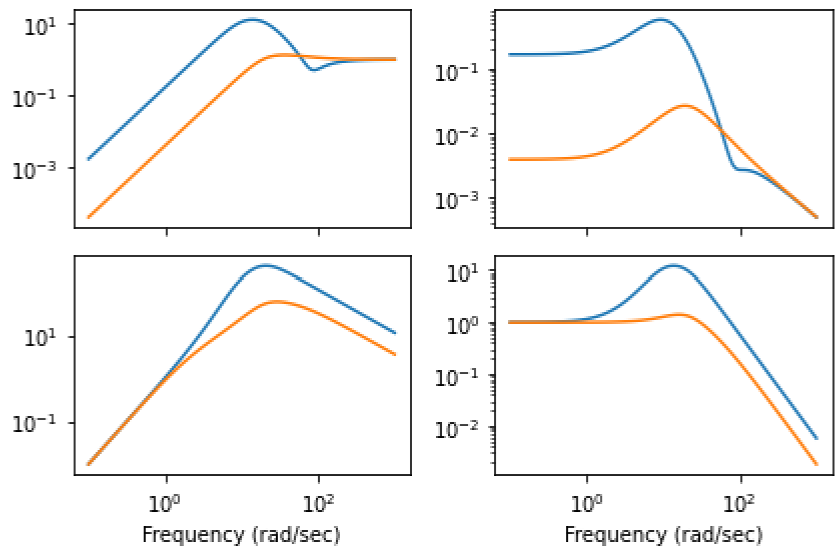


[15]:
```
# Modified control law
wc = 10
zc = 2.6
eigs = np.roots([1, 2*zc*wc, wc**2])
K = ct.place(A, B, eigs)
kr = np.real(1/clsys.evalfr(0))
print("K = ", np.squeeze(K))

# Construct an output-based controller for the system
C2 = ct.ss2tf(ct.StateSpace(A - B@K - L@C, L, K, 0))
print("C(s) = ", C2)
```

```
K =  [100.   2.]
C(s) =
    3628 s + 4e+04
---------------------
s^2 + 80.28 s + 156.6
```

[16]:

```
# Plot the gang of four for the two designs
ct.gangof4(P, C1, np.logspace(-1, 3, 100))
ct.gangof4(P, C2, np.logspace(-1, 3, 100))
```



[ ]:

8.2.3 Vertical takeoff and landing aircraft

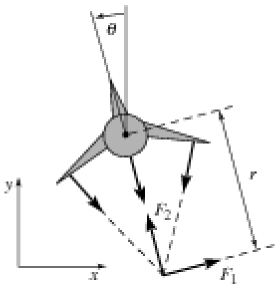
This notebook demonstrates the use of the python-control package for analysis and design of a controller for a vectored thrust aircraft model that is used as a running example through the text *Feedback Systems* by Astrom and Murray. This example makes use of MATLAB compatible commands.

Additional information on this system is available at

http://www.cds.caltech.edu/~murray/wiki/index.php/Python-control/Example:_Vertical_takeoff_and_landing_aircraft

System Description

This example uses a simplified model for a (planar) vertical takeoff and landing aircraft (PVTOL), as shown below:



$$m\ddot{x} = F_1 \cos\theta - F_2 \sin\theta - c\dot{x},$$
$$m\ddot{y} = F_1 \sin\theta + F_2 \cos\theta - mg - c\dot{y},$$
$$J\ddot{\theta} = rF_1.$$

The position and orientation of the center of mass of the aircraft is denoted by (x, y, θ), m is the mass of the vehicle, J the moment of inertia, g the gravitational constant and c the damping coefficient. The forces generated by the main downward thruster and the maneuvering thrusters are modeled as a pair of forces F_1 and F_2 acting at a distance r below the aircraft (determined by the geometry of the thrusters).

Letting $z = (x, y, \theta, \dot{x}, \dot{y}, \dot{\theta})$, the equations can be written in state space form as:

$$\frac{dz}{dt} = \begin{bmatrix} z_4 \\ z_5 \\ z_6 \\ -\frac{c}{m} z_4 \\ -g - \frac{c}{m} z_5 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \\ \frac{1}{m} \cos\theta F_1 + \frac{1}{m} \sin\theta F_2 \\ \frac{1}{m} \sin\theta F_1 + \frac{1}{m} \cos\theta F_2 \\ \frac{r}{J} F_1 \end{bmatrix}$$

LQR state feedback controller

This section demonstrates the design of an LQR state feedback controller for the vectored thrust aircraft example. This example is pulled from Chapter 6 (Linear Systems, Example 6.4) and Chapter 7 (State Feedback, Example 7.9) of Astrom and Murray. The python code listed here are contained the the file pvtol-lqr.py.

To execute this example, we first import the libraries for SciPy, MATLAB plotting and the python-control package:

[1]:
```
from numpy import *              # Grab all of the NumPy functions
from matplotlib.pyplot import * # Grab MATLAB plotting functions
from control.matlab import *    # MATLAB-like functions
%matplotlib inline
```

The parameters for the system are given by

[2]:
```
m = 4                          # mass of aircraft
J = 0.0475                     # inertia around pitch axis
r = 0.25                       # distance to center of force
g = 9.8                        # gravitational constant
c = 0.05                       # damping factor (estimated)
```

Choosing equilibrium inputs to be $u_e = (0, mg)$, the dynamics of the system $\frac{dz}{dt}$, and their linearization A about equilibrium point $z_e = (0, 0, 0, 0, 0, 0)$ are given by

$$\frac{dz}{dt} = \begin{bmatrix} z_4 \\ z_5 \\ z_6 \\ -g\sin z_3 - \frac{c}{m} z_4 \\ g(\cos z_3 - 1) - \frac{c}{m} z_5 \\ 0 \end{bmatrix} \qquad A = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & -g & -c/m & 0 & 0 \\ 0 & 0 & 0 & 0 & -c/m & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

[3]:
```
# State space dynamics
xe = [0, 0, 0, 0, 0, 0]          # equilibrium point of interest
ue = [0, m*g]                    # (note these are lists, not matrices)
```

[4]:
```
# Dynamics matrix (use matrix type so that * works for multiplication)
# Note that we write A and B here in full generality in case we want
# to test different xe and ue.
A = matrix(
   [[ 0,    0,    0,    1,    0,    0],
    [ 0,    0,    0,    0,    1,    0],
    [ 0,    0,    0,    0,    0,    1],
    [ 0, 0, (-ue[0]*sin(xe[2]) - ue[1]*cos(xe[2]))/m, -c/m, 0, 0],
    [ 0, 0, (ue[0]*cos(xe[2]) - ue[1]*sin(xe[2]))/m, 0, -c/m, 0],
    [ 0,    0,    0,    0,    0,    0 ]])

# Input matrix
B = matrix(
   [[0, 0], [0, 0], [0, 0],
    [cos(xe[2])/m, -sin(xe[2])/m],
    [sin(xe[2])/m,  cos(xe[2])/m],
    [r/J, 0]])

# Output matrix
C = matrix([[1, 0, 0, 0, 0, 0], [0, 1, 0, 0, 0, 0]])
D = matrix([[0, 0], [0, 0]])
```

To compute a linear quadratic regulator for the system, we write the cost function as

$$J = \int_0^\infty (\xi^T Q_\xi \xi + v^T Q_v v) dt,$$

where $\xi = z - z_e$ and $v = u - u_e$ represent the local coordinates around the desired equilibrium point (z_e, u_e). We begin with diagonal matrices for the state and input costs:

[5]:
```
Qx1 = diag([1, 1, 1, 1, 1, 1])
Qu1a = diag([1, 1])
(K, X, E) = lqr(A, B, Qx1, Qu1a); K1a = matrix(K)
```

This gives a control law of the form $v = -K\xi$, which can then be used to derive the control law in terms of the original variables:

$$u = v + u_e = -K(z - z_d) + u_d.$$
$$where : math : `u_e = (0, mg)`and : math : `z_d = (x_d, y_d, 0, 0, 0, 0)`$$

The way we setup the dynamics above, A is already hardcoding u_d, so we don't need to include it as an external input. So we just need to cascade the $-K(z - z_d)$ controller with the PVTOL aircraft's dynamics to control it. For didactic purposes, we will cheat in two small ways:

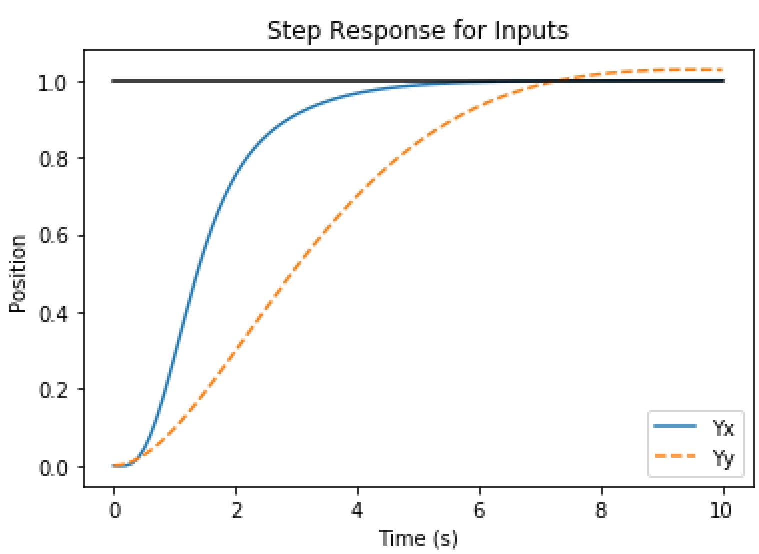
- First, we will only interface our controller with the linearized dynamics. Using the nonlinear dynamics would require the `NonlinearIOSystem` functionalities, which we leave to another notebook to introduce.

2. Second, as written, our controller requires full state feedback (K multiplies full state vectors z), which we do not have access to because our system, as written above, only returns x and y (because of C matrix). Hence, we would need a state observer, such as a Kalman Filter, to track the state variables. Instead, we assume that we have access to the full state.

The following code implements the closed loop system:

[6]:
```
# Our input to the system will only be (x_d, y_d), so we need to
# multiply it by this matrix to turn it into z_d.
Xd = matrix([[1,0,0,0,0,0],
             [0,1,0,0,0,0]]).T

# Closed loop dynamics
H = ss(A-B*K,B*K*Xd,C,D)

# Step response for the first input
x,t = step(H,input=0,output=0,T=linspace(0,10,100))
# Step response for the second input
y,t = step(H,input=1,output=1,T=linspace(0,10,100))
```

[7]:
```
plot(t,x,'-',t,y,'--')
plot([0, 10], [1, 1], 'k-')
ylabel('Position')
xlabel('Time (s)')
title('Step Response for Inputs')
legend(('Yx', 'Yy'), loc='lower right')
show()
```



The plot above shows the x and y positions of the aircraft when it is commanded to move 1 m in each direction. The following shows the x motion for control weights $\rho = 1, 10^2, 10^4$. A higher weight of the input term in the cost function causes a more sluggish response. It is created using the code:

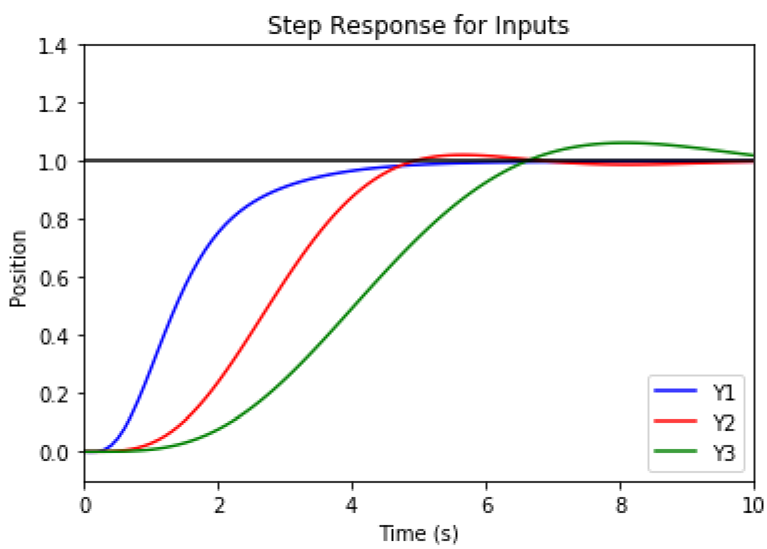
[8]:
```
# Look at different input weightings
Qu1a = diag([1, 1])
K1a, X, E = lqr(A, B, Qx1, Qu1a)
```

(continues on next page)

(continued from previous page)

```
H1ax = H = ss(A-B*K1a,B*K1a*Xd,C,D)

Qu1b = (40**2)*diag([1, 1])
K1b, X, E = lqr(A, B, Qx1, Qu1b)
H1bx = H = ss(A-B*K1b,B*K1b*Xd,C,D)

Qu1c = (200**2)*diag([1, 1])
K1c, X, E = lqr(A, B, Qx1, Qu1c)
H1cx = ss(A-B*K1c,B*K1c*Xd,C,D)

[Y1, T1] = step(H1ax, T=linspace(0,10,100), input=0,output=0)
[Y2, T2] = step(H1bx, T=linspace(0,10,100), input=0,output=0)
[Y3, T3] = step(H1cx, T=linspace(0,10,100), input=0,output=0)
```

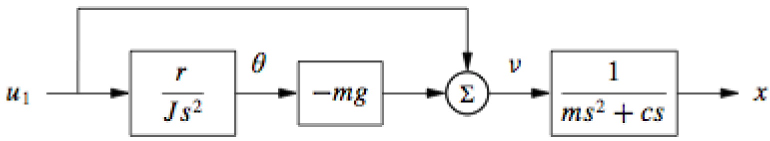
[9]:
```
plot(T1, Y1.T, 'b-', T2, Y2.T, 'r-', T3, Y3.T, 'g-')
plot([0 ,10], [1, 1], 'k-')
title('Step Response for Inputs')
ylabel('Position')
xlabel('Time (s)')
legend(('Y1','Y2','Y3'),loc='lower right')
axis([0, 10, -0.1, 1.4])
show()
```



Lateral control using inner/outer loop design

This section demonstrates the design of loop shaping controller for the vectored thrust aircraft example. This example is pulled from Chapter 11 (Frequency Domain Design) of Astrom and Murray.

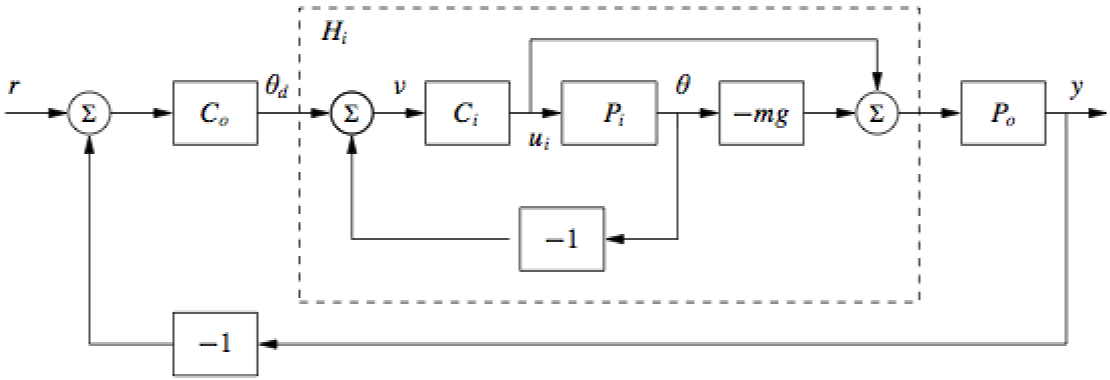
To design a controller for the lateral dynamics of the vectored thrust aircraft, we make use of a “inner/outer” loop design methodology. We begin by representing the dynamics using the block diagram



where

$$H_{\theta u_1} = \frac{r}{Js^2}, \qquad H_{xu_1} = \frac{Js^2 - mgr}{Js^2(ms^2+cs)}.$$

The controller is constructed by splitting the process dynamics and controller into two components: an inner loop consisting of the roll dynamics P_i and control C_i and an outer loop consisting of the lateral position dynamics P_o and controller C_o.



The closed inner loop dynamics H_i control the roll angle of the aircraft using the vectored thrust while the outer loop controller C_o commands the roll angle to regulate the lateral position.

The following code imports the libraries that are required and defines the dynamics:

[10]:
```
from matplotlib.pyplot import * # Grab MATLAB plotting functions
from control.matlab import *    # MATLAB-like functions
```

[12]:
```
# System parameters
m = 4                          # mass of aircraft
J = 0.0475                     # inertia around pitch axis
r = 0.25                       # distance to center of force
g = 9.8                        # gravitational constant
c = 0.05                       # damping factor (estimated)
```

[13]:
```
# Transfer functions for dynamics
Pi = tf([r], [J, 0, 0])        # inner loop (roll)
Po = tf([1], [m, c, 0])        # outer loop (position)
```

For the inner loop, use a lead compensator

[14]:
```
k = 200
a = 2
b = 50
Ci = k*tf([1, a], [1, b])      # lead compensator
Li = Pi*Ci
```

The closed loop dynamics of the inner loop, H_i, are given by

[15]:
```
Hi = parallel(feedback(Ci, Pi), -m*g*feedback(Ci*Pi, 1))
```

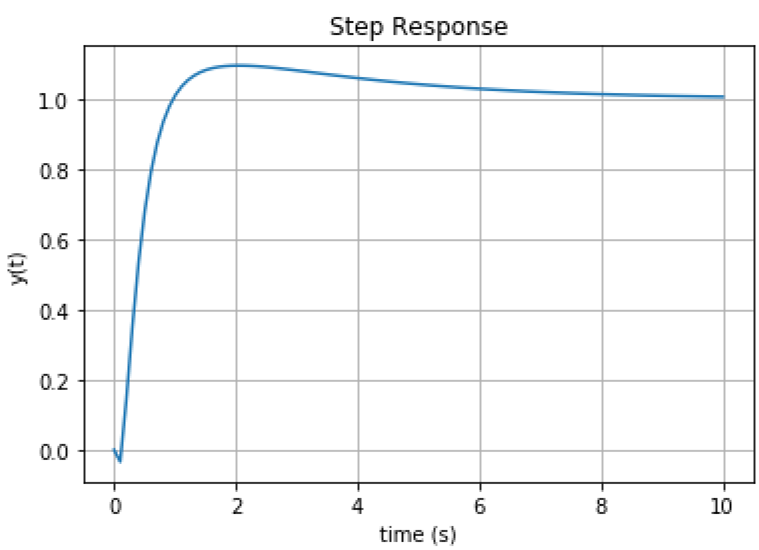
Finally, we design the lateral compensator using another lead compenstor

[16]:
```
# Now design the lateral control system
a = 0.02
b = 5
K = 2
Co = -K*tf([1, 0.3], [1, 10])          # another lead compensator
Lo = -m*g*Po*Co
```

The performance of the system can be characterized using the sensitivity function and the complementary sensitivity function:

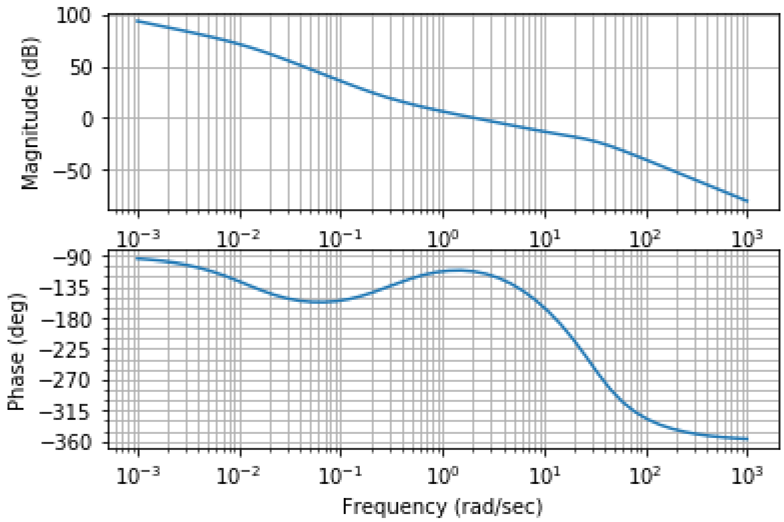
[17]:
```
L = Co*Hi*Po
S = feedback(1, L)
T = feedback(L, 1)
```

[18]:
```
t, y = step(T, T=linspace(0,10,100))
plot(y, t)
title("Step Response")
grid()
xlabel("time (s)")
ylabel("y(t)")
show()
```

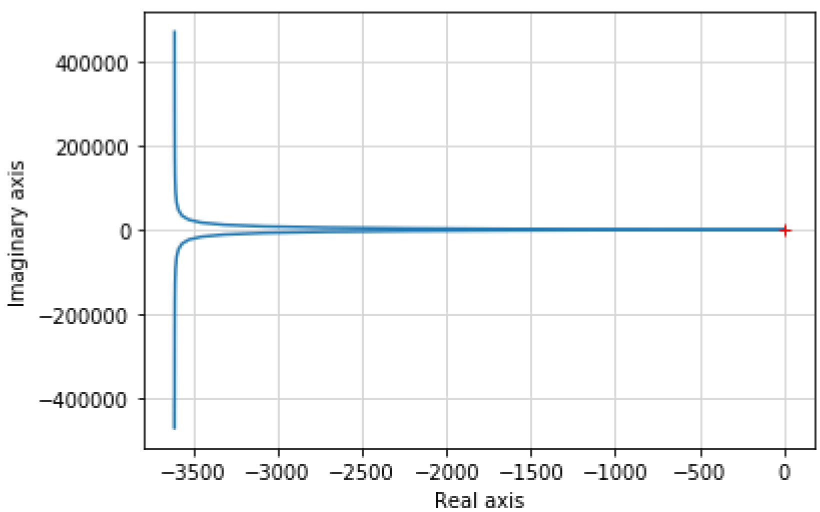


The frequency response and Nyquist plot for the loop transfer function are computed using the commands

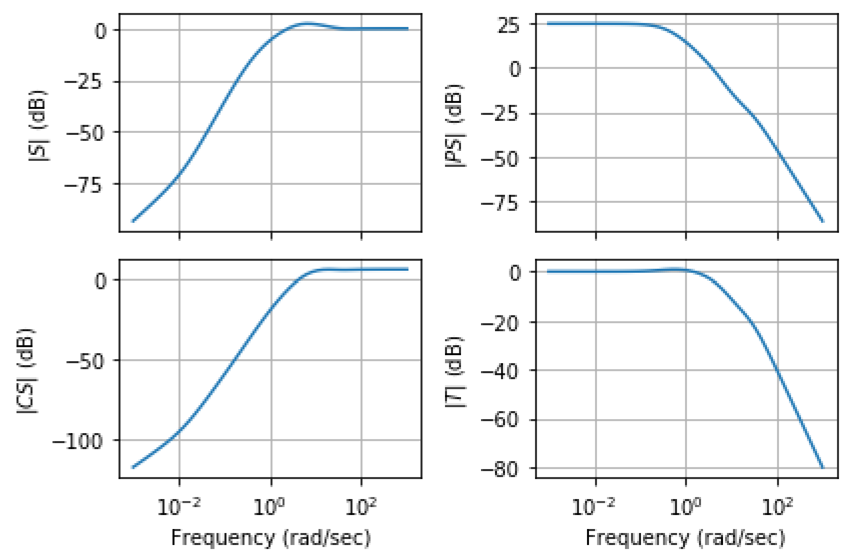
[19]:
```
bode(L)
show()
```



[20]:
```
nyquist(L, (0.0001, 1000))
show()
```



[21]:
```
gangof4(Hi*Po, Co)
```



- genindex

Development

You can check out the latest version of the source code with the command:

```
git clone https://github.com/python-control/python-control.git
```

You can run the unit tests with pytest to make sure that everything is working correctly. Inside the source directory, run:

```
pytest -v
```

or to test the installed package:

```
pytest --pyargs control -v
```

Your contributions are welcome! Simply fork the GitHub repository and send a pull request.

Links

- Issue tracker: https://github.com/python-control/python-control/issues
- Mailing list: http://sourceforge.net/p/python-control/mailman/

PYTHON MODULE INDEX

c

INDEX

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

Z