# CONTENTS

1 Introduction 3
   1.1 Overview of the toolbox 3
   1.2 Some differences from MATLAB 3
   1.3 Installation 3
   1.4 Getting started 4

2 Library conventions 5
   2.1 LTI system representation 5
   2.2 Time series data 7
   2.3 Package configuration parameters 8

3 Function reference 11
   3.1 System creation 11
   3.2 System interconnections 16
   3.3 Frequency domain plotting 20
   3.4 Time domain simulation 26
   3.5 Control system analysis 35
   3.6 Matrix computations 43
   3.7 Control system synthesis 49
   3.8 Model simplification tools 57
   3.9 Nonlinear system support 62
   3.10 Stochastic system support 70
   3.11 Utility functions and conversions 75

4 Control system classes 87
   4.1 control.StateSpace 87
   4.2 control.TransferFunction 97
   4.3 control.InputOutputSystem 106
   4.4 control.FrequencyResponseData 111
   4.5 control.TimeResponseData 117
   4.6 Input/output system subclasses 121
   4.7 Additional classes 122

5 MATLAB compatibility module 123
   5.1 Creating linear models 123
   5.2 Utility functions and conversions 128
   5.3 System interconnections 132
   5.4 System gain and dynamics 137
   5.5 Time-domain analysis 139
   5.6 Frequency-domain analysis 142
## Introduction

The document is organized into several sections, each detailing specific aspects of control systems and related mathematical tools. Here's a breakdown of the contents:

### 5. Compensator design
- Compensator design (5.7) ........................................... 147
- State-space (SS) models (5.8) ..................................... 151
- Model simplification (5.9) ........................................... 153
- Time delays (5.10) .................................................... 157
- Matrix equation solvers and linear algebra (5.11) ............ 158
- Additional functions (5.12) ......................................... 161
- Functions imported from other modules (5.13) ............... 163

### 6. Differentially flat systems
- Overview of differential flatness (6.1) ........................... 165
- Module usage (6.2) ................................................... 166
- Example (6.3) ......................................................... 167
- Module classes and functions (6.4) .............................. 169

### 7. Input/output systems
- Module usage (7.1) .................................................... 189
- Example (7.2) ......................................................... 190
- Additional features (7.3) ............................................ 192
- Module classes and functions (7.4) .............................. 193

### 8. Describing functions
- Module usage (8.1) .................................................... 225
- Pre-defined nonlinearities (8.2) ................................... 226
- Module classes and functions (8.3) .............................. 226

### 9. Optimal control
- Problem setup (9.1) ................................................... 229
- Module usage (9.2) .................................................... 230
- Example (9.3) ......................................................... 231
- Optimization Tips (9.4) ............................................. 234
- Module classes and functions (9.5) .............................. 235

### 10. Examples
- Python scripts (10.1) ............................................... 247
- Jupyter notebooks (10.2) ........................................... 290

### Additional Resources
- Python Module Index ............................................... 363
- Index ........................................................................ 365
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
- Nonlinear input/output system modeling, simulation, and analysis
- Block diagram algebra: serial, parallel, and feedback interconnections
- Time response: initial, step, impulse
- Frequency response: Bode and Nyquist plots
- Control analysis: stability, reachability, observability, stability margins
- Control design: eigenvalue placement, LQR, H2, Hinf
- Model reduction: balanced realizations, Hankel singular values
- Estimator design: linear quadratic estimator (Kalman filter)

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

1.1 Overview of the toolbox

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

1.2 Some differences from MATLAB

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

In terms of the python-control package more specifically, here are some 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. Users can check to insure that slycot is installed correctly by running the command:
and verifying that no error message appears. 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:

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

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:

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

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

```shell
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:

```python
>>> import control
```

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

```python
>>> 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 :fun:`ss` function:

```python
sys = ct.ss(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 :fun:`tf` function:

```python
sys = ct.tf(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.

The FRD class is also used as the return type for the `frequency_response()` function (and the equivalent method for the `StateSpace` and `TransferFunction` classes). This object can be assigned to a tuple using

```python
mag, phase, omega = response
```

where `mag` is the magnitude (absolute value, not dB or log10) of the system frequency response, `phase` is the wrapped phase in radians of the system frequency response, and `omega` is the (sorted) frequencies at which the response was evaluated. If the system is SISO and the `squeeze` argument to `frequency_response()` is not True, `magnitude` and `phase` are 1D, indexed by frequency. If the system is not SISO or `squeeze` is False, the array is 3D, indexed by the output, input, and frequency. If `squeeze` is True then single-dimensional axes are removed. The processing of the `squeeze` keyword can be changed by calling the response function with a new argument:

```python
mag, phase, omega = response(squeeze=False)
```

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

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 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. Similarly, 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. 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 value of `control.config.defaults['control.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 \texttt{ss2tf()} and \texttt{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 \texttt{forced_response()}, \texttt{step_response()}, \texttt{impulse_response()}, and \texttt{initial_response()}.

**Note:** The convention used by python-control is different from the convention used in the \texttt{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 \texttt{scipy.signal}.

The time vector is a 1D array with shape \((n,)\):

\[
T = [t1, t2, t3, \ldots, tn]
\]

Input, state, and output all follow the same convention. Columns are different points in time, rows are different components:

\[
U = \begin{bmatrix}
u1(t1), u1(t2), u1(t3), \ldots, u1(tn) \\
u2(t1), u2(t2), u2(t3), \ldots, u2(tn) \\
\vdots \\
u(t1), u(t2), u(t3), \ldots, u(tn)
\end{bmatrix}
\]

Same for \(X, Y\)

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

When there is only one row, a 1D object is accepted or returned, which adds convenience for SISO systems:

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

\[
X0 = \begin{bmatrix}x1 \\
x2 \\
\vdots \\
xj
\end{bmatrix}
\]

Functions that return time responses (e.g., \texttt{forced_response()}, \texttt{impulse_response()}, \texttt{input_output_response()}, \texttt{initial_response()}, and \texttt{step_response()}) return a \texttt{TimeResponseData} object that contains the data for the time response. These data can be accessed via the \texttt{time}, \texttt{outputs}, \texttt{states} and \texttt{inputs} properties:

```python
sys = ct.rss(4, 1, 1)
response = ct.step_response(sys)
plot(response.time, response.outputs)
```
The dimensions of the response properties depend on the function being called and whether the system is SISO or MIMO. In addition, some time response function can return multiple “traces” (input/output pairs), such as the `step_response()` function applied to a MIMO system, which will compute the step response for each input/output pair. See `TimeResponseData` for more details.

The time response functions can also be assigned to a tuple, which extracts the time and output (and optionally the state, if the `return_x` keyword is used). This allows simple commands for plotting:

```python
t, y = ct.step_response(sys)
plot(t, y)
```

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

```python
t, y = ct.forced_response(sys, t, u)
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 (2D array) of a linear system, and $U$ is its input (array), then the feedthrough part of the system’s response, can be computed like this:

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

Finally, the `to_pandas()` function can be used to create a pandas dataframe:

```python
df = response.to_pandas()
```

The column labels for the data frame are `time` and the labels for the input, output, and state signals ($u[i]$, $y[i]$, and $x[i]$) by default, but these can be changed using the `inputs`, `outputs`, and `states` keywords when constructing the system, as described in `ss()`, `tf()`, and other system creation function. Note that when exporting to pandas, “rows” in the data frame correspond to time and “cols” (DataSeries) correspond to signals.

### 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:

```python
c configurations = [module.parameter] = value
```

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

```python
c 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:

- `freqplot.dB` (False): Bode plot magnitude plotted in dB (otherwise powers of 10)
- `freqplot.deg` (True): Bode plot phase plotted in degrees (otherwise radians)
- `freqplot.Hz` (False): Bode plot frequency plotted in Hertz (otherwise rad/sec)
- `freqplot.grid` (True): Include grids for magnitude and phase plots
• freqplot.number_of_samples (1000): 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:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>reset_defaults()</code></td>
<td>Reset configuration values to their default (initial) values.</td>
</tr>
<tr>
<td><code>use_fbs_defaults()</code></td>
<td>Use Feedback Systems (FBS) compatible settings.</td>
</tr>
<tr>
<td><code>use_matlab_defaults()</code></td>
<td>Use MATLAB compatible configuration settings.</td>
</tr>
<tr>
<td><code>use_numpy_matrix(flag, warn)</code></td>
<td>Turn on/off use of Numpy matrix class for state space operations.</td>
</tr>
<tr>
<td><code>use_legacy_defaults(version)</code></td>
<td>Sets the defaults to whatever they were in a given release.</td>
</tr>
</tbody>
</table>

### 2.3.1 control.reset_defaults

```python
control.reset_defaults()
```
Reset configuration values to their default (initial) values.

### 2.3.2 control.use_fbs_defaults

```python
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
- Nyquist plots use dashed lines for mirror image of Nyquist curve

### 2.3.3 control.use_matlab_defaults

```python
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 deprecated Numpy `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’.
FUNCTION REFERENCE

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

3.1 System creation

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ss(A, B, C, D[, dt])</td>
<td>Create a state space system. The function accepts either 1, 2, 4 or 5 parameters:</td>
</tr>
<tr>
<td>tf(num, den[, dt])</td>
<td>Create a transfer function system.</td>
</tr>
<tr>
<td>frd(d, w)</td>
<td>Construct a frequency response data model</td>
</tr>
<tr>
<td>rss([states, outputs, inputs, strictly_proper])</td>
<td>Create a stable random state space object.</td>
</tr>
<tr>
<td>drss(*args, **kwargs)</td>
<td>Create a stable, discrete-time, random state space system</td>
</tr>
</tbody>
</table>

3.1.1 control.ss

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

Create a state space system.

The function accepts either 1, 2, 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 state space system.

- **ss(updfcn, outfucn)**
  Create a nonlinear input/output system with update function updfcn and output function outfucn. See NonlinearIOSystem for more information.

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

- **ss(A, B, C, D, dt)**
  Create a discrete-time state space system from the matrices of its state and output equations:
  \[
  x[k+1] = Ax[k] + Bu[k] \\
  y[k] = Cx[k] + Du[k]
  \]

The matrices can be given as array like data types or strings. Everything that the constructor of numpy. matrix accepts is permissible here too.
Create a system with named input, output, and state signals.

Parameters

- **sys** *(StateSpace or TransferFunction)* – A linear system.
- **A** *(array_like or string)* – System, control, output, and feed forward matrices.
- **B** *(array_like or string)* – System, control, output, and feed forward matrices.
- **C** *(array_like or string)* – System, control, output, and feed forward matrices.
- **D** *(array_like or string)* – System, control, output, and feed forward matrices.
- **dt** *(None, True or float, optional)* – System timebase. 0 (default) indicates continuous time, True indicates discrete time with unspecified sampling time, positive number is discrete time with specified sampling time, None indicates unspecified timebase (either continuous or discrete time).
- **inputs** *(str, or list of str, optional)* – List of strings that name the individual signals. If this parameter is not given or given as `None`, the signal names will be of the form `s[i]` (where `s` is one of `u`, `y`, or `x`). See `InputOutputSystem` for more information.
- **outputs** *(str, or list of str, optional)* – List of strings that name the individual signals. If this parameter is not given or given as `None`, the signal names will be of the form `s[i]` (where `s` is one of `u`, `y`, or `x`). See `InputOutputSystem` for more information.
- **states** *(str, or list of str, optional)* – List of strings that name the individual signals. If this parameter is not given or given as `None`, the signal names will be of the form `s[i]` (where `s` is one of `u`, `y`, or `x`). See `InputOutputSystem` for more information.
- **name** *(string, optional)* – System name (used for specifying signals). If unspecified, a generic name `<sys[id]>` is generated with a unique integer id.

Returns

- **out** – Linear input/output system.

Return type

- **LinearIOSystem**

Raises

- **ValueError** – If matrix sizes are not self-consistent.

See also:

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

Examples

```python
>>> # Create a Linear I/O system object from from for 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
- inputs (str, or list of str, optional) – List of strings that name the individual signals of the transformed system. If not given, the inputs and outputs are the same as the original system.
- outputs (str, or list of str, optional) – List of strings that name the individual signals of the transformed system. If not given, the inputs and outputs are the same as the original system.
- name (string, optional) – System name. If unspecified, a generic name <sys[id]> is generated with a unique integer id.

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

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

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

```python
>>> # 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:

```python
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]
```

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

Parameters

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

Returns

- **sys** – New frequency response system

Return type

- **FRD**
3.1.4 controlrss

controlrss\(\)\(\text{states}=1, \text{outputs}=1, \text{inputs}=1, \text{strictly_proper}=False, **\text{kwargs}**\)

Create a stable random state space object.

**Parameters**

- **inputs**\(\)\(\text{int}, \text{list of str}, \text{or None}\) – 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 \(u, y, \text{or } x\)).

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

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

- **strictly_proper**\(\)\(\text{bool, optional}\) – If set to ‘True’, returns a proper system (no direct term).

- **dt**\(\)\(\text{None, True or float, optional}\) – System timebase. 0 (default) indicates continuous time, True indicates discrete time with unspecified sampling time, positive number is discrete time with specified sampling time, None indicates unspecified timebase (either continuous or discrete time).

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

**Returns**

- **sys** – The randomly created linear system

**Return type**

*StateSpace*

**Raises**

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

**Notes**

If the number of states, inputs, or outputs is not specified, then the missing numbers are assumed to be 1. If dt is not specified or is given as 0 or None, the poles of the returned system will always have a negative real part. If dt is True or a positive float, the poles of the returned system will have magnitude less than 1.
3.1.5 control.drss

control.drss(*args, **kwargs)

Create a stable, discrete-time, random state space system

Create a stable discrete time random state space object. This function calls rss() using either the dt keyword provided by the user or dt=True if not specified.

3.2 System interconnections

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

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

```python
>>> sys1 = ss([[1., -2], [3., -4]], [[5.], [7]], [[6., 8]], [[9.]])
>>> sys2 = ss([[-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 or 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

```python
>>> 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])
```

Notes

The interconnect() function in the input/output systems module allows the use of named signals and provides an alternative method for interconnecting multiple systems.

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. \( \text{sign} = -1 \) indicates negative feedback, and \( \text{sign} = 1 \) indicates positive feedback. \( \text{sign} \) is an optional argument; it assumes a value of -1 if not specified.

**Returns**

out

**Return type**

*StateSpace or TransferFunction*

**Raises**

- \( \text{ValueError} \) – if \( \text{sys1} \) does not have as many inputs as \( \text{sys2} \) has outputs, or if \( \text{sys2} \) does not have as many inputs as \( \text{sys1} \) has outputs

- \( \text{NotImplementedError} \) – 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 \( \text{sys1} \) is a TransferFunction object, and StateSpace.feedback if \( \text{sys1} \) is a StateSpace object. If \( \text{sys1} \) is a scalar, then it is converted to \( \text{sys2} \)’s type, and the corresponding feedback function is used. If \( \text{sys1} \) and \( \text{sys2} \) 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

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

### 3.2.5 control.parallel

control.parallel(sys1, sys2[, ..., sysn])

Return the parallel connection \( sys1 + sys2 (\ldots + sysn) \).

**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**

```python
>>> 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, sys2[, ..., sysn])

Return the series connection \( (sysn \ldots \ast) sys2 \ast sys1 \).

**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.ninputs does not equal sys1.noutputs if sys1.dt is not compatible with sys2.dt

See also:
parallel, feedback

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

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

Examples

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bode_plot(syslist[, omega, plot, ...])</td>
<td>Bode plot for a system</td>
</tr>
<tr>
<td>describing_function_plot(H, F, A[, omega, ...])</td>
<td>Plot a Nyquist plot with a describing function for a nonlinear system.</td>
</tr>
<tr>
<td>nyquist_plot(syslist[, omega, plot, ...])</td>
<td>Nyquist plot for a system</td>
</tr>
<tr>
<td>gangof4_plot(P, C[, omega])</td>
<td>Plot the &quot;Gang of 4&quot; transfer functions for a system</td>
</tr>
<tr>
<td>nichols_plot(sys_list[, omega, grid])</td>
<td>Nichols plot for a system</td>
</tr>
<tr>
<td>nichols_grid([cl_mags, cl_phases, ...])</td>
<td>Nichols chart grid</td>
</tr>
</tbody>
</table>

3.3.1 control.bode_plot

control.bode_plot(syslist, omega=None, plot=True, omega_limits=None, omega_num=None, margins=None, method='best', *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** (*array_like*) – 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['freqplot.Hz']

• **deg** (*bool*) – If True, plot phase in degrees (else radians). Default value (True) config.defaults['freqplot.deg']

• **plot** (*bool*) – If True (default), plot magnitude and phase

• **omega_limits** (*array_like 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.

• **method** (method to use in computing margins (see stability_margins())) –

• **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)

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

• **initial_phase** (*float*) – Set the reference phase to use for the lowest frequency. If set, the initial phase of the Bode plot will be set to the value closest to the value specified. Units are in either degrees or radians, depending on the deg parameter. Default is -180 if wrap_phase is False, 0 if wrap_phase is True.

• **wrap_phase** (*bool or float*) – If wrap_phase is False (default), then the phase will be unwrapped so that it is continuously increasing or decreasing. If wrap_phase is True the phase will be restricted to the range [-180, 180) (or [-π, π) radians). If wrap_phase is specified as a float, the phase will be offset by 360 degrees if it falls below the specified value. Default value is False and can be set using config.defaults['freqplot.wrap_phase'].

• **reset** (The default values for Bode plot configuration parameters can be) –

• **dictionary** (using the config.defaults) –

• 'bode'.(with module name) –

**Returns**

• **mag** (*ndarray (or list of ndarray if len(syslist) > 1))*) – magnitude

• **phase** (*ndarray (or list of ndarray if len(syslist) > 1))*) – phase in radians

• **omega** (*ndarray (or list of ndarray if len(syslist) > 1))*) – frequency in rad/sec
Notes

1. Alternatively, you may use the lower-level methods LTI.frequency_response() or sys(s) or sys(z) or to generate the frequency response for a single system.

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

Examples

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

3.3.2 control.describing_function_plot

control.describing_function_plot(H, F, A, omega=None, refine=True, label='%5.2g @ %-5.2g', warn=None, **kwargs)

Plot a Nyquist plot with a describing function for a nonlinear system.

This function generates a Nyquist plot for a closed loop system consisting of a linear system with a static nonlinear function in the feedback path.

Parameters

- **H** (*LTI system*) – Linear time-invariant (LTI) system (state space, transfer function, or FRD)
- **F** (*static nonlinear function*) – A static nonlinearity, either a scalar function or a single-input, single-output, static input/output system.
- **A** (*list*) – List of amplitudes to be used for the describing function plot.
- **omega** (*list, optional*) – List of frequencies to be used for the linear system Nyquist curve.
- **label** (*str, optional*) – Formatting string used to label intersection points on the Nyquist plot. Defaults to “%5.2g @ %-5.2g”. Set to None to omit labels.
- **warn** (*bool, optional*) – Set to True to turn on warnings generated by nyquist_plot or False to turn off warnings. If not set (or set to None), warnings are turned off if omega is specified, otherwise they are turned on.

Returns

- **intersections** – A list of all amplitudes and frequencies in which \( H(j\omega)N(a) = -1 \), where \( N(a) \) is the describing function associated with \( F \), or None if there are no such points. Each pair represents a potential limit cycle for the closed loop system with amplitude given by the first value of the tuple and frequency given by the second value.

Return type

1D array of 2-tuples or None
Example

```python
>>> H_simple = ct.tf([8], [1, 2, 2, 1])
>>> F_saturation = ct.descfcn.saturation_nonlinearity(1)
>>> amp = np.linspace(1, 4, 10)
>>> ct.describing_function_plot(H_simple, F_saturation, amp)
[(3.344008947853124, 1.414213099755523)]
```

### 3.3.3 control.nyquist_plot

`control.nyquist_plot(syslist, omega=None, plot=True, omega_limits=None, omega_num=None, label_freq=0, color=None, return_contour=False, warn_encirclements=True, warn_nyquist=True, **kwargs)`

Nyquist plot for a system

Plots a Nyquist plot for the system over a (optional) frequency range. The curve is computed by evaluating the Nyquist segment along the positive imaginary axis, with a mirror image generated to reflect the negative imaginary axis. Poles on or near the imaginary axis are avoided using a small indentation. The portion of the Nyquist contour at infinity is not explicitly computed (since it maps to a constant value for any system with a proper transfer function).

**Parameters**

- `syslist` *(list of LTI)* – List of linear input/output systems (single system is OK). Nyquist curves for each system are plotted on the same graph.
- `plot` *(boolean)* – If True, plot magnitude
- `omega` *(array_like)* – Set of frequencies to be evaluated, in rad/sec.
- `omega_limits` *(array_like of two values)* – Limits to the range of frequencies. Ignored if `omega` is provided, and auto-generated if omitted.
- `omega_num` *(int)* – Number of frequency samples to plot. Defaults to config.defaults['freqplot.number_of_samples'].
- `color` *(string)* – Used to specify the color of the line and arrowhead.
- `return_contour` *(bool, optional)* – If ‘True’, return the contour used to evaluate the Nyquist plot.
- `**kwargs` *(matplotlib.pyplot.plot() keyword properties, optional)* – Additional keywords (passed to matplotlib)

**Returns**

- `count` *(int (or list of int if len(syslist) > 1))* – Number of encirclements of the point -1 by the Nyquist curve. If multiple systems are given, an array of counts is returned.
- `contour` *(ndarray (or list of ndarray if len(syslist) > 1), optional)* – The contour used to create the primary Nyquist curve segment, returned if `return_contour` is True. To obtain the Nyquist curve values, evaluate system(s) along contour.

**Additional Parameters**

- `arrows` *(int or 1D/2D array of floats, optional)* – Specify the number of arrows to plot on the Nyquist curve. If an integer is passed, that number of equally spaced arrows will be plotted on each of the primary segment and the mirror image. If a 1D array is passed, it should
consist of a sorted list of floats between 0 and 1, indicating the location along the curve to plot an arrow. If a 2D array is passed, the first row will be used to specify arrow locations for the primary curve and the second row will be used for the mirror image.

- **arrow_size** *(float, optional)* – Arrowhead width and length (in display coordinates). Default value is 8 and can be set using config.defaults['nyquist.arrow_size'].

- **arrow_style** *(matplotlib.patches.ArrowStyle, optional)* – Define style used for Nyquist curve arrows (overrides arrow_size).

- **encirclement_threshold** *(float, optional)* – Define the threshold for generating a warning if the number of net encirclements is a non-integer value. Default value is 0.05 and can be set using config.defaults['nyquist.encirclement_threshold'].

- **indent_direction** *(str, optional)* – For poles on the imaginary axis, set the direction of indentation to be ‘right’ (default), ‘left’, or ‘none’.

- **indent_points** *(int, optional)* – Number of points to insert in the Nyquist contour around poles that are at or near the imaginary axis.

- **indent_radius** *(float, optional)* – Amount to indent the Nyquist contour around poles on or near the imaginary axis. Portions of the Nyquist plot corresponding to indented portions of the contour are plotted using a different line style.

- **label_freq** *(int, optional)* – Label every nth frequency on the plot. If not specified, no labels are generated.

- **max_curve_magnitude** *(float, optional)* – Restrict the maximum magnitude of the Nyquist plot to this value. Portions of the Nyquist plot whose magnitude is restricted are plotted using a different line style.

- **max_curve_offset** *(float, optional)* – When plotting scaled portion of the Nyquist plot, increase/decrease the magnitude by this fraction of the max_curve_magnitude to allow any overlaps between the primary and mirror curves to be avoided.

- **mirror_style** *(str, str or False)* – Linestyles for mirror image of the Nyquist curve. The first element is used for unscaled portions of the Nyquist curve, the second element is used for portions that are scaled (using max_curve_magnitude). If False then omit completely. Default linestyle (['−', ‘:’]) is determined by config.defaults['nyquist.mirror_style'].

- **primary_style** *(str, str, optional)* – Linestyles for primary image of the Nyquist curve. The first element is used for unscaled portions of the Nyquist curve, the second element is used for portions that are scaled (using max_curve_magnitude). Default linestyle (['−', ‘−.’]) is determined by config.defaults['nyquist.mirror_style'].

- **start_marker** *(str, optional)* – Matplotlib marker to use to mark the starting point of the Nyquist plot. Defaults value is ‘o’ and can be set using config.defaults['nyquist.start_marker'].

- **start_marker_size** *(float, optional)* – Start marker size (in display coordinates). Default value is 4 and can be set using config.defaults['nyquist.start_marker_size'].

- **warn_nyquist** *(bool, optional)* – If set to ‘False’, turn off warnings about frequencies above Nyquist.

- **warn_encirclements** *(bool, optional)* – If set to ‘False’, turn off warnings about number of encirclements not meeting the Nyquist criterion.
Notes

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

2. If a continuous-time system contains poles on or near the imaginary axis, a small indentation will be used to avoid the pole. The radius of the indentation is given by \text{indent_radius} and it is taken to the right of stable poles and the left of unstable poles. If a pole is exactly on the imaginary axis, the \text{indent_direction} parameter can be used to set the direction of indentation. Setting \text{indent_direction} to \text{none} will turn off indentation. If \text{return_contour} is True, the exact contour used for evaluation is returned.

Examples

```python
>>> sys = ss([[1, -2], [3, -4]], [[5], [7]], [[6, 8]], [[9]])
>>> count = nyquist_plot(sys)
```

3.3.4 control.gangof4_plot

control.gangof4_plot($P$, $C$, \text{omega}=None, **\text{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

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

Return type

None

3.3.5 control.nichols_plot

control.nichols_plot($\text{sys_list}$, \text{omega}=None, \text{grid}=None)

Nichols plot for a system

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

Parameters

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

Return type

None
3.3.6 control.nichols_grid

control.nichols_grid(cl_mags=None, cl_phases=None, line_style='dotted', ax=None, label_cl_phases=True)

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 \(-360 < \text{cl_phases} < 0\)
- **line_style** (string, optional) – Matplotlib linestyle
- **ax** (matplotlib.axes.Axes, optional) – Axes to add grid to. If None, use plt.gca().
- **label_cl_phases** (bool, optional) – If True, closed-loop phase lines will be labelled.

Returns

- **cl_mag_lines** (list of matplotlib.line.Line2D) – The constant closed-loop gain contours
- **cl_phase_lines** (list of matplotlib.line.Line2D) – The constant closed-loop phase contours
- **cl_mag_labels** (list of matplotlib.text.Text) – Mcontour labels; each entry corresponds to the respective entry in cl_mag_lines
- **cl_phase_labels** (list of matplotlib.text.Text) – Ncontour labels; each entry corresponds to the respective entry in cl_phase_lines

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

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

3.4 Time domain simulation

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>forced_response(sys[, T, U, X0, transpose, ...])</td>
<td>Simulate the output of a linear system.</td>
</tr>
<tr>
<td>impulse_response(sys[, T, X0, input, ...])</td>
<td>Compute the impulse response for a linear system.</td>
</tr>
<tr>
<td>initial_response(sys[, T, X0, input, ...])</td>
<td>Initial condition response of a linear system.</td>
</tr>
<tr>
<td>input_output_response(sys, T[, U, X0, ...])</td>
<td>Compute the output response of a system to a given input.</td>
</tr>
<tr>
<td>step_response(sys[, T, X0, input, output, ...])</td>
<td>Compute the step response for a linear system.</td>
</tr>
<tr>
<td>phase_plot(odefun[, X, Y, scale, X0, T, ...])</td>
<td>Phase plot for 2D dynamical systems</td>
</tr>
</tbody>
</table>
3.4.1 control.forced_response

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

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` (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.
  - If None, $U$ must be given and `len(U)` time steps of `sys.dt` are simulated. If `sys.dt` is None or True (undetermined time step), a time step of 1.0 is assumed.
- `U` (array_like or float, optional) – Input array giving input at each time $T$. If $U$ is None or 0, $T$ must be given, even for discrete time systems. In this case, for continuous time systems, a direct calculation of the matrix exponential is used, which is faster than the general interpolating algorithm used otherwise.
- `X0` (array_like or float, default=0.) – Initial condition.
- `transpose` (bool, default=False) – If True, transpose all input and output arrays (for backward compatibility with MATLAB and `scipy.signal.lsim()`).
- `interpolate` (bool, 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.
- `return_x` (bool, default=None) – Used if the time response data is assigned to a tuple:
  - If False, return only the time and output vectors.
  - If True, also return the the state vector.
  - If None, determine the returned variables by `config.defaults[‘forced_response.return_x’]`, which was True before version 0.9 and is False since then.
- `squeeze` (bool, optional) – By default, if a system is single-input, single-output (SISO) then the output response is returned as a 1D array (indexed by time). If `squeeze` is True, remove single-dimensional entries from the shape of the output even if the system is not SISO. If `squeeze` is False, keep the output as a 2D array (indexed by the output number and time) even if the system is SISO. The default behavior can be overridden by `config.defaults[‘control.squeeze_time_response’]`.

Returns

- `results` – Time response represented as a `TimeResponseData` object containing the following properties:
  - `time` (array): Time values of the output.
  - `outputs` (array): Response of the system. If the system is SISO and `squeeze` is not True, the array is 1D (indexed by time). If the system is not SISO or `squeeze` is False, the array is 2D (indexed by output and time).
• states (array): Time evolution of the state vector, represented as a 2D array indexed by state and time.
• inputs (array): Input(s) to the system, indexed by input and time.

The return value of the system can also be accessed by assigning the function to a tuple of length 2 (time, output) or of length 3 (time, output, state) if return_x is True.

Return type

TimeResponseData

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

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

See Time series data and Package configuration parameters.

3.4.2 control.impulse_response

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

Compute the impulse response for a linear system.

If the system has multiple inputs and/or multiple outputs, the impulse response is computed for each input/output pair, with all other inputs set to zero. Optionally, a single input and/or single output can be selected, in which case all other inputs are set to 0 and 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, optional) – Only compute the impulse response for the listed input. If not specified, the impulse responses for each independent input are computed.
- `output` (int, optional) – Only report the step response for the listed output. If not specified, all outputs are reported.
- `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, optional) – If True, transpose all input and output arrays (for backward compatibility with MATLAB and scipy.signal.lsim()). Default value is False.

• **return_x** (bool, optional) – If True, return the state vector when assigning to a tuple (default = False). See `forced_response()` for more details.

• **squeeze** (bool, optional) – By default, if a system is single-input, single-output (SISO) then the output response is returned as a 1D array (indexed by time). If squeeze=True, remove single-dimensional entries from the shape of the output even if the system is not SISO. If squeeze=False, keep the output as a 2D array (indexed by the output number and time) even if the system is SISO. The default value can be set using config.defaults['control.squeeze_time_response'].

**Returns**

results – Impulse response represented as a `TimeResponseData` object containing the following properties:

• time (array): Time values of the output.

• outputs (array): Response of the system. If the system is SISO and squeeze is not True, the array is 1D (indexed by time). If the system is not SISO or squeeze is False, the array is 3D (indexed by the output, trace, and time).

• states (array): Time evolution of the state vector, represented as either a 2D array indexed by state and time (if SISO) or a 3D array indexed by state, trace, and time. Not affected by squeeze.

The return value of the system can also be accessed by assigning the function to a tuple of length 2 (time, output) or of length 3 (time, output, state) if return_x is True.

**Return type**

`TimeResponseData`

**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**

```python
>>> 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=None)

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, optional) – If True, transpose all input and output arrays (for backward compatibility with MATLAB and scipy.signal.lsim()). Default value is False.
- `return_x` (bool, optional) – If True, return the state vector when assigning to a tuple (default = False). See forced_response() for more details.
- `squeeze` (bool, optional) – By default, if a system is single-input, single-output (SISO) then the output response is returned as a 1D array (indexed by time). If squeeze=True, remove single-dimensional entries from the shape of the output even if the system is not SISO. If squeeze=False, keep the output as a 2D array (indexed by the output number and time) even if the system is SISO. The default value can be set using config.defaults[‘control.squeeze_time_response’].

Returns

results – Time response represented as a TimeResponseData object containing the following properties:

- `time` (array): Time values of the output.
- `outputs` (array): Response of the system. If the system is SISO and squeeze is not True, the array is 1D (indexed by time). If the system is not SISO or squeeze is False, the array is 2D (indexed by the output and time).
- `states` (array): Time evolution of the state vector, represented as either a 2D array indexed by state and time (if SISO). Not affected by squeeze.

The return value of the system can also be accessed by assigning the function to a tuple of length 2 (time, output) or of length 3 (time, output, state) if return_x is True.

Return type

TimeResponseData
See also:

forced_response, impulse_response, step_response

Notes

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

Examples

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

3.4.4 control.input_output_response

```python
control.input_output_response(sys, T, U=0.0, X0=0, params={}, transpose=False, return_x=False, squeeze=None, solve_ivp_kwars={}, t_eval='T', **kwargs)
```

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, list, or number, optional) – Input array giving input at each time 
  \( T \) (default = 0). If a list is specified, each element in the list will be treated as a portion of 
  the input and broadcast (if necessary) to match the time vector.
- **X0** (array-like, list, or number, optional) – Initial condition (default = 0). If a 
  list is given, each element in the list will be flattened and stacked into the initial condition. 
  If a smaller number of elements are given that the number of states in the system, the initial 
  condition will be padded with zeros.
- **return_x** (bool, optional) – If True, return the state vector when assigning to a tuple 
  (default = False). See forced_response() for more details. If True, return the values of 
  the state at each time (default = False).
- **squeeze** (bool, optional) – If True and if the system has a single output, return 
  the system output as a 1D array rather than a 2D array. If False, return the 
  system output as a 2D array even if the system is SISO. Default value set by con-
  fig.defaults[‘control.squeeze_time_response’].
- **solve_ivp_method** (str, optional) – Set the method used by scipy.integrate.
  solve_ivp(). Defaults to ‘RK45’.
- **solve_ivp_kwargs** (str, optional) – Pass additional keywords to scipy.integrate.
  solve_ivp().

Returns

- **results** – Time response represented as a TimeResponseData object containing the following 
  properties:
  - **time** (array): Time values of the output.
• outputs (array): Response of the system. If the system is SISO and `squeeze` is not True, the array is 1D (indexed by time). If the system is not SISO or `squeeze` is False, the array is 2D (indexed by output and time).

• states (array): Time evolution of the state vector, represented as a 2D array indexed by state and time.

• inputs (array): Input(s) to the system, indexed by input and time.

The return value of the system can also be accessed by assigning the function to a tuple of length 2 (time, output) or of length 3 (time, output, state) if `return_x` is True. If the input/output system signals are named, these names will be used as labels for the time response.

Return type

`TimeResponseData`

Raises

• `TypeError` – If the system is not an input/output system.

• `ValueError` – If time step does not match sampling time (for discrete time systems).

Notes

1. If a smaller number of initial conditions are given than the number of states in the system, the initial conditions will be padded with zeros. This is often useful for interconnected control systems where the process dynamics are the first system and all other components start with zero initial condition since this can be specified as `[xsys_0, 0]`. A warning is issued if the initial conditions are padded and and the final listed initial state is not zero.

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=None)
```

Compute the step response for a linear system.

If the system has multiple inputs and/or multiple outputs, the step response is computed for each input/output pair, with all other inputs set to zero. Optionally, a single input and/or single output can be selected, in which case all other inputs are set to 0 and 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, optional) – Only compute the step response for the listed input. If not specified, the step responses for each independent input are computed (as separate traces).
• **output** *(int, optional)* – Only report the step response for the listed output. If not specified, all outputs are reported.

• **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, optional)* – If True, transpose all input and output arrays (for backward compatibility with MATLAB and *scipy.signal.lsim()*). Default value is False.

• **return_x** *(bool, optional)* – If True, return the state vector when assigning to a tuple (default = False). See *forced_response()* for more details.

• **squeeze** *(bool, optional)* – By default, if a system is single-input, single-output (SISO) then the output response is returned as a 1D array (indexed by time). If squeeze=True, remove single-dimensional entries from the shape of the output even if the system is not SISO. If squeeze=False, keep the output as a 3D array (indexed by the output, input, and time) even if the system is SISO. The default value can be set using config.defaults[‘control.squeeze_time_response’].

**Returns**

- **results** – Time response represented as a *TimeResponseData* object containing the following properties:
  - **time** *(array):* Time values of the output.
  - **outputs** *(array):* Response of the system. If the system is SISO and squeeze is not True, the array is 1D (indexed by time). If the system is not SISO or squeeze is False, the array is 3D (indexed by the output, trace, and time).
  - **states** *(array):* Time evolution of the state vector, represented as either a 2D array indexed by state and time (if SISO) or a 3D array indexed by state, trace, and time. Not affected by squeeze.
  - **inputs** *(array):* Input(s) to the system, indexed in the same manner as outputs.

The return value of the system can also be accessed by assigning the function to a tuple of length 2 (time, output) or of length 3 (time, output, state) if return_x is True.

**Return type**

*TimeResponseData*

**See also:**

*forced_response, initial_response, impulse_response*

**Notes**

This function uses the *forced_response* function with the input set to a unit step.
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, X0=[...], T=Tmax, lingrid=N, ...) - plot both phase_plot(func, X0=[...], T=Tmax, 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 dx/dt = 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 Control system analysis

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>dcgain(sys)</code></td>
<td>Return the zero-frequency (or DC) gain of the given system</td>
</tr>
<tr>
<td><code>describing_function(F, A[, num_points, ...])</code></td>
<td>Numerical compute the describing function of a non-linear function</td>
</tr>
<tr>
<td><code>evalfr(sys, x[, squeeze])</code></td>
<td>Evaluate the transfer function of an LTI system for complex frequency x.</td>
</tr>
<tr>
<td><code>freqresp(sys, omega[, squeeze])</code></td>
<td>Frequency response of an LTI system at multiple angular frequencies.</td>
</tr>
<tr>
<td><code>margin(sysdata)</code></td>
<td>Calculate gain and phase margins and associated crossover frequencies.</td>
</tr>
<tr>
<td><code>stability_margins(sysdata[, returnall, ...])</code></td>
<td>Calculate stability margins and associated crossover frequencies.</td>
</tr>
<tr>
<td><code>phase_crossover_frequencies(sys)</code></td>
<td>Compute frequencies and gains at intersections with real axis in Nyquist plot.</td>
</tr>
<tr>
<td><code>poles(sys)</code></td>
<td>Compute system poles.</td>
</tr>
<tr>
<td><code>zeros(sys)</code></td>
<td>Compute system zeros.</td>
</tr>
<tr>
<td><code>pzmap(sys[, plot, grid, title])</code></td>
<td>Plot a pole/zero map for a linear system.</td>
</tr>
<tr>
<td><code>root_locus(sys[, kvect, xlim, ylim, ...])</code></td>
<td>Root locus plot</td>
</tr>
<tr>
<td><code>sisotool(sys[, kvect, xlim_rlocus, ...])</code></td>
<td>Sisotool style collection of plots inspired by MATLAB's sisotool.</td>
</tr>
</tbody>
</table>

### 3.5.1 control.dcgain

`control.dcgain(sys)`

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

**Returns**

- `gain` – The zero-frequency gain, or `(inf + nanj)` if the system has a pole at the origin, `(nan + nanj)` if there is a pole/zero cancellation at the origin.

**Return type**

`ndarray`
3.5.2 control.describing_function

control.describing_function(F, A, num_points=100, zero_check=True, try_method=True)

Numerical compute the describing function of a nonlinear function

The describing function of a nonlinearity is given by magnitude and phase of the first harmonic of the function when evaluated along a sinusoidal input \( A \sin \omega t \). This function returns the magnitude and phase of the describing function at amplitude \( A \).

Parameters

- \( F \) (callable) – The function \( F() \) should accept a scalar number as an argument and return a scalar number. For compatibility with (static) nonlinear input/output systems, the output can also return a 1D array with a single element.

  If the function is an object with a method `describing_function` then this method will be used to computing the describing function instead of a nonlinear computation. Some common non-linearities use the `DescribingFunctionNonlinearity` class, which provides this functionality.

- \( A \) (array_like) – The amplitude(s) at which the describing function should be calculated.

- zero_check (bool, optional) – If True (default) then \( A \) is zero, the function will be evaluated and checked to make sure it is zero. If not, a `TypeError` exception is raised. If zero_check is False, no check is made on the value of the function at zero.

- try_method (bool, optional) – If True (default), check the \( F \) argument to see if it is an object with a `describing_function` method and use this to compute the describing function. More information in the `describing_function` method for the `DescribingFunctionNonlinearity` class.

Returns

\( df \) – The (complex) value of the describing function at the given amplitudes.

Return type

array of complex

Raises

`TypeError` – If \( A[i] < 0 \) or if \( A[i] = 0 \) and the function \( F(0) \) is non-zero.

3.5.3 control.evalfr

colorful.evalfr(sys, x, squeeze=None)

Evaluate the transfer function of an LTI system for complex frequency \( x \).

Returns the complex frequency response \( sys(x) \) where \( x \) is \( s \) for continuous-time systems and \( z \) for discrete-time systems, with \( m = sys.ninputs \) number of inputs and \( p = sys.noutputs \) number of outputs.

To evaluate at a frequency omega in radians per second, enter \( x = \omega \) * 1j for continuous-time systems, or \( x = \exp(1j * \omega * dt) \) for discrete-time systems, or use `freqresp(sys, omega)`.

Parameters

- \( sys \) (StateSpace or TransferFunction) – Linear system

- \( x \) (complex scalar or 1D array_like) – Complex frequency(s)

- squeeze (bool, optional (default=True)) – If squeeze=True, remove single-dimensional entries from the shape of the output even if the system is not SISO.
If `squeeze=False`, keep all indices (output, input and, if `omega` is array_like, frequency) even if the system is SISO. The default value can be set using `config.defaults['control.squeeze_frequency_response']`.

**Returns**

- `fresp` – The frequency response of the system. If the system is SISO and `squeeze` is not `True`, the shape of the array matches the shape of `omega`. If the system is not SISO or `squeeze` is `False`, the first two dimensions of the array are indices for the output and input and the remaining dimensions match `omega`. If `squeeze` is `True` then single-dimensional axes are removed.

**Return type**

complex ndarray

**See also:**

`freqresp`, `bode`

**Notes**

This function is a wrapper for `StateSpace.__call__()` and `TransferFunction.__call__()`.

**Examples**

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

**Todo:** Add example with MIMO system

### 3.5.4 control.freqresp

`control.freqresp(sys, omega, squeeze=None)`

Frequency response of an LTI system at multiple angular frequencies.

In general the system may be multiple input, multiple output (MIMO), where $m = sys.ninputs$ number of inputs and $p = sys.noutputs$ number of outputs.

**Parameters**

- `sys` ([`StateSpace` or `TransferFunction`]) – Linear system
- `omega` (float or 1D 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.
- `squeeze` (bool, optional) – If `squeeze=True`, remove single-dimensional entries from the shape of the output even if the system is not SISO. If `squeeze=False`, keep all indices (output, input and, if `omega` is array_like, frequency) even if the system is SISO. The default value can be set using `config.defaults['control.squeeze_frequency_response']`.

**Returns**

- `response` – Frequency response data object representing the frequency response. This object can be assigned to a tuple using
mag, phase, omega = response

where mag is the magnitude (absolute value, not dB or log10) of the system frequency response, phase is the wrapped phase in radians of the system frequency response, and omega is the (sorted) frequencies at which the response was evaluated. If the system is SISO and squeeze is not True, magnitude and phase are 1D, indexed by frequency. If the system is not SISO or squeeze is False, the array is 3D, indexed by the output, input, and frequency. If squeeze is True then single-dimensional axes are removed.

**Return type**

*FrequencyResponseData*

**See also:**

evalfr, bode

**Notes**

This function is a wrapper for `StateSpace.frequency_response()` and `TransferFunction.frequency_response()`.

**Examples**

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

**Todo:** Add example with MIMO system

```python
code
#>>> 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.
```

### 3.5.5 `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 representing the loop transfer function
  - `mag, phase, omega` *(sequence of array_like)* Input magnitude, phase (in deg.), and frequencies (rad/sec) from bode frequency response data

**Returns**
• **gm** (*float*) – Gain margin  
• **pm** (*float*) – Phase margin (in degrees)  
• **wcg** (*float or array_like*) – Crossover frequency associated with gain margin (phase crossover frequency), where phase crosses below -180 degrees.  
• **wcp** (*float or array_like*) – Crossover frequency associated with phase margin (gain crossover frequency), where gain crosses below 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**

```python
>>> sys = tf(1, [1, 2, 1, 0])
>>> gm, pm, wcg, wcp = margin(sys)
```

### 3.5.6 `control.stability_margins`

`control.stability_margins(sysdata, returnall=False, epsw=0.0, method='best')`

Calculate stability margins and associated crossover frequencies.

**Parameters**

- **sysdata** (*LTI system or (mag, phase, omega) sequence*) – 
  - *sys*  
  [LTI system] Linear SISO system representing the loop transfer function

- **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.

- **method** (*string, optional*) – Method to use (default is ‘best’): ‘poly’: use polynomial method if passed a LTI system. ‘frd’: calculate crossover frequencies using numerical interpolation of a `FrequencyResponseData` representation of the system if passed a LTI system. ‘best’: use the ‘poly’ method if possible, reverting to ‘frd’ if it is detected that numerical inaccuracy is likely to arise in the ‘poly’ method for discrete-time systems.

**Returns**

- **gm** (*float or array_like*) – Gain margin  
- **pm** (*float or array_like*) – Phase margin
• **sm (float or array_like)** – Stability margin, the minimum distance from the Nyquist plot to -1
• **wpc (float or array_like)** – Phase crossover frequency (where phase crosses -180 degrees), which is associated with the gain margin.
• **wgc (float or array_like)** – Gain crossover frequency (where gain crosses 1), which is associated with the phase margin.
• **wms (float or array_like)** – Stability margin frequency (where Nyquist plot is closest to -1)

Note that the gain margin is determined by the gain of the loop
• transfer function at the phase crossover frequency(s), the phase
• margin is determined by the phase of the loop transfer function at
• the gain crossover frequency(s), and the stability margin is determined by the frequency of maximum sensitivity (given by the magnitude of $1/(1+L)$).

### 3.5.7 control.phase_crossover_frequencies

```python
def 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

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

### 3.5.8 control.poles

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

    Parameters
    sys (StateSpace or TransferFunction) – Linear system

    Returns

    • **poles** – Array that contains the system’s poles.

    Return type
    ndarray
```
3.5.9 control.zeros

control.zeros(sys)

Compute system zeros.

Parameters
   sys (StateSpace or TransferFunction) – Linear system

Returns
   zeros – Array that contains the system’s zeros.

Return type
   ndarray

See also:
   poles, StateSpace.zeros, TransferFunction.zeros

3.5.10 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
   • poles (array) – The systems poles
   • zeros (array) – The system’s zeros.

Notes

The pzmap function calls matplotlib.pyplot.axis(‘equal’), which means that trying to reset the axis limits may not behave as expected. To change the axis limits, use matplotlib.pyplot.gca().axis(‘auto’) and then set the axis limits to the desired values.
3.5.11 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() formatstring, 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.

Notes

The root_locus function calls matplotlib.pyplot.axis(‘equal’), which means that trying to reset the axis limits may not behave as expected. To change the axis limits, use matplotlib.pyplot.gca().axis(‘auto’) and then set the axis limits to the desired values.

3.5.12 control.sisotool

control.sisotool(sys, kvect=None, xlim_rlocus=None, ylim_rlocus=None, plotstr_rlocus=None, 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. If sys is SISO, use the same system for the root locus and step response. If it is desired to see a different step response than feedback(K*loop,1), sys can be provided as a two-input, two-output system (e.g. by using
bdgalg.connect' or :func:`iosys.interconnect()`. Sisotool inserts the negative of the selected gain K between the first output and first input and uses the second input and output for computing the step response. This allows you to see the step responses of more complex systems, for example, systems with a feedforward path into the plant or in which the gain appears in the feedback path.

- **kvec** *(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- or z-plane grid.
- **omega** *(array_like)* – List of frequencies in rad/sec to be used for 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** *(array_like of two values)* – Limits of the to generate frequency vector. If Hz=True the limits are in Hz otherwise in rad/s. Ignored if omega is provided, and auto-generated if omitted.
- **omega_num** *(int)* – Number of samples to plot. Defaults to config.defaults[‘freqplot.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**

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

### 3.6 Matrix computations

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>care(A, B, Q[, R, S, E, stabilizing, ...])</td>
<td>X, L, G = care(A, B, Q, R=None) solves the continuous-time algebraic Riccati equation</td>
</tr>
<tr>
<td>dare(A, B, Q, R[, S, E, stabilizing, ...])</td>
<td>X, L, G = dare(A, B, Q, R) solves the discrete-time algebraic Riccati equation</td>
</tr>
<tr>
<td>lyap(A, Q[, C, E, method])</td>
<td>X = lyap(A, Q) solves the continuous-time Lyapunov equation</td>
</tr>
<tr>
<td>dlyap(A, Q[, C, E, method])</td>
<td>dlyap(A, Q) solves the discrete-time Lyapunov equation</td>
</tr>
<tr>
<td>ctrb(A, B)</td>
<td>Controllability matrix</td>
</tr>
<tr>
<td>obsv(A, C)</td>
<td>Observability matrix</td>
</tr>
<tr>
<td>gram(sys, type)</td>
<td>Gramian (controllability or observability)</td>
</tr>
</tbody>
</table>

### 3.6. Matrix computations
### 3.6.1 control.care

The function `control.care` solves the continuous-time algebraic Riccati equation:

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

where `A` and `Q` are square matrices of the same dimension. Further, `Q` and `R` are 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`.

#### Parameters
- `A` *(2D array_like)* – Input matrices for the Riccati equation
- `B` *(2D array_like)* – Input matrices for the Riccati equation
- `Q` *(2D array_like)* – Input matrices for the Riccati equation
- `R` *(2D array_like, optional)* – Input matrices for generalized Riccati equation
- `S` *(2D array_like, optional)* – Input matrices for generalized Riccati equation
- `E` *(2D array_like, optional)* – Input matrices for generalized Riccati equation
- `method` *(str, optional)* – Set the method used for computing the result. Current methods are ‘slycot’ and ‘scipy’. If set to None (default), try ‘slycot’ first and then ‘scipy’.

#### Returns
- `X` *(2D array (or matrix))* – Solution to the Riccati 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()`.

### 3.6.2 control.dare

The function `control.dare` solves the discrete-time algebraic Riccati equation:

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

#### Parameters
- `A` *(2D array_like)* – Input matrices for the Riccati equation
- `B` *(2D array_like)* – Input matrices for the Riccati equation
- `Q` *(2D array_like)* – Input matrices for the Riccati equation
- `R` *(2D array_like, optional)* – Input matrices for generalized Riccati equation
- `S` *(2D array_like, optional)* – Input matrices for generalized Riccati equation
- `E` *(2D array_like, optional)* – Input matrices for generalized Riccati equation
- `method` *(str, optional)* – Set the method used for computing the result. Current methods are ‘slycot’ and ‘scipy’. If set to None (default), try ‘slycot’ first and then ‘scipy’.

#### Returns
- `X` *(2D array (or matrix))* – Solution to the Riccati equation
- `L` *(1D array)* – Closed loop eigenvalues
- `G` *(2D array (or matrix))* – Gain matrix
where $A$ and $Q$ are square matrices of the same dimension. Further, $Q$ is a symmetric matrix. The function returns the solution $X$, the gain matrix $G = (B^T X B + R)^{-1} B^T X A$ and the closed loop eigenvalues $L$, i.e., the eigenvalues of $A - B G$.

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

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

where $A$, $Q$ and $E$ are square matrices of the same dimension. Further, $Q$ and $R$ are symmetric matrices. If $R$ is None, it is set to the identity matrix. 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 (generalized) eigenvalues of $A - B G$ (with respect to $E$, if specified).

**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
- **method (str, optional)** – Set the method used for computing the result. Current methods are ‘slycot’ and ‘scipy’. If set to None (default), try ’slycot’ first and then ‘scipy’.

**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()`.

### 3.6.3 control.lyap

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

$X = \text{lyap}(A, Q)$ solves the continuous-time Lyapunov equation

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

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

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

$$
AX + XQ + C = 0
$$

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

$X = \text{lyap}(A, Q, None, E)$ solves the generalized continuous-time Lyapunov equation

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

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

- **A** *(2D array_like)* – Input matrices for the Lyapunov or Sylvester equation
- **Q** *(2D array_like)* – Input matrices for the Lyapunov or Sylvester equation
- **C** *(2D array_like, optional)* – If present, solve the Sylvester equation
- **E** *(2D array_like, optional)* – If present, solve the generalized Lyapunov equation
- **method** *(str, optional)* – Set the method used for computing the result. Current methods are ‘slycot’ and ‘scipy’. If set to None (default), try ‘slycot’ first and then ‘scipy’.

Returns

**X** – 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()*.

3.6.4 control.dlyap

close.

control.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.

Parameters

- **A** *(2D array_like)* – Input matrices for the Lyapunov or Sylvester equation
- **Q** *(2D array_like)* – Input matrices for the Lyapunov or Sylvester equation
- **C** *(2D array_like, optional)* – If present, solve the Sylvester equation
- **E** *(2D array_like, optional)* – If present, solve the generalized Lyapunov equation
- **method** *(str, optional)* – Set the method used for computing the result. Current methods are ‘slycot’ and ‘scipy’. If set to None (default), try ‘slycot’ first and then ‘scipy’.

Returns

**X** – 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().

3.6.5 control ctrb

control.ctrb(A, B)
Controllability 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

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

3.6.6 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

```python
>>> 0 = obsv(A, C)
```

3.6.7 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)
- `ControlSlycot` – if slycot routines 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

```python
>>> Wc = gram(sys, 'c')
>>> Wo = gram(sys, 'o')
>>> Rc = gram(sys, 'cf'), where Wc = Rc * Rc
>>> Ro = gram(sys, 'of'), where Wo = Ro * Ro
```
3.7 Control system synthesis

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>acker(A, B, poles)</code></td>
<td>Pole placement using Ackermann method</td>
</tr>
<tr>
<td><code>create_statefbk_iosystem(sys, K[, ...])</code></td>
<td>Create an I/O system using a (full) state feedback controller</td>
</tr>
<tr>
<td><code>dlqr(A, B, Q, R[, N])</code></td>
<td>Discrete-time linear quadratic regulator design</td>
</tr>
<tr>
<td><code>h2syn(P, nmeas, ncon)</code></td>
<td>H_2 control synthesis for plant P.</td>
</tr>
<tr>
<td><code>hinfsyn(P, nmeas, ncon)</code></td>
<td>H_{inf} control synthesis for plant P.</td>
</tr>
<tr>
<td><code>lqr(A, B, Q, R[, N])</code></td>
<td>Linear quadratic regulator design</td>
</tr>
<tr>
<td><code>mixsyn(g[, w1, w2, w3])</code></td>
<td>Mixed-sensitivity H-infinity synthesis.</td>
</tr>
<tr>
<td><code>place(A, B, p)</code></td>
<td>Place closed loop eigenvalues</td>
</tr>
<tr>
<td><code>rootlocus_pid_designer(plant[, gain, sign, ...])</code></td>
<td>Manual PID controller design based on root locus using Sisotool</td>
</tr>
</tbody>
</table>

3.7.1 control.acker

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

Pole placement using Ackermann method

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

Parameters

- `A` (2D array_like) – State and input matrix of the system
- `B` (2D array_like) – State and input matrix of the system
- `poles` (1D array_like) – 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()`.

3.7.2 control.create_statefbk_iosystem

`control.create_statefbk_iosystem(sys, K, integral_action=None, xd_labels='xd[i]', ud_labels='ud[i]', estimator=None, type='linear')`

Create an I/O system using a (full) state feedback controller

This function creates an input/output system that implements a state feedback controller of the form

\[ u = ud - K_p (x - xd) - K_i \text{ integral}(C x - C x_d) \]

It can be called in the form

```
ctrl, clsys = ct.create_statefbk_iosystem(sys, K)
```
where \( \text{sys} \) is the process dynamics and \( K \) is the state (+ integral) feedback gain (eg, from LQR). The function returns the controller \( \text{ctrl} \) and the closed loop systems \( \text{clsys} \), both as I/O systems.

**Parameters**

- **\( \text{sys} \) (InputOutputSystem)** – The I/O system that represents the process dynamics. If no estimator is given, the output of this system should represent the full state.

- **\( K \) (ndarray)** – The state feedback gain. This matrix defines the gains to be applied to the system. If \( \text{integral\_action} \) is None, then the dimensions of this array should be \((\text{sys.ninputs}, \text{sys.nstates})\). If \( \text{integral\_action} \) is set to a matrix or a function, then additional columns represent the gains of the integral states of the controller.

- **\( \text{xd\_labels} \) (str or list of str, optional)** – Set the name of the signals to use for the desired state and inputs. If a single string is specified, it should be a format string using the variable \( i \) as an index. Otherwise, a list of strings matching the size of \( \text{xd} \) and \( \text{ud} \), respectively, should be used. Default is '\( \text{xd}[\{i\}] \)' for \( \text{xd\_labels} \) and '\( \text{xd}[\{i\}] \)' for \( \text{ud\_labels} \).

- **\( \text{ud\_labels} \) (str or list of str, optional)** – Set the name of the signals to use for the desired state and inputs. If a single string is specified, it should be a format string using the variable \( i \) as an index. Otherwise, a list of strings matching the size of \( \text{xd} \) and \( \text{ud} \), respectively, should be used. Default is '\( \text{xd}[\{i\}] \)' for \( \text{xd\_labels} \) and '\( \text{xd}[\{i\}] \)' for \( \text{ud\_labels} \).

- **\( \text{integral\_action} \) (None, ndarray, or func, optional)** – If this keyword is specified, the controller can include integral action in addition to state feedback. If \( \text{integral\_action} \) is an ndarray, it will be multiplied by the current and desired state to generate the error for the internal integrator states of the control law. If \( \text{integral\_action} \) is a function \( \text{h} \), that function will be called with the signature \( \text{h}(t, x, u, \text{params}) \) to obtain the outputs that should be integrated. The number of outputs that are to be integrated must match the number of additional columns in the \( K \) matrix.

- **\( \text{estimator} \) (InputOutputSystem, optional)** – If an estimator is provided, using the states of the estimator as the system inputs for the controller.

- **\( \text{type} \) (\'nonlinear\' or \'linear\', optional)** – Set the type of controller to create. The default is a linear controller implementing the LQR regulator. If the type is \'nonlinear\', a :class:NonlinearIOSystem is created instead, with the gain \( K \) as a parameter (allowing modifications of the gain at runtime).

**Returns**

- **\( \text{ctrl} \) (InputOutputSystem)** – Input/output system representing the controller. This system takes as inputs the desired state \( \text{xd} \), the desired input \( \text{ud} \), and the system state \( x \). It outputs the controller action \( u \) according to the formula \( u = \text{ud} - K(x - \text{xd}) \). If the keyword \( \text{integral\_action} \) is specified, then an additional set of integrators is included in the control system (with the gain matrix \( K \) having the integral gains appended after the state gains).

- **\( \text{clsys} \) (InputOutputSystem)** – Input/output system representing the closed loop system. This systems takes as inputs the desired trajectory \((\text{xd}, \text{ud})\) and outputs the system state \( x \) and the applied input \( u \) (vertically stacked).
3.7.3 control.dlqr

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

Discrete-time linear quadratic regulator design

The dlqr() function computes the optimal state feedback controller \( u[n] = -K x[n] \) that minimizes the quadratic cost

\[
J = \sum_{0}^{\infty} (x[n]'Qx[n] + u[n]'Ru[n] + 2x[n]'Nu[n])
\]

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

- dlqr(dsys, Q, R)
- dlqr(dsys, Q, R, N)
- dlqr(A, B, Q, R)
- dlqr(A, B, Q, R, N)

where dsys is a discrete-time StateSpace system, and A, B, Q, R, and N are 2d arrays of appropriate dimension (dsys.dt must not be 0.)

Parameters

- A (2D array) – Dynamics and input matrices
- B (2D array) – Dynamics and input matrices
- dsys (LTI StateSpace) – Discrete-time linear system
- Q (2D array) – State and input weight matrices
- R (2D array) – State and input weight matrices
- N (2D array, optional) – Cross weight matrix
- integral_action (ndarray, optional) – If this keyword is specified, the controller includes integral action in addition to state feedback. The value of the integral_action keyword should be an ndarray that will be multiplied by the current to generate the error for the internal integrator states of the control law. The number of outputs that are to be integrated must match the number of additional rows and columns in the Q matrix.
- method (str, optional) – Set the method used for computing the result. Current methods are ‘slycot’ and ‘scipy’. If set to None (default), try ‘slycot’ first and then ‘scipy’.

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:

lqr, lqe, dlqe
Notes

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

Examples

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

3.7.4 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` (controller to stabilize P (State-space sys))

Raises

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

See also:

- `StateSpace`

Examples

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

3.7.5 control.hinf

`control.hinf(P, nmeas, ncon)`

H\(_\infty\) 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**

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

### 3.7.6 control.lqr

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

Linear quadratic regulator design

The lqr() function computes the optimal state feedback controller \( u = -K x \) that minimizes the quadratic cost

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

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

- \( K, S, E = \text{lqr}(\text{sys}, Q, R) \)
- \( K, S, E = \text{lqr}(\text{sys}, Q, R, N) \)
- \( K, S, E = \text{lqr}(A, B, Q, R) \)
- \( K, S, E = \text{lqr}(A, B, Q, R, N) \)

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_like)* – Dynamics and input matrices
- **B** *(2D array_like)* – Dynamics and input matrices
- **sys** *(LTI StateSpace system)* – Linear system
- **Q** *(2D array)* – State and input weight matrices
- **R** *(2D array)* – State and input weight matrices
- **N** *(2D array, optional)* – Cross weight matrix
- **integral_action** *(ndarray, optional)* – If this keyword is specified, the controller includes integral action in addition to state feedback. The value of the `integral_action` keyword should be an ndarray that will be multiplied by the current to generate the error for the internal integrator states of the control law. The number of outputs that are to be integrated must match the number of additional rows and columns in the **Q** matrix.
• **method**(str, optional) – Set the method used for computing the result. Current methods are ‘slycot’ and ‘scipy’. If set to None (default), try ‘slycot’ first and then ‘scipy’.

**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, dlqr, dlqe

**Notes**

1. If the first argument is an LTI object, then this object will be used to define the dynamics and input matrices. Furthermore, if the LTI object corresponds to a discrete time system, the dlqr() function will be called.

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

**Examples**

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

### 3.7.7 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** (At least one of) –
• **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) –
• **w1** –
• **w2** –
• **None. (and w3 must not be)** –

**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`

### 3.7.8 `control.place`

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

Place closed loop eigenvalues

\[
K = \text{place}(A, B, p)
\]

**Parameters**

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

**Returns**

- *K* – Gain such that \(A - B \cdot K\) 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 \(\text{rank}(B)\) times.

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

References

Examples

```python
>>> 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()`.

3.7.9 control.rootlocus_pid_designer

```python
control.rootlocus_pid_designer(plant, gain=’P’, sign=1, input_signal=’r’, Kp0=0, Ki0=0, Kd0=0, tau=0.01, C_ff=0, derivative_in_feedback_path=False, plot=True)
```

Manual PID controller design based on root locus using Sisotool

Uses Sisotool to investigate the effect of adding or subtracting an amount \( \delta K \) to the proportional, integral, or derivative (PID) gains of a controller. One of the PID gains, \( K_p \), \( K_i \), or \( K_d \), respectively, can be modified at a time. Sisotool plots the step response, frequency response, and root locus.

When first run, \( \delta K \) is set to 0; click on a branch of the root locus plot to try a different value. Each click updates plots and prints the corresponding \( \delta K \). To tune all three PID gains, repeatedly call `rootlocus_pid_designer`, and select a different gain each time (‘P’, ‘I’, or ‘D’). Make sure to add the resulting \( \delta K \) to your chosen initial gain on the next iteration.

Example: to examine the effect of varying \( K_p \) starting from an initial value of 10, use the arguments `gain=’P’, Kp0=10`. Suppose a \( \delta K \) value of 5 gives satisfactory performance. Then on the next iteration, to tune the derivative gain, use the arguments `gain=’D’, Kp0=15`.

By default, all three PID terms are in the forward path \( C_f \) in the diagram shown below, that is,

\[
C_f = K_p + \frac{K_i}{s} + \frac{K_d}{s}\frac{1}{\tau_s + 1}.
\]

If \( plant \) is a discrete-time system, then the proportional, integral, and derivative terms are given instead by \( K_p \), \( K_i dt(2(z+1))/(z-1) \), and \( K_d dt(z-1)/z \), respectively.

\[
\text{———} \rightarrow C_{ff} \quad \text{———} \rightarrow C_f \quad \text{———} \rightarrow \text{plant} \quad \text{———} \rightarrow C_b
\]

It is also possible to move the derivative term into the feedback path \( C_b \) using `derivative_in_feedback_path=True`. This may be desired to avoid that the plant is subject to an impulse function when the reference \( r \) is a step input. \( C_b \) is otherwise set to zero.

If \( plant \) is a 2-input system, the disturbance \( d \) is fed directly into its second input rather than being added to \( u \).

Remark: It may be helpful to zoom in using the magnifying glass on the plot. Just make sure to deactivate magnification mode when you are done by clicking the magnifying glass. Otherwise you will not be able to be able to choose a gain on the root locus plot.
Parameters

- **plant** (LTI (TransferFunction or StateSpace system)) – The dynamical system to be controlled
- **gain** (string (optional)) – Which gain to vary by deltaK. Must be one of ‘P’, ‘I’, or ‘D’ (proportional, integral, or derivative)
- **sign** (int (optional)) – The sign of deltaK gain perturbation
- **input** (string (optional)) – The input used for the step response; must be ‘r’ (reference) or ‘d’ (disturbance) (see figure above)
- **Kp0** (float (optional)) – Initial values for proportional, integral, and derivative gains, respectively
- **Ki0** (float (optional)) – Initial values for proportional, integral, and derivative gains, respectively
- **Kd0** (float (optional)) – Initial values for proportional, integral, and derivative gains, respectively
- **tau** (float (optional)) – The time constant associated with the pole in the continuous-time derivative term. This is required to make the derivative transfer function proper.
- **C_ff** (float or LTI system (optional)) – Feedforward controller. If LTI, must have timebase that is compatible with plant.
- **derivative_in_feedback_path** (bool (optional)) – Whether to place the derivative term in feedback transfer function $C_b$ instead of the forward transfer function $C_f$.
- **plot** (bool (optional)) – Whether to create Sisotool interactive plot.

Returns

- **closedloop** – The closed-loop system using initial gains.

Return type

class: StateSpace system

3.8 Model simplification tools

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>minreal</strong></td>
<td>Eliminates uncontrollable or unobservable states in state-space models or cancelling pole-zero pairs in transfer functions.</td>
</tr>
<tr>
<td><strong>balred</strong></td>
<td>Balanced reduced order model of sys of a given order.</td>
</tr>
<tr>
<td><strong>hsvd</strong></td>
<td>Calculate the Hankel singular values.</td>
</tr>
<tr>
<td><strong>modred</strong></td>
<td>Model reduction of sys by eliminating the states in ELIM using a given method.</td>
</tr>
<tr>
<td><strong>era</strong></td>
<td>Calculate an ERA model of order $r$ based on the impulse-response data YY.</td>
</tr>
<tr>
<td><strong>markov</strong></td>
<td>Calculate the first $m$ Markov parameters [D CB CAB ...] from input $U$, output $Y$.</td>
</tr>
</tbody>
</table>
3.8.1 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*

3.8.2 control.balred

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

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


Parameters

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

Returns

rsys – 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 slycop routine ab09ad, ab09md, or ab09nd is not found
- **ValueError** – if there are more unstable modes than any value in orders
Examples

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

### 3.8.3 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

**Notes**

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

**Examples**

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

### 3.8.4 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

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

### 3.8.5 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

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

### 3.8.6 control.markov

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

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

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

60 Chapter 3. Function reference
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*. Default value is False.

**Returns**

\( H \) – First \( m \) Markov parameters, \([D \ C B \ \text{CAB} \ldots]\)

**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**

```python
code
>>> 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)
```

3.9 Nonlinear system support

**describing_function**(F, A[, num_points, ...])  
Numerical compute the describing function of a nonlinear function

**find_eqpt**(sys, x0[, u0, y0, t, params, iu, ...])  
Find the equilibrium point for an input/output system.

**interconnect**(syslist[, connections, ...])  
Interconnect a set of input/output systems.

**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, **kwargs)  
Create an I/O system from a state space linear system.

**summing_junction**(inputs, output, ...)  
Create a summing junction as an input/output system.

**tf2io**(sys)  
Convert a transfer function into an I/O system

**flatsys.point_to_point**(sys, timepts[, x0, ...])  
Compute trajectory between an initial and final conditions.

### 3.9.1 control.find_eqpt

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

Find the equilibrium point for an input/output system.

Returns the value of an equilibrium 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.

### 3.9.2 control.interconnect

control.interconnect(syslist, connections=None, inplist=[], outlist=[], params={}, check_unused=True, ignore_inputs=None, ignore_outputs=None, warn_duplicate=None, **kwargs)

Interconnect a set of input/output systems.

This function creates a new system that is an interconnection of a set of input/output systems. If all of the input systems are linear I/O systems (type LinearIOSystem) then the resulting system will be a linear interconnected I/O system (type LinearICSystem) with the appropriate inputs, outputs, and states. Otherwise, an interconnected I/O system (type InterconnectedSystem) will be created.

Parameters

- syslist (list of InputOutputSystems) – The list of input/output systems to be connected

- connections (list of connections, optional) – Description of the internal connections between the subsystems:

  [connection1, connection2, ...]

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

  [input-spec, output-spec1, output-spec2, ...]

The input-spec can be in a number of different forms. The lowest level representation is 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.

Similarly, each output-spec should describe an output signal from one of the subsystems. The lowest level representation is 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 `interconnect` function will attempt to create the interconnection map by connecting all signals with the same base names (ignoring the system name). Specifically, for each input signal name in the list of systems, if that signal name corresponds to the output signal in any of the systems, it will be connected to that input (with a summation across all signals if the output name occurs in more than one system).

The `connections` keyword can also be set to `False`, which will leave the connection map empty and it can be specified instead using the low-level `set_connect_map()` method.

- **inplist** (list of input connections, optional) – List of connections 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 for a connection are thus of the form:

  [input-spec1, input-spec2, ...]

Each system input is added to the input for the listed subsystem. If the system input connects to only one subsystem input, a single input specification can be given (without the inner list).

If omitted, the input map can be specified using the `set_input_map()` method.

- **outlist** (list of output connections, optional) – List of connections for how the outputs from the subsystems are mapped to overall system outputs. The output connection description 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 an output connection contains more than one signal specification, then those signals are added together (multiplying by the any gain term) to form the system output.

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. The default is `None`, in which case the states will be given names 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 = 0: continuous time system (default)
  - dt > 0: discrete time system with sampling period ‘dt’
- `dt = True`: discrete time with unspecified sampling period
- `dt = None`: no timebase specified

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

- **check-unused** (`bool`) – If True, check for unused sub-system signals. This check is not done if connections is False, and neither input nor output mappings are specified.

- **ignore-inputs** (`list of input-spec`) – A list of sub-system inputs known not to be connected. This is only used in checking for unused signals, and does not disable use of the input.

Besides the usual input-spec forms (see `connections`), an input-spec can be just the signal base name, in which case all signals from all sub-systems with that base name are considered ignored.

- **ignore-outputs** (`list of output-spec`) – A list of sub-system outputs known not to be connected. This is only used in checking for unused signals, and does not disable use of the output.

Besides the usual output-spec forms (see `connections`), an output-spec can be just the signal base name, in which all outputs from all sub-systems with that base name are considered ignored.

- **warn-duplicate** (`None`, `True`, or `False`) – Control how warnings are generated if duplicate objects or names are detected. In `None` (default), then warnings are generated for systems that have non-generic names. If `False`, warnings are not generated and if `True` then warnings are always generated.

**Example**

```python
>>> P = control.LinearIOSystem(
    control.rss(2, 2, 2, strictly_proper=True), name='P')
>>> C = control.LinearIOSystem(control.rss(2, 2, 2), name='C')
>>> T = control.interconnect([
    P, C],
    connections = [
        ['P.u[0]', 'C.y[0]'], ['P.u[1]', 'C.y[1]'],
        ['C.u[0]', '-P.y[0]'], ['C.u[1]', '-P.y[1]'],
    ],
    inplist = ['C.u[0]', 'C.u[1]'],
    outlist = ['P.y[0]', 'P.y[1]'],
)
```

For a SISO system, this example can be simplified by using the `summing_block()` function and the ability to automatically interconnect signals with the same names:

```python
>>> P = control.tf2io(control.tf(1, [1, 0]), inputs='u', outputs='y')
>>> C = control.tf2io(control.tf(10, [1, 1]), inputs='e', outputs='u')
>>> sumblk = control.summing_junction(inputs=['r', '-y'], output='e')
>>> T = control.interconnect([P, C, sumblk], inputs='r', outputs='y')
```
Notes

If a system is duplicated in the list of systems to be connected, a warning is generated and a copy of the system is created with the name of the new system determined by adding the prefix and suffix strings in config.defaults[‘iosys.linearized_system_name_prefix’] and config.defaults[‘iosys.linearized_system_name_suffix’], with the default being to add the suffix ‘$copy$’ to the system name.

It is possible to replace lists in most of arguments with tuples instead, but strictly speaking the only use of tuples should be in the specification of an input- or output-signal via the tuple notation (subsys_i, signal_j, gain) (where gain is optional). If you get an unexpected error message about a specification being of the wrong type, check your use of tuples.

In addition to its use for general nonlinear I/O systems, the interconnect() function allows linear systems to be interconnected using named signals (compared with the connect() function, which uses signal indices) and to be treated as both a StateSpace system as well as an InputOutputSystem.

The input and output keywords can be used instead of inputs and outputs, for more natural naming of SISO systems.

3.9.3 control.linearize

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

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

This function computes the linearization of an input/output system at a given state and input value and returns a StateSpace object. The evaluation 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 equilibrium state).
- **ueq** (array) – The input at which the linearization will be evaluated (does not need to correspond to an equilibrium 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.
- **copy** (bool, Optional) – If copy is True, copy the names of the input signals, output signals, and states to the linearized system. If name is not specified, the system name is set to the input system name with the string ‘_linearized’ appended.
- **name** (string, optional) – Set the name of the linearized system. If not specified and if copy is False, a generic name <sys[id]> is generated with a unique integer id. If copy is True, the new system name is determined by adding the prefix and suffix strings in config.defaults[‘iosys.linearized_system_name_prefix’] and config.defaults[‘iosys.linearized_system_name_suffix’], with the default being to add the suffix ‘$linearized$’.

Returns

- **ss_sys** (LinearIOSystem) – The linearization of the system, as a LinearIOSystem object (which is also a StateSpace object).
• Additional Parameters

• **inputs** (int, list of str or None, optional) – Description of the system inputs. If not specified, the original system inputs are used. See *InputOutputSystem* for more information.

• **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**.

### 3.9.4 control.ss2io

control.ss2io(*args, **kwargs)

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.

### 3.9.5 control.summing_junction

control.summing_junction(inputs=None, output=None, dimension=None, prefix='u', **kwargs)

Create a summing junction as an input/output system.

This function creates a static input/output system that outputs the sum of the inputs, potentially with a change in sign for each individual input. The input/output system that is created by this function can be used as a component in the `interconnect()` function.

**Parameters**

• **inputs** (int, string or list of strings) – Description of the inputs to the summing junction. This can be given as an integer count, a string, or a list of strings. If an integer count is specified, the names of the input signals will be of the form \( u[i] \).

• **output** (string, optional) – Name of the system output. If not specified, the output will be ‘y’.

• **dimension** (int, optional) – The dimension of the summing junction. If the dimension is set to a positive integer, a multi-input, multi-output summing junction will be created. The input and output signal names will be of the form \(<signal>[i]\) where **signal** is the input/output signal name specified by the **inputs** and **output** keywords. Default value is None.

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

• **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].

**Returns**

sys – Linear input/output system object with no states and only a direct term that implements the summing junction.

**Return type**

static LinearIOSystem

### 3.9. Nonlinear system support
Example

```python
>>> P = control.tf2io(ct.tf(1, \([1, 0]\)], inputs='u', outputs='y')
>>> C = control.tf2io(ct.tf(10, \([1, 1]\)], inputs='e', outputs='u')
>>> sumblk = control.summing_junction(inputs=['r', '-y'], output='e')
>>> T = control.interconnect((P, C, sumblk), inputs='r', outputs='y')
```

### 3.9.6 `control.tf2io`

`control.tf2io(sys)`

Convert a transfer function into an I/O system

The function accepts either 1 or 2 parameters:

- `tf2io(sys)`
  - Convert a linear system into space space form. Always creates a new system, even if sys is already a StateSpace object.

- `tf2io(num, den)`
  - Create a linear I/O 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.
- `inputs` *(str, or list of str, optional)* – List of strings that name the individual signals of the transformed system. If not given, the inputs and outputs are the same as the original system.
- `outputs` *(str, or list of str, optional)* – List of strings that name the individual signals of the transformed system. If not given, the inputs and outputs are the same as the original system.
- `name` *(string, optional)* – System name. If unspecified, a generic name `<sys[id]>` is generated with a unique integer id.

**Returns**

- `out` – New I/O system (in state space form).

**Return type**

- `LinearIOSystem`

**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:**

- `ss2io`, `tf2ss`
Examples

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

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

3.9.7 control.flatsys.point_to_point

`control.flatsys.point_to_point(sys, timepts, x0=0, u0=0, xf=0, uf=0, T0=0, cost=None, basis=None, trajectory_constraints=None, initial_guess=None, params=None, **kwargs)`

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 produces the flag of flat outputs and a system `flatsys.reverse()` that takes the flag of the flat output and produces the state and input.

- `timepts (float or 1D array_like)` – The list of points for evaluating cost and constraints, as well as the time horizon. If given as a float, indicates the final time for the trajectory (corresponding to xf).

- `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.

- `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).

- `cost (callable)` – Function that returns the integral cost given the current state and input. Called as `cost(x, u)`.

- `trajectory_constraints (list of tuples, optional)` – List of constraints that should hold at each point in the time vector. Each element of the list should consist of a tuple with first element given by `scipy.optimize.LinearConstraint` or `scipy.optimize.NonlinearConstraint` and the remaining elements of the tuple are the arguments that would be passed to those functions. The following tuples are supported:
– (LinearConstraint, A, lb, ub): The matrix A is multiplied by stacked vector of the state and input at each point on the trajectory for comparison against the upper and lower bounds.
– (NonlinearConstraint, fun, lb, ub): a user-specific constraint function \( \text{fun}(x, u) \) is called at each point along the trajectory and compared against the upper and lower bounds.

The constraints are applied at each time point along the trajectory.

• **minimize_kwargs** *(str, optional) – Pass additional keywords to scipy.optimize.minimize().*

**Returns**

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

**Return type**

*SystemTrajectory* object

**Notes**

Additional keyword parameters can be used to fine tune the behavior of the underlying optimization function. See **minimize_*** keywords in *OptimalControlProblem()* for more information.

### 3.10 Stochastic system support

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>correlation(T, X[, Y, squeeze])</code></td>
<td>Compute the correlation of time signals.</td>
</tr>
<tr>
<td><code>create_estimator_iosystem(sys, QN, RN[, P0, ...])</code></td>
<td>Create an I/O system implementing a linear quadratic estimator</td>
</tr>
<tr>
<td><code>dlqe(A, G, C, QN, RN[, N])</code></td>
<td>Linear quadratic estimator design (Kalman filter) for discrete-time systems.</td>
</tr>
<tr>
<td><code>lqe(A, G, C, QN, RN[, N, NN])</code></td>
<td>Linear quadratic estimator design (Kalman filter) for continuous-time systems.</td>
</tr>
<tr>
<td><code>white_noise(T, Q[, dt])</code></td>
<td>Generate a white noise signal with specified intensity.</td>
</tr>
</tbody>
</table>

#### 3.10.1 control.correlation

**control.correlation**(\(T, X, Y=\text{None}, \text{squeeze}=\text{True})\)

Compute the correlation of time signals.

For a time series \( X(t) \) (and optionally \( Y(t) \)), the \( \text{correlation()} \) function computes the correlation matrix \( \text{E}(X'(t+\tau) X(t)) \) or the cross-correlation matrix \( \text{E}(X'(t+\tau) Y(t)) \):

\[
\tau, R_{\tau} = \text{correlation}(T, X[, Y])
\]

The signal \( X \) (and \( Y \), if present) represent a continuous time signal sampled at times \( T \). The return value provides the correlation \( R_{\tau} \) between \( X(t+\tau) \) and \( X(t) \) at a set of time offsets \( \tau \).

**Parameters**

• **\( T \)** *(1D array_like) – Sample times for the signal(s).*
• **\( X \)** *(1D or 2D array_like) – Values of the signal at each time in \( T \). The signal can either be scalar or vector values.*
• \(Y(1D \text{ or } 2D \text{ array_like, optional})\) – If present, the signal with which to compute the correlation. Defaults to \(X\).

• \texttt{squeeze(bool, optional)} – If True, squeeze \(\tau_o\) to remove extra dimensions (useful if the signals are scalars).

### 3.10.2 \texttt{control.create_estimator_iosystem}

\texttt{control.create_estimator_iosystem}(
\texttt{sys, QN, RN, P0=None, G=None, C=None, state_labels='\texttt{xhat[\{i\]}}, output_labels='\texttt{xhat[\{i\]}}, covariance_labels='\texttt{P[\{i\},\{j\}]}'},
sensor_labels=None)

Create an I/O system implementing a linear quadratic estimator

This function creates an input/output system that implements a state estimator of the form

\[
\begin{align*}
x_{hat}[k+1] &= A \cdot x[k] + B \cdot u[k] - L \cdot (C \cdot x_{hat}[k] - y[k])
\end{align*}
\]

\[
\begin{align*}
P[k+1] &= A \cdot P \cdot A^T + F \cdot QN \cdot F^T - A \cdot P \cdot C^T \cdot R_{\text{ps}}^{-1} \cdot C \cdot P \cdot A \cdot L
\end{align*}
\]

where \(R_{\text{ps}} = RN + C \cdot P \cdot C^T\). It can be called in the form

\[
\text{estim} = \text{ct.create_estimator_iosystem}(
\text{sys, QN, RN})
\]

where \(\text{sys}\) is the process dynamics and \(QN\) and \(RN\) are the covariance of the disturbance noise and sensor noise. The function returns the estimator \(\text{estim}\) as I/O system with a parameter \texttt{correct}\ that can be used to turn off the correction term in the estimation (for forward predictions).

#### Parameters

- \texttt{sys (InputOutputSystem)} – The I/O system that represents the process dynamics. If no estimator is given, the output of this system should represent the full state.
- \texttt{QN (ndarray)} – Process and sensor noise covariance matrices.
- \texttt{RN (ndarray)} – Process and sensor noise covariance matrices.
- \texttt{P0(ndarray, optional)} – Initial covariance matrix. If not specified, defaults to the steady state covariance.
- \texttt{G(ndarray, optional)} – Disturbance matrix describing how the disturbances enter the dynamics. Defaults to \(\text{sys.B}\).
- \texttt{C(ndarray, optional)} – If the system has all full states output, define the measured values to be used by the estimator. Otherwise, use the system output as the measured values.
- \texttt{\{state (str or list of str, optional)\}} – Set the name of the signals to use for the internal state, covariance, sensors, and outputs (state estimate). If a single string is specified, it should be a format string using the variable \(i\) as an index (or \(i\) and \(j\) for covariance). Otherwise, a list of strings matching the size of the respective signal should be used. Default is \('xhat[\{i\}]'\) for state and output labels, \('y[\{i\}]'\) for output labels and \('P[\{i\},\{j\}]'\) for covariance labels.
- \texttt{covariance (str or list of str, optional)} – Set the name of the signals to use for the internal state, covariance, sensors, and outputs (state estimate). If a single string is specified, it should be a format string using the variable \(i\) as an index (or \(i\) and \(j\) for covariance). Otherwise, a list of strings matching the size of the respective signal should be used. Default is \('xhat[\{i\}]'\) for state and output labels, \('y[\{i\}]'\) for output labels and \('P[\{i\},\{j\}]'\) for covariance labels.
- \texttt{sensor (str or list of str, optional)} – Set the name of the signals to use for the internal state, covariance, sensors, and outputs (state estimate). If a single string is specified, it should be a format string using the variable \(i\) as an index (or \(i\) and \(j\) for covariance).
Otherwise, a list of strings matching the size of the respective signal should be used. Default is \(xhat[\{i\}]\) for state and output labels, \(y[\{i\}]\) for output labels and \(P[\{i\},\{j\}]\) for covariance labels.

- **output_labels** *(str or list of str, optional)* – Set the name of the signals to use for the internal state, covariance, sensors, and outputs (state estimate). If a single string is specified, it should be a format string using the variable \(i\) as an index (or \(i\) and \(j\) for covariance). Otherwise, a list of strings matching the size of the respective signal should be used. Default is \(xhat[\{i\}]\) for state and output labels, \(y[\{i\}]\) for output labels and \(P[\{i\},\{j\}]\) for covariance labels.

**Returns**
- **estim** – Input/output system representing the estimator. This system takes the system input and output and generates the estimated state.

**Return type**
- InputOutputSystem

**Notes**

This function can be used with the `create_statefbk_iosystem()` function to create a closed loop, output-feedback, state space controller:

```python
K, _, _ = ct.lqr(sys, Q, R) est = ct.create_estimator_iosystem(sys, QN, RN, P0) ctrl, clsys = ct.create_statefbk_iosystem(sys, K, estimator=est)
```

The estimator can also be run on its own to process a noisy signal:

```python
resp = ct.input_output_response(est, T, [Y, U], [X0, P0])
```

If desired, the `correct` parameter can be set to `False` to allow prediction with no additional sensor information:

```python
resp = ct.input_output_response(est, T, 0, [X0, P0], param={'correct': False})
```

### 3.10.3 control.dlqe

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

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

\[
x[n + 1] = Ax[n] + Bu[n] + Gu[n]
\]
\[
y[n] = Cx[n] + Du[n] + v[n]
\]

with unbiased process noise \(w\) and measurement noise \(v\) with covariances

\[
Ew'w' = QN, Ev'v' = RN, Ew'v' = NN
\]

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

\[
x_e[n + 1] = Ax_e[n] + Bu[n] + L(y[n] - Cx_e[n] - Du[n])
\]

produces a state estimate \(x_e[n]\) 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_like)* – Dynamics and noise input matrices
• $G$ (2D array_like) – Dynamics and noise input matrices
• $QN$ (2D array_like) – Process and sensor noise covariance matrices
• $RN$ (2D array_like) – Process and sensor noise covariance matrices
• $NN$ (2D array, optional) – Cross covariance matrix (not yet supported)
• `method` (str, optional) – Set the method used for computing the result. Current methods are ‘slycot’ and ‘scipy’. If set to None (default), try ‘slycot’ first and then ‘scipy’.

**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**

```python
>>> L, P, E = dlqe(A, G, C, QN, RN)
>>> L, P, E = dlqe(A, G, C, QN, RN, NN)
```

See also:

dlqr, lqe, lqr

### 3.10.4 control.lqe

control.lqe(A, G, C, QN, RN[, NN ])

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

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

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

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

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

The function can be called with either 3, 4, 5, or 6 arguments:
L, P, E = lqe(sys, QN, RN)
L, P, E = lqe(sys, QN, RN, NN)
L, P, E = lqe(A, G, C, QN, RN)
L, P, E = lqe(A, G, C, QN, RN, NN)

where sys is an LTI object, and A, G, C, QN, RN, and NN are 2D arrays or matrices of appropriate dimension.

Parameters

- A (2D array_like) – Dynamics, process noise (disturbance), and output matrices
- G (2D array_like) – Dynamics, process noise (disturbance), and output matrices
- C (2D array_like) – Dynamics, process noise (disturbance), and output matrices
- sys (LTI (StateSpace or TransferFunction)) – Linear I/O system, with the process noise input taken as the system input.
- QN (2D array_like) – Process and sensor noise covariance matrices
- RN (2D array_like) – Process and sensor noise covariance matrices
- NN (2D array, optional) – Cross covariance matrix. Not currently implemented.
- method (str, optional) – Set the method used for computing the result. Current methods are ‘slycot’ and ‘scipy’. If set to None (default), try ‘slycot’ first and then ‘scipy’.

Returns

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

\[ AP + PA^T - (PC^T + GN)R^{-1}(CP + N^TG^T) + GQG^T = 0 \]

- E (1D array) – Eigenvalues of estimator poles \( \text{eig}(A - L C) \)

Notes

1. If the first argument is an LTI object, then this object will be used to define the dynamics, noise and output matrices. Furthermore, if the LTI object corresponds to a discrete time system, the dlqe() function will be called.

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

Examples

```python
>>> L, P, E = lqe(A, G, C, QN, RN)
>>> L, P, E = lqe(A, G, C, Q, RN, NN)
```

See also:

lqr, dlqe, dlqr
3.10.5 control.white_noise

control.white_noise(T, Q, dt=0)

Generate a white noise signal with specified intensity.

This function generates a (multi-variable) white noise signal of specified intensity as either a sampled continuous time signal or a discrete time signal. A white noise signal along a 1D array of linearly spaced set of times T can be computing using

\[ V = ct.white_noise(T, Q, dt) \]

where Q is a positive definite matrix providing the noise intensity.

In continuous time, the white noise signal is scaled such that the integral of the covariance over a sample period is Q, thus approximating a white noise signal. In discrete time, the white noise signal has covariance Q at each point in time (without any scaling based on the sample time).

3.11 Utility functions and conversions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>augw(g[, w1, w2, w3])</td>
<td>Augment plant for mixed sensitivity problem.</td>
</tr>
<tr>
<td>bdschur(a[, condmax, sort])</td>
<td>Block-diagonal Schur decomposition</td>
</tr>
<tr>
<td>canonical_form(xsys[, form])</td>
<td>Convert a system into canonical form</td>
</tr>
<tr>
<td>damp(sys[, doprint])</td>
<td>Compute natural frequency, damping ratio, and poles of a system</td>
</tr>
<tr>
<td>db2mag(db)</td>
<td>Convert a gain in decibels (dB) to a magnitude</td>
</tr>
<tr>
<td>isctime(sys[, strict])</td>
<td>Check to see if a system is a continuous-time system</td>
</tr>
<tr>
<td>isdtime(sys[, strict])</td>
<td>Check to see if a system is a discrete time system</td>
</tr>
<tr>
<td>issiso(sys[, strict])</td>
<td>Check to see if a system is single input, single output</td>
</tr>
<tr>
<td>issys(obj)</td>
<td>Return True if an object is a system, otherwise False</td>
</tr>
<tr>
<td>mag2db(mag)</td>
<td>Convert a magnitude to decibels (dB)</td>
</tr>
<tr>
<td>modal_form(xsys[, condmax, sort])</td>
<td>Convert a system into modal canonical form</td>
</tr>
<tr>
<td>observable_form(xsys)</td>
<td>Convert a system into observable canonical form</td>
</tr>
<tr>
<td>pade(T[, n, numdeg])</td>
<td>Create a linear system that approximates a delay.</td>
</tr>
<tr>
<td>reachable_form(xsys)</td>
<td>Convert a system into reachable canonical form</td>
</tr>
<tr>
<td>reset_defaults()</td>
<td>Reset configuration values to their default (initial) values.</td>
</tr>
<tr>
<td>sample_system(sysc, Ts[, method, alpha, ...])</td>
<td>Convert a continuous time system to discrete time by sampling</td>
</tr>
<tr>
<td>ss2tf(sys)</td>
<td>Transform a state space system to a transfer function.</td>
</tr>
<tr>
<td>ssdata(sys)</td>
<td>Return state space data objects for a system</td>
</tr>
<tr>
<td>tf2ss(sys)</td>
<td>Transform a transfer function to a state space system.</td>
</tr>
<tr>
<td>tfdata(sys)</td>
<td>Return transfer function data objects for a system</td>
</tr>
<tr>
<td>timebase(sys[, strict])</td>
<td>Return the timebase for a system</td>
</tr>
<tr>
<td>timebaseEqual(sys1, sys2)</td>
<td>Check to see if two systems have the same timebase</td>
</tr>
<tr>
<td>unwrap(angle[, period])</td>
<td>Unwrap a phase angle to give a continuous curve</td>
</tr>
<tr>
<td>use_fbs_defaults()</td>
<td>Use Feedback Systems (FBS) compatible settings.</td>
</tr>
<tr>
<td>use_matlab_defaults()</td>
<td>Use MATLAB compatible configuration settings.</td>
</tr>
<tr>
<td>use_numpy_matrix([flag, warn])</td>
<td>Turn on/off use of Numpy matrix class for state space operations.</td>
</tr>
</tbody>
</table>
3.11.1 control.augw

```python
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` *(ny-by-ny for w1 and)*
- `p` *(augmented plant; StateSpace object)*
- `None` *(If a weighting is)*
- `least` *(no augmentation is done for it. At)*
- `None` *(one weighting must not be)*
- `scalar` *(If a weighting w is)*
- `I*w` *(it will be replaced by)*
- `is` *(where I)*
- `w3`
- `w2` *(and nu-by-nu for)*

**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.bdschur

```python
control.bdschur(a, condmax=None, sort=None)
```

Block-diagonal Schur decomposition

**Parameters**

- `a` *(M, M) array_like)* – Real matrix to decompose
- `condmax` *(None or float, optional)* – If None (default), use 1/sqrt(eps), which is approximately 1e8
- `sort` *(None, 'continuous', 'discrete')* – Block sorting; see below.

**Returns**

- `amodal` *(M, M real ndarray)* – Block-diagonal Schur decomposition of `a`
• **tmodal** \((M, M) \text{ real ndarray}\) – Similarity transform relating \(a\) and \(amodal\)

• **blksizes** \((N,) \text{ int ndarray}\) – Array of Schur block sizes

**Notes**

If \(sort\) is None, the blocks are not sorted.

If \(sort\) is ‘continuous’, the blocks are sorted according to associated eigenvalues. The ordering is first by real part of eigenvalue, in descending order, then by absolute value of imaginary part of eigenvalue, also in decreasing order.

If \(sort\) is ‘discrete’, the blocks are sorted as for ‘continuous’, but applied to log of eigenvalues (i.e., continuous-equivalent eigenvalues).

### 3.11.3 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** (`str`) –

  **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** \((M, M) \text{ real ndarray}\) – Coordinate transformation matrix, \(z = T * x\)

### 3.11.4 control.damp

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

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

The function takes 1 or 2 parameters

**Parameters**

• **sys** (`LTI (StateSpace or TransferFunction)`) – A linear system object

• **doprint** – if true, print table with values

**Returns**

• **wn** (array) – Natural frequencies of the poles

• **damping** (array) – Damping values

• **poles** (array) – Pole locations

**Algorithm**
• If the system is continuous, \(- wn = \text{abs}(\text{poles}) Z = -\text{real}(\text{poles})/\text{poles}\).
• If the system is discrete, the discrete poles are mapped to their equivalent location in the s-plane via \(- s = \log_{10}(\text{poles})/dt\)
• and \(- wn = \text{abs}(s) Z = -\text{real}(s)/wn\).

See also:

pole

3.11.5 control.db2mag

control.db2mag(db)

Convert a gain in decibels (dB) to a magnitude

If \(A\) is magnitude,
\[
db = 20 \times \log_{10}(A)
\]

Parameters

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

Returns

- \(mag\) – corresponding magnitudes

Return type

float or ndarray

3.11.6 control.isctime

control.isctime(sys, strict=False)

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

Parameters

- \(sys\) (I/O or 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.isdtime

control.isdtime(sys, strict=False)

Check to see if a system is a discrete time system

Parameters

- \(sys\) (I/O or LTI system) – System to be checked
- \(strict\) (bool (default = False)) – If strict is True, make sure that timebase is not None
3.11.8 control.issiso

control.issiso(sys, strict=False)

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

Parameters

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

3.11.9 control.issys

control.issys(obj)

Return True if an object is a system, otherwise False

3.11.10 control.mag2db

control.mag2db(mag)

Convert a magnitude to decibels (dB)

If \( A \) is magnitude,

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

Parameters

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

Returns

- **db** – corresponding values in decibels

Return type

float or ndarray

3.11.11 control.modal_form

control.modal_form(xsys, condmax=None, sort=False)

Convert a system into modal canonical form

Parameters

- **xs``ys** (StateSpace object) – System to be transformed, with state \( x \)
- **condmax** (None or float, optional) – An upper bound on individual transformations. If None, use bdschur default.
- **sort** (bool, optional) – If False (default), Schur blocks will not be sorted. See bdschur for sort order.

Returns

- **zsys** (StateSpace object) – System in modal canonical form, with state \( z \)
- **T** ((M, M) ndarray) – Coordinate transformation: \( z = T \times x \)
3.11.12 control.observable_form

countrol.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 ((M, M) real ndarray) – Coordinate transformation: z = T * x

3.11.13 control.pade

countrol.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

3.11.14 control.reachable_form

countrol.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 ((M, M) real ndarray) – Coordinate transformation: z = T * x
3.11.15 control.sample_system

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

Convert a continuous time system to discrete time by sampling

Parameters

- `sysc` (LTI (StateSpace or TransferFunction)) – Continuous time system to be converted
- `Ts` (float > 0) – Sampling period
- `method` (string) – Method to use for conversion, e.g. ‘bilinear’, ‘zoh’ (default)
- `alpha` (float within [0, 1]) – The generalized bilinear transformation weighting parameter, which should only be specified with method=”gbt”, and is ignored otherwise. See scipy.signal.cont2discrete().
- `prewarp_frequency` (float within [0, infinity)) – The frequency [rad/s] at which to match with the input continuous-time system’s magnitude and phase (only valid for method='bilinear')

Returns

`sysd` – Discrete time system, with sampling rate Ts

Return type

linsys

Notes

See StateSpace.sample() or TransferFunction.sample() for further details.

Examples

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

3.11.16 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 from state space into transfer function form. Always creates a new system.
- `ss2tf(A, B, C, D)`
  Create a transfer function system from the matrices of its state and output equations.
  For details see: `tf()`

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

• **inputs** *(str, or list of str, optional)* – List of strings that name the individual signals of the transformed system. If not given, the inputs and outputs are the same as the original system.

• **outputs** *(str, or list of str, optional)* – List of strings that name the individual signals of the transformed system. If not given, the inputs and outputs are the same as the original system.

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

**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**

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

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

### 3.11.17 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**

dict
3.11.18 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 space space form. Always creates a new system, even if sys is already a StateSpace object.

tf2ss(num, den)
Create a state space 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
• inputs (str, or list of str, optional) – List of strings that name the individual signals of the transformed system. If not given, the inputs and outputs are the same as the original system.
• outputs (str, or list of str, optional) – List of strings that name the individual signals of the transformed system. If not given, the inputs and outputs are the same as the original system.
• name (string, optional) – System name. If unspecified, a generic name <sys[id]> is generated with a unique integer id.

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

```python
>>> num = [[[1., 2.], [3., 4.]], [[5., 6.], [7., 8.]]]
>>> den = [[[9., 8., 7.], [6., 5., 4.]], [[3., 2., 1.], [-1., -2., -3.]]]
>>> sys1 = tf2ss(num, den)
>>> sys_tf = tf(num, den)
>>> sys2 = tf2ss(sys_tf)
```

### 3.11.19 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

### 3.11.20 control.timebase

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

Return the timebase for a system

```python
dt = timebase(sys)
```

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

### 3.11.21 control.timebaseEqual

`control.timebaseEqual(sys1, sys2)`

Check to see if two systems have the same timebase

```python
timebaseEqual(sys1, sys2)
```

returns True if the timebases for the two systems are compatible. By default, systems with timebase ‘None’ are compatible with either discrete or continuous timebase systems. If two systems have a discrete timebase (`dt > 0`) then their timebases must be equal.
3.11.22 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 input/output systems (both linear time-invariant and nonlinear). They are usually created from factory functions such as \( \text{tf()} \) and \( \text{ss()} \), so the user should normally not need to instantiate these directly.

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>StateSpace(A, B, C, D[, dt])</code></td>
<td>A class for representing state-space models.</td>
</tr>
<tr>
<td><code>TransferFunction(num, den[, dt])</code></td>
<td>A class for representing transfer functions.</td>
</tr>
<tr>
<td><code>InputOutputSystem([params])</code></td>
<td>A class for representing input/output systems.</td>
</tr>
<tr>
<td><code>FrequencyResponseData(d, w[, smooth])</code></td>
<td>A class for models defined by frequency response data (FRD).</td>
</tr>
<tr>
<td><code>TimeResponseData(time, outputs[, states, ...])</code></td>
<td>A class for returning time responses.</td>
</tr>
</tbody>
</table>

### 4.1 control.StateSpace

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

Bases: LTI

A class for representing state-space models.

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.

**Parameters**

- \( A \) (array_like) – System matrices of the appropriate dimensions.
- \( B \) (array_like) – System matrices of the appropriate dimensions.
- \( C \) (array_like) – System matrices of the appropriate dimensions.
- \( D \) (array_like) – System matrices of the appropriate dimensions.
- \( dt \) (None, True or float, optional) – System timebase. 0 (default) indicates continuous time, True indicates discrete time with unspecified sampling time, positive number is discrete time with specified sampling time, None indicates unspecified timebase (either continuous or discrete time).

\( \text{ninputs, noutputs, nstates} \)

Number of input, output and state variables.
Type
int

A, B, C, D
System matrices defining the input/output dynamics.

Type
2D arrays

dt
System timebase. 0 (default) indicates continuous time, True indicates discrete time with unspecified sampling time, positive number is discrete time with specified sampling time, None indicates unspecified time-base (either continuous or discrete time).

Type
None, True or float

Notes
The main data members in the StateSpace class 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.

A discrete time system is created by specifying a nonzero ‘timebase’, dt when the system is constructed:

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

Systems must have compatible timebases in order to be combined. 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. Similarly, a system with timebase None can be combined with a system having any timebase; the result will have the timebase of the latter system. The default value of dt can be changed by changing the value of control.config.defaults['control.default_dt'].

Note: timebase processing has moved to namedio.

A state space system is callable and returns the value of the transfer function evaluated at a point in the complex plane. See __call__() for a more detailed description.

StateSpace instances have support for IPython LaTeX output, intended for pretty-printing in Jupyter notebooks. The LaTeX output can be configured using control.config.defaults['statesp.latex_num_format'] and control.config.defaults['statesp.latex_repr_type']. The LaTeX output is tailored for MathJax, as used in Jupyter, and may look odd when typeset by non-MathJax LaTeX systems.

control.config.defaults['statesp.latex_num_format'] is a format string fragment, specifically the part of the format string after ‘:’ used to convert floating-point numbers to strings. By default it is ‘.3g’.

control.config.defaults['statesp.latex_repr_type'] must either be ‘partitioned’ or ‘separate’. If ‘partitioned’, the A, B, C, D matrices are shown as a single, partitioned matrix; if ‘separate’, the matrices are shown separately.
**Methods**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>append</code></td>
<td>Append a second model to the present model.</td>
</tr>
<tr>
<td><code>copy</code></td>
<td>Make a copy of an input/output system.</td>
</tr>
<tr>
<td><code>damp</code></td>
<td>Natural frequency, damping ratio of system poles.</td>
</tr>
<tr>
<td><code>dcgain</code></td>
<td>Return the zero-frequency gain.</td>
</tr>
<tr>
<td><code>dynamics</code></td>
<td>Compute the dynamics of the system.</td>
</tr>
<tr>
<td><code>feedback</code></td>
<td>Feedback interconnection between two LTI systems.</td>
</tr>
<tr>
<td><code>find_input</code></td>
<td>Find the index for an input given its name (None if not found).</td>
</tr>
<tr>
<td><code>find_output</code></td>
<td>Find the index for an output given its name (None if not found).</td>
</tr>
<tr>
<td><code>find_state</code></td>
<td>Find the index for a state given its name (None if not found).</td>
</tr>
<tr>
<td><code>freqresp</code></td>
<td>(deprecated) Evaluate transfer function at complex frequencies.</td>
</tr>
<tr>
<td><code>frequency_response</code></td>
<td>Evaluate the linear time-invariant system at an array of angular frequencies.</td>
</tr>
<tr>
<td><code>horner</code></td>
<td>Evaluate system's transfer function at complex frequency using Laub's or Horner's method.</td>
</tr>
<tr>
<td><code>isctime</code></td>
<td>Check to see if a system is a continuous-time system.</td>
</tr>
<tr>
<td><code>isdtime</code></td>
<td>Check to see if a system is a discrete-time system.</td>
</tr>
<tr>
<td><code>issiso</code></td>
<td>Check to see if a system is single input, single output.</td>
</tr>
<tr>
<td><code>lft</code></td>
<td>Return the Linear Fractional Transformation.</td>
</tr>
<tr>
<td><code>minreal</code></td>
<td>Calculate a minimal realization, removes unobservable and uncontrollable states</td>
</tr>
<tr>
<td><code>output</code></td>
<td>Compute the output of the system.</td>
</tr>
<tr>
<td><code>pole</code></td>
<td></td>
</tr>
<tr>
<td><code>poles</code></td>
<td>Compute the poles of a state space system.</td>
</tr>
<tr>
<td><code>returnScipySignalLTI</code></td>
<td>Return a list of a list of scipy.signal.lti objects.</td>
</tr>
<tr>
<td><code>sample</code></td>
<td>Convert a continuous time system to discrete time.</td>
</tr>
<tr>
<td><code>set_inputs</code></td>
<td>Set the number/names of the system inputs.</td>
</tr>
<tr>
<td><code>set_outputs</code></td>
<td>Set the number/names of the system outputs.</td>
</tr>
<tr>
<td><code>set_states</code></td>
<td>Set the number/names of the system states.</td>
</tr>
<tr>
<td><code>slycot_laub</code></td>
<td>Evaluate system's transfer function at complex frequency using Laub's method from Slycot.</td>
</tr>
<tr>
<td><code>zero</code></td>
<td></td>
</tr>
<tr>
<td><code>zeros</code></td>
<td>Compute the zeros of a state space system.</td>
</tr>
</tbody>
</table>

A

Dynamics matrix.

B

Input matrix.

C

Output matrix.

4.1. `control.StateSpace` 89
D

Direct term.

__add__(other)

Add two LTI systems (parallel connection).

__call__(x, squeeze=None, warn_infinite=True)

Evaluate system's transfer function at complex frequency.

Returns the complex frequency response $sys(x)$ where $x$ is $s$ for continuous-time systems and $z$ for discrete-time systems.

To evaluate at a frequency $\omega$ in radians per second, enter $x = \omega * 1j$, for continuous-time systems, or $x = \exp(1j * \omega * dt)$ for discrete-time systems. Or use `StateSpace.frequency_response()`.

Parameters

- **x** (complex or complex 1D array_like) – Complex frequencies
- **squeeze** (bool, optional) – If squeeze=True, remove single-dimensional entries from the shape of the output even if the system is not SISO. If squeeze=False, keep all indices (output, input and, if omega is array_like, frequency) even if the system is SISO. The default value can be set using config.defaults['control.squeeze_frequency_response'].
- **warn_infinite** (bool, optional) – If set to False, don’t warn if frequency response is infinite.

Returns

- **fresp** – The frequency response of the system. If the system is SISO and squeeze is not True, the shape of the array matches the shape of omega. If the system is not SISO or squeeze is False, the first two dimensions of the array are indices for the output and input and the remaining dimensions match omega. If squeeze is True then single-dimensional axes are removed.

Return type

complex ndarray

__div__(other)

Divide two LTI systems.

__getitem__(indices)

Array style access

__mul__(other)

Multiply two LTI objects (serial connection).

__neg__()

Negate a state space system.

__radd__(other)

Right add two LTI systems (parallel connection).

__rdiv__(other)

Right divide two LTI systems.
__rmul__(other)
Right multiply two LTI objects (serial connection).

__rsub__(other)
Right subtract two LTI systems.

__sub__(other)
Subtract two LTI systems.

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(name=None, use_prefix_suffix=True)
Make a copy of an input/output system

A copy of the system is made, with a new name. The name keyword can be used to specify a specific name for the system. If no name is given and use_prefix_suffix is True, the name is constructed by prepending config.defaults['iosys.duplicate_system_name_prefix'] and appending config.defaults['iosys.duplicate_system_name_suffix']. Otherwise, a generic system name of the form sys[<id>] is used, where <id> is based on an internal counter.

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(warn_infinite=False)
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:

Parameters

- warn_infinite (bool, optional) – By default, don’t issue a warning message if the zero-frequency gain is infinite. Setting warn_infinite to generate the warning message.

Returns

- gain – Array or scalar value for SISO systems, depending on config.defaults['control.squeeze_frequency_response']. The value of the array elements or the scalar is either the zero-frequency (or DC) gain, or inf, if the frequency response is singular.

For real valued systems, the empty imaginary part of the complex zero-frequency response is discarded and a real array or scalar is returned.

Return type

(noutputs, ninputs) ndarray or scalar

4.1. control.StateSpace
dynamics(t, x, u=None)
Compute the dynamics of the system
Given input u and state x, returns the dynamics of the state-space system. If the system is continuous, returns the time derivative dx/dt
\[
dx/dt = A \ x + B \ u
\]
where A and B are the state-space matrices of the system. If the system is discrete-time, returns the next value of x:
\[
x[t+dt] = A \ x[t] + B \ u[t]
\]
The inputs x and u must be of the correct length for the system.
The first argument t is ignored because StateSpace systems are time-invariant. It is included so that the dynamics can be passed to numerical integrators, such as scipy.integrate.solve_ivp() and for consistency with IOSystem systems.

Parameters
- **t** (*float* (ignored)) – time
- **x** (*array_like*) – current state
- **u** (*array_like* (optional)) – input, zero if omitted

Returns
- dx/dt or x[t+dt]

Return type
- ndarray

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

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

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

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

freqresp(omega)
(deprecated) Evaluate transfer function at complex frequencies.

frequency_response(omega, squeeze=None)
Evaluate the linear time-invariant system at an array of angular frequencies.
Reports the frequency response of the system,
\[
G(j*omega) = mag * exp(j*phase)
\]
for continuous time systems. For discrete time systems, the response is evaluated around the unit circle such that
\[
G(exp(j*omega*dt)) = mag * exp(j*phase).
\]
In general the system may be multiple input, multiple output (MIMO), where \(m = \text{self.ninputs}\) number of inputs and \(p = \text{self.noutputs}\) number of outputs.

Parameters
• **omega** *(float or 1D array_like)* – A list, tuple, array, or scalar value of frequencies in radians/sec at which the system will be evaluated.

• **squeeze** *(bool, optional)* – If squeeze=True, remove single-dimensional entries from the shape of the output even if the system is not SISO. If squeeze=False, keep all indices (output, input and, if omega is array_like, frequency) even if the system is SISO. The default value can be set using config.defaults['control.squeeze_frequency_response'].

**Returns**

**response** – Frequency response data object representing the frequency response. This object can be assigned to a tuple using

```python
mag, phase, omega = response
```

where **mag** is the magnitude (absolute value, not dB or log10) of the system frequency response, **phase** is the wrapped phase in radians of the system frequency response, and **omega** is the (sorted) frequencies at which the response was evaluated. If the system is SISO and squeeze is not True, **magnitude** and **phase** are 1D, indexed by frequency. If the system is not SISO or squeeze is False, the array is 3D, indexed by the output, input, and frequency. If squeeze is True then single-dimensional axes are removed.

**Return type**

FrequencyResponseData

**horner** *(x, warn_infinite=True)*

Evaluate system’s transfer function at complex frequency using Laub’s or Horner’s method.

Evaluates *sys*(x) where x is *s* for continuous-time systems and *z* for discrete-time systems.

Expects inputs and outputs to be formatted correctly. Use *sys*(x) for a more user-friendly interface.

**Parameters**

**x** *(complex array_like or complex)* – Complex frequencies

**Returns**

**output** – Frequency response

**Return type**

*(self.noutputs, self.ninputs, len(x)) complex ndarray*

**Notes**

Attempts to use Laub’s method from Slycot library, with a fall-back to python code.

**property inputs**

 Deprecated attribute; use **ninputs** instead.

The **inputs** attribute was used to store the number of system inputs. It is no longer used. If you need access to the number of inputs for an LTI system, use **ninputs**.

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

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

**Parameters**

• **sys** *(Named I/O system)* – System to be checked

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

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

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

Calculate a minimal realization, removes unobservable and uncontrollable states

ninputs

Number of system inputs.

noutputs

Number of system outputs.

nstates

Number of system states.

output

Compute the output of the system

Given input \( u \) and state \( x \), returns the output \( y \) of the state-space system:

\[
 y = C x + D u
\]

where \( A \) and \( B \) are the state-space matrices of the system.

The first argument \( t \) is ignored because \( StateSpace \) systems are time-invariant. It is included so that the dynamics can be passed to most numerical integrators, such as scipy’s `integrate.solve_ivp` and for consistency with \( IOSystem \) systems.

The inputs \( x \) and \( u \) must be of the correct length for the system.

Parameters

\( t \) (float (ignored)) – time

\( x \) (array_like) – current state
• `u (array_like (optional))` – input (zero if omitted)

Returns

`y`

Return type

`ndarray`

**property outputs**

Deprecated attribute; use `noutputs` instead.

The `outputs` attribute was used to store the number of system outputs. It is no longer used. If you need access to the number of outputs for an LTI system, use `noutputs`.

**poles()**

Compute the poles of a state space system.

**returnScipySignalLTI(strict=True)**

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

For instance,

```python
>>> out = ssobject.returnScipySignalLTI()
```

```python
>>> out[3][5]
```

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

**Parameters**

- `strict (bool, optional)` –
  
  **True** (default):
  The timebase `ssobject.dt` cannot be None; it must be continuous (0) or discrete (True or > 0).

  **False:**
  If `ssobject.dt` is None, continuous time `scipy.signal.lti` objects are returned.

**Returns**

- `out` – continuous time (inheriting from `scipy.signal.lti`) or discrete time (inheriting from `scipy.signal.dlti`) SISO objects

**Return type**

list of list of `scipy.signal.StateSpace`

**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 $[0, \infty)$) – 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 $T_s$

Return type

*StateSpace*

Notes

Uses `scipy.signal.cont2discrete()`

Examples

```python
>>> 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].

`slycot_laub(x)`

Evaluate system’s transfer function at complex frequency using Laub’s method from Slycot.

Expects inputs and outputs to be formatted correctly. Use `sys(x)` for a more user-friendly interface.

**Parameters**

- **x** *(complex array_like or complex)* – Complex frequency

**Returns**

- **output** – Frequency response

**Return type**

- (number_outputs, number_inputs, len(x)) complex ndarray

**property states**

Deprecated attribute; use *nstates* instead.

The *state* attribute was used to store the number of states for a state space system. It is no longer used.

If you need to access the number of states, use *nstates*.

`zeros()`

Compute the zeros of a state space system.

### 4.2 control.TransferFunction

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

**Bases:** LTI

A class for representing transfer functions.

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

**Parameters**

- **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

- **dt** *(None, True or float, optional)* – System timebase. 0 (default) indicates continuous time, True indicates discrete time with unspecified sampling time, positive number is discrete time with specified sampling time, None indicates unspecified timebase (either continuous or discrete time).

**ninputs, noutputs, nstates**

Number of input, output and state variables.

**Type**

- int
num, den
Polynomial coefficients of the numerator and denominator.

Type
2D list of array

dt
System timebase. 0 (default) indicates continuous time, True indicates discrete time with unspecified sampling time, positive number is discrete time with specified sampling time, None indicates unspecified timebase (either continuous or discrete time).

Type
None, True or float

Notes
The attributes ‘num’ and ‘den’ are 2-D lists of arrays containing MIMO numerator and denominator coefficients. For example,

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

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

A discrete time transfer function is created by specifying a nonzero ‘timebase’ dt when the system is constructed:

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

Systems must have compatible timebases in order to be combined. 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. Similarly, a system with timebase None can be combined with a system having any timebase; the result will have the timebase of the latter system. The default value of dt can be changed by changing the value of `control.config.defaults['control.default_dt']`.

A transfer function is callable and returns the value of the transfer function evaluated at a point in the complex plane. See `__call__()` for a more detailed description.

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,

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

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>copy</td>
<td>Make a copy of an input/output system</td>
</tr>
<tr>
<td>damp</td>
<td>Natural frequency, damping ratio of system poles</td>
</tr>
<tr>
<td>dcmgain</td>
<td>Return the zero-frequency (or DC) gain</td>
</tr>
<tr>
<td>feedback</td>
<td>Feedback interconnection between two LTI objects</td>
</tr>
<tr>
<td>find_input</td>
<td>Find the index for an input given its name (None if not found)</td>
</tr>
<tr>
<td>find_output</td>
<td>Find the index for an output given its name (None if not found)</td>
</tr>
<tr>
<td>find_state</td>
<td>Find the index for a state given its name (None if not found)</td>
</tr>
<tr>
<td>freqresp</td>
<td>(deprecated) Evaluate transfer function at complex frequencies.</td>
</tr>
<tr>
<td>frequency_response</td>
<td>Evaluate the linear time-invariant system at an array of angular frequencies.</td>
</tr>
<tr>
<td>horner</td>
<td>Evaluate system's transfer function at complex frequency using Horner's method.</td>
</tr>
<tr>
<td>isctime</td>
<td>Check to see if a system is a continuous-time system</td>
</tr>
<tr>
<td>isdtime</td>
<td>Check to see if a system is a discrete-time system</td>
</tr>
<tr>
<td>issiso</td>
<td>Check to see if a system is single input, single output</td>
</tr>
<tr>
<td>minreal</td>
<td>Remove cancelling pole/zero pairs from a transfer function</td>
</tr>
<tr>
<td>poles</td>
<td>Compute the poles of a transfer function.</td>
</tr>
<tr>
<td>returnScipySignalLTI</td>
<td>Return a list of a list of scipy.signal.lti objects.</td>
</tr>
<tr>
<td>sample</td>
<td>Convert a continuous-time system to discrete time</td>
</tr>
<tr>
<td>set_inputs</td>
<td>Set the number/names of the system inputs.</td>
</tr>
<tr>
<td>set_outputs</td>
<td>Set the number/names of the system outputs.</td>
</tr>
<tr>
<td>set_states</td>
<td>Set the number/names of the system states.</td>
</tr>
<tr>
<td>zero</td>
<td>Compute the zeros of a transfer function.</td>
</tr>
</tbody>
</table>

__add__(other)  
Add two LTI objects (parallel connection).  

__call__(x, squeeze=None, warn_infinite=True)  
Evaluate system’s transfer function at complex frequencies.  
Returns the complex frequency response $y(s)$ where $s$ is $s$ for continuous-time systems and $z$ for discrete-time systems.  
In general the system may be multiple input, multiple output (MIMO), where $m = self.ninputs$ number of inputs and $p = self.noutputs$ number of outputs.  
To evaluate at a frequency $omega$ in radians per second, enter $x = omega * 1j$, for continuous-time systems, or $x = exp(1j * omega * dt)$ for discrete-time systems. Or use TransferFunction. frequency_response().

Parameters

- $x$ (complex or complex 1D array_like) – Complex frequencies
• **squeeze** (*bool*, *optional*) – If `squeeze=True`, remove single-dimensional entries from the shape of the output even if the system is not SISO. If `squeeze=False`, keep all indices (output, input and, if `omega` is `array_like`, frequency) even if the system is SISO. The default value can be set using `config.defaults['control.squeeze_frequency_response']`. If `True` and the system is single-input single-output (SISO), return a 1D array rather than a 3D array. Default value (True) set by `config.defaults['control.squeeze_frequency_response']`.

• **warn_infinite** (*bool*, *optional*) – If set to `False`, turn off divide by zero warning.

**Returns**

- `fresp` – The frequency response of the system. If the system is SISO and `squeeze` is not `True`, the shape of the array matches the shape of `omega`. If the system is not SISO or `squeeze` is `False`, the first two dimensions of the array are indices for the output and input and the remaining dimensions match `omega`. If `squeeze` is `True` then single-dimensional axes are removed.

**Return type**

`complex ndarray`  

- `__mul__`(other)  
  Multiply two LTI objects (serial connection).

- `__neg__`()  
  Negate a transfer function.

- `__radd__`(other)  
  Right add two LTI objects (parallel connection).

- `__rmul__`(other)  
  Right multiply two LTI objects (serial connection).

- `__rsub__`(other)  
  Right subtract two LTI objects.

- `__rtruediv__`(other)  
  Right divide two LTI objects.

- `__sub__`(other)  
  Subtract two LTI objects.

- `__truediv__`(other)  
  Divide two LTI objects.

- `copy`(name=None, use_prefix_suffix=True)  
  Make a copy of an input/output system

A copy of the system is made, with a new name. The `name` keyword can be used to specify a specific name for the system. If no name is given and `use_prefix_suffix` is `True`, the name is constructed by prepending `config.defaults['iosys.duplicate_system_name_prefix']` and appending `config.defaults['iosys.duplicate_system_name_suffix']`. Otherwise, a generic system name of the form `sys[<id>]` is used, where `<id>` is based on an internal counter.

- `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** *(warn_infinite=False)*  
Return the zero-frequency (or DC) gain  
For a continuous-time transfer function \( G(s) \), the DC gain is \( G(0) \) For a discrete-time transfer function \( G(z) \), the DC gain is \( G(1) \)

**Parameters**

- **warn_infinite** (bool, optional) – By default, don’t issue a warning message if the zero-frequency gain is infinite. Setting `warn_infinite` to generate the warning message.

**Returns**

- **gain** – Array or scalar value for SISO systems, depending on config.defaults['control.squeeze_frequency_response']. The value of the array elements or the scalar is either the zero-frequency (or DC) gain, or `inf`, if the frequency response is singular.

For real valued systems, the empty imaginary part of the complex zero-frequency response is discarded and a real array or scalar is returned.

**Return type**

(noutputs, ninputs) ndarray or scalar

**den**

Transfer function denominator polynomial (array)

The numerator of the transfer function is store as an 2D list of arrays containing MIMO numerator coefficients, indexed by outputs and inputs. For example, `den[2][5]` is the array of coefficients for the denominator of the transfer function from the sixth input to the third output.

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

**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)*  
(deprecated) Evaluate transfer function at complex frequencies.

**frequency_response** *(omega, squeeze=None)*  
Evaluate the linear time-invariant system at an array of angular frequencies.

Reports the frequency response of the system,

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

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

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

In general the system may be multiple input, multiple output (MIMO), where \( m = self.ninputs \) number of inputs and \( p = self.noutputs \) number of outputs.
Parameters

- **omega** *(float or 1D array_like)* – A list, tuple, array, or scalar value of frequencies in radians/sec at which the system will be evaluated.

- **squeeze** *(bool, optional)* – If squeeze=True, remove single-dimensional entries from the shape of the output even if the system is not SISO. If squeeze=False, keep all indices (output, input and, if omega is array_like, frequency) even if the system is SISO. The default value can be set using config.defaults['control.squeeze_frequency_response'].

Returns

- **response** – Frequency response data object representing the frequency response. This object can be assigned to a tuple using

  ```python
  mag, phase, omega = response
  ```

  where mag is the magnitude (absolute value, not dB or log10) of the system frequency response, phase is the wrapped phase in radians of the system frequency response, and omega is the (sorted) frequencies at which the response was evaluated. If the system is SISO and squeeze is not True, magnitude and phase are 1D, indexed by frequency. If the system is not SISO or squeeze is False, the array is 3D, indexed by the output, input, and frequency. If squeeze is True then single-dimensional axes are removed.

Return type

FrequencyResponseData

**horner** *(x, warn_infinite=True)*

Evaluate system’s transfer function at complex frequency using Horner’s method.

Evaluates $sys(x)$ where $x$ is $s$ for continuous-time systems and $z$ for discrete-time systems.

Expects inputs and outputs to be formatted correctly. Use $sys(x)$ for a more user-friendly interface.

Parameters

- **x** *(complex array_like or complex scalar)* – Complex frequencies

Returns

- **output** – Frequency response

Return type

*(self.noutputs, self.ninputs, len(x)) complex ndarray*

**property inputs**

Deprecated attribute; use **ninputs** instead.

The inputs attribute was used to store the number of system inputs. It is no longer used. If you need access to the number of inputs for an LTI system, use **ninputs**.

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

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

Parameters

- **sys** *(Named I/O 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

ninputs
Number of system inputs.

noutputs
Number of system outputs.

nstates
Number of system states.
	num
Transfer function numerator polynomial (array)
The numerator of the transfer function is stored as an 2D list of arrays containing MIMO numerator coefficients, indexed by outputs and inputs. For example, num[2][5] is the array of coefficients for the numerator of the transfer function from the sixth input to the third output.

property outputs
Deprecated attribute; use noutputs instead.
The outputs attribute was used to store the number of system outputs. It is no longer used. If you need access to the number of outputs for an LTI system, use noutputs.

poles()
Compute the poles of a transfer function.

returnScipySignalLTI(strict=True)
Return a list of a list of scipy.signal.lti objects.
For instance,

```python
>>> out = tfobject.returnScipySignalLTI()
>>> out[3][5]
```
is a scipy.signal.lti object corresponding to the transfer function from the 6th input to the 4th output.

Parameters

strict (bool, optional) –

True (default):
The timebase tfobject.dt cannot be None; it must be continuous (0) or discrete (True or > 0).

False:
if tfobject.dt is None, continuous time scipy.signal.lti objects are returned
Returns

out – continuous time (inheriting from `scipy.signal.lti`) or discrete time (inheriting from `scipy.signal.dlti`) SISO objects

Return type

list of list of `scipy.signal.TransferFunction`

`s`

Differentiation operator (continuous time)

The `s` constant can be used to create continuous time transfer functions using algebraic expressions.

Example

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

`sampling` *(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. See `scipy.signal.cont2discrete()`.*
- **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 sample period Ts

Return type

TransferFunction system
Notes

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

Examples

```python
>>> sys = TransferFunction(1, [1,1])
>>> 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[$i$].

`z`

Delay operator (discrete time)

The $z$ constant can be used to create discrete time transfer functions using algebraic expressions.
Example

```python
>>> z = TransferFunction.z
>>> G = 2 * z / (4 * z**3 + 3*z - 1)
```

`zeros()`
Compute the zeros of a transfer function.

4.3 control.InputOutputSystem

class control.InputOutputSystem(params={}, **kwargs)

Bases: NamedIOSystem

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 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. 0 (default) indicates continuous time, True indicates discrete time with unspecified sampling time, positive number is discrete time with specified sampling time, None indicates unspecified timebase (either continuous or discrete time).
- **name** (string, optional) – System name (used for specifying signals). If unspecified, a generic name <sys[id]> is generated with a unique integer id.
- **params** (dict, optional) – Parameter values for the systems. Passed to the evaluation functions for the system as default values, overriding internal defaults.

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. 0 (default) indicates continuous time, True indicates discrete time with unspecified sampling time, positive number is discrete time with specified sampling time, None indicates unspecified timebase (either continuous or discrete 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 InputOutputSystem 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.

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>copy</td>
<td>Make a copy of an input/output system</td>
</tr>
<tr>
<td>dynamics</td>
<td>Compute the dynamics of a differential or difference equation.</td>
</tr>
<tr>
<td>feedback</td>
<td>Feedback interconnection between two input/output systems</td>
</tr>
<tr>
<td>find_input</td>
<td>Find the index for an input given its name (None if not found)</td>
</tr>
<tr>
<td>find_output</td>
<td>Find the index for an output given its name (None if not found)</td>
</tr>
<tr>
<td>find_state</td>
<td>Find the index for a state given its name (None if not found)</td>
</tr>
<tr>
<td>isctime</td>
<td>Check to see if a system is a continuous-time system</td>
</tr>
<tr>
<td>isdtime</td>
<td>Check to see if a system is a discrete-time system</td>
</tr>
<tr>
<td>issiso</td>
<td>Check to see if a system is single input, single output</td>
</tr>
<tr>
<td>linearize</td>
<td>Linearize an input/output system at a given state and input.</td>
</tr>
<tr>
<td>output</td>
<td>Compute the output of the system</td>
</tr>
<tr>
<td>set_inputs</td>
<td>Set the number/names of the system inputs.</td>
</tr>
<tr>
<td>set_outputs</td>
<td>Set the number/names of the system outputs.</td>
</tr>
<tr>
<td>set_states</td>
<td>Set the number/names of the system states.</td>
</tr>
</tbody>
</table>
__add__(sys2)
Add two input/output systems (parallel interconnection)

__mul__(sys1)
Multiply two input/output systems (series interconnection)

__neg__()
Negate an input/output systems (rescale)

__radd__(sys2)
Parallel addition of input/output system to a compatible object.

__rmul__(sys2)
Pre-multiply an input/output systems by a scalar/matrix

__rsub__(sys2)
Parallel subtraction of I/O system to a compatible object.

__sub__(sys2)
Subtract two input/output systems (parallel interconnection)

copy(name=None, use_prefix_suffix=True)
Make a copy of an input/output system

A copy of the system is made, with a new name. The name keyword can be used to specify a specific name for the system. If no name is given and use_prefix_suffix is True, the name is constructed by prepending config.defaults['iosys.duplicate_system_name_prefix'] and appending config.defaults['iosys.duplicate_system_name_suffix']. Otherwise, a generic system name of the form sys[<id>] is used, where <id> is based on an internal counter.

dynamics(t, x, u)
Compute the dynamics of a differential or difference equation.

Given time t, input u and state x, returns the value of the right hand side of the dynamical system. If the system is continuous, returns the time derivative

\[ \frac{dx}{dt} = f(t, x, u) \]

where \( f \) is the system’s (possibly nonlinear) dynamics function. If the system is discrete-time, returns the next value of \( x \):

\[ x[t+dt] = f(t, x[t], u[t]) \]

Where \( t \) is a scalar.

The inputs \( x \) and \( u \) must be of the correct length.

Parameters

- \( t \) (float) – the time at which to evaluate
- \( x \) (array_like) – current state
- \( u \) (array_like) – input

Returns

\( \frac{dx}{dt} \) or \( x[t+dt] \)

Return type

ndarray
feedback(\texttt{other=1, sign=-1, params={}})

Feedback interconnection between two input/output systems

**Parameters**

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

**Returns**

\texttt{out}

**Return type**

\texttt{InputOutputSystem}

**Raises**

\texttt{ValueError} – if the inputs, outputs, or timebases of the systems are incompatible.

\texttt{find_input(name)}

Find the index for an input given its name (\texttt{None} if not found)

\texttt{find_output(name)}

Find the index for an output given its name (\texttt{None} if not found)

\texttt{find_state(name)}

Find the index for a state given its name (\texttt{None} if not found)

\texttt{isctime(strict=False)}

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

**Parameters**

- \texttt{sys (Named I/O system)} – System to be checked
- \texttt{strict (bool, optional)} – If strict is True, make sure that timebase is not None. Default is False.

\texttt{isdtime(strict=False)}

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

**Parameters**

\texttt{strict (bool, optional)} – If strict is True, make sure that timebase is not None. Default is False.

\texttt{issiso()}  

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

\texttt{linearize(x0, u0, t=0, params={}, eps=1e-06, name=None, copy=False, **kwargs)}

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 \texttt{linearize()} for complete documentation.

\texttt{ninputs}

Number of system inputs.
noutputs
Number of system outputs.

nstates
Number of system states.

output(t, x, u)
Compute the output of the system
Given time t, input u and state x, returns the output of the system:
\[ y = g(t, x, u) \]
The inputs x and u must be of the correct length.

Parameters
- t (float) – the time at which to evaluate
- x (array_like) – current state
- u (array_like) – input

Returns
y

Return type
ndarray

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 $prefix[i]$.

## 4.4 control.FrequencyResponseData

class control.FrequencyResponseData(d, w[, smooth ])

Bases: LTI

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

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

**Parameters**

• **d** (*1D or 3D complex array_like*) – The frequency response at each frequency point. If 1D, the system is assumed to be SISO. If 3D, the system is MIMO, 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.

• **w** (*iterable of real frequencies*) – List of frequency points for which data are available.

• **smooth** (*bool, optional*) – If True, create an interpolation function that allows the frequency response to be computed at any frequency within the range of frequencies given in w. If False (default), frequency response can only be obtained at the frequencies specified in w.

**ninputs, noutputs**

Number of input and output variables.

**Type**

int

**omega**

Frequency points of the response.

**Type**

1D array

**fresp**

Frequency response, indexed by output index, input index, and frequency point.

**Type**

3D array
Notes

The main data members are ‘omega’ and ‘fresp’, where ‘omega’ is a 1D array of frequency points and and ‘fresp’ is a 3D array of frequency responses, 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.

A frequency response data object is callable and returns the value of the transfer function evaluated at a point in the complex plane (must be on the imaginary access). See `__call__()` for a more detailed description.

Methods

- `copy` Make a copy of an input/output system
- `damp` Natural frequency, damping ratio of system poles
- `dcgain` Return the zero-frequency gain
- `eval` Evaluate a transfer function at angular frequency omega.
- `feedback` Feedback interconnection between two FRD objects.
- `find_input` Find the index for an input given its name (None if not found)
- `find_output` Find the index for an output given its name (None if not found)
- `find_state` Find the index for a state given its name (None if not found)
- `freqresp` (deprecated) Evaluate transfer function at complex frequencies.
- `frequency_response` Evaluate the linear time-invariant system at an array of angular frequencies.
- `isctime` Check to see if a system is a continuous-time system
- `isdtime` Check to see if a system is a discrete-time system
- `issiso` Check to see if a system is single input, single output
- `pole`
- `set_inputs` Set the number/names of the system inputs.
- `set_outputs` Set the number/names of the system outputs.
- `set_states` Set the number/names of the system states.
- `to_pandas`
- `zero`

`__add__`(other)

Add two LTI objects (parallel connection).

`__call__`(s=None, squeeze=None, return_magphase=None)

Evaluate system’s transfer function at complex frequencies.
Returns the complex frequency response $sys(s)$ of system $sys$ with $m = sys.ninputs$ number of inputs and $p = sys.noutputs$ number of outputs.

To evaluate at a frequency $\omega$ in radians per second, enter $s = \omega * 1j$ or use $sys.eval(\omega)$.

For a frequency response data object, the argument must be an imaginary number (since only the frequency response is defined).

If $s$ is not given, this function creates a copy of a frequency response data object with a different set of output settings.

**Parameters**

- $s$ (complex scalar or 1D array_like) – Complex frequencies. If not specified, return a copy of the frequency response data object with updated settings for output processing (squeeze, return_magphase).
- $squeeze$ (bool, optional) – If squeeze=True, remove single-dimensional entries from the shape of the output even if the system is not SISO. If squeeze=False, keep all indices (output, input and, if omega is array_like, frequency) even if the system is SISO. The default value can be set using config.defaults['control.squeeze_frequency_response'].
- $return\_magphase$ (bool, optional) – If True, then a frequency response data object will enumerate as a tuple of the form (mag, phase, omega) where where mag is the magnitude (absolute value, not dB or log10) of the system frequency response, phase is the wrapped phase in radians of the system frequency response, and omega is the (sorted) frequencies at which the response was evaluated.

**Returns**

$fresp$ – The frequency response of the system. If the system is SISO and squeeze is not True, the shape of the array matches the shape of omega. If the system is not SISO or squeeze is False, the first two dimensions of the array are indices for the output and input and the remaining dimensions match omega. If squeeze is True then single-dimensional axes are removed.

**Return type**

complex ndarray

**Raises**

- ValueError – If $s$ is not purely imaginary, because *FrequencyDomainData* systems are only defined at imaginary frequency values.

__mul__(other)

Multiply two LTI objects (serial connection).

__neg__()

Negate a transfer function.

__radd__(other)

Right add two LTI objects (parallel connection).

__rmul__(other)

Right Multiply two LTI objects (serial connection).

__rsub__(other)

Right subtract two LTI objects.

__rtruediv__(other)

Right divide two LTI objects.
Subtract two LTI objects.

Divide two LTI objects.

Make a copy of an input/output system

A copy of the system is made, with a new name. The name keyword can be used to specify a specific name for the system. If no name is given and use_prefix_suffix is True, the name is constructed by prepending config.defaults['iosys.duplicate_system_name_prefix'] and appending config.defaults['iosys.duplicate_system_name_suffix']. Otherwise, a generic system name of the form sys<id> is used, where <id> is based on an internal counter.

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, squeeze=None)

Evaluate a transfer function at angular 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.

Parameters

- omega (float or 1D array_like) – Frequencies in radians per second
- squeeze (bool, optional) – If squeeze=True, remove single-dimensional entries from the shape of the output even if the system is not SISO. If squeeze=False, keep all indices (output, input and, if omega is array_like, frequency) even if the system is SISO. The default value can be set using config.defaults['control.squeeze_frequency_response'].

Returns

fresp – The frequency response of the system. If the system is SISO and squeeze is not True, the shape of the array matches the shape of omega. If the system is not SISO or squeeze is False, the first two dimensions of the array are indices for the output and input and the remaining dimensions match omega. If squeeze is True then single-dimensional axes are removed.

Return type

complex ndarray

Feedback interconnection between two FRD objects.

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)

defreqresp(omega)
    (deprecated) Evaluate transfer function at complex frequencies.

defrequency_response(omega, squeeze=None)
    Evaluate the linear time-invariant system at an array of angular frequencies.
    Reports the frequency response of the system,
        G(j*omega) = mag * exp(j*phase)
    for continuous time systems. For discrete time systems, the response is evaluated around the unit circle
    such that
        G(exp(j*omega*dt)) = mag * exp(j*phase).
    In general the system may be multiple input, multiple output (MIMO), where
    m = self.ninputs number of
    inputs and p = self.noutputs number of outputs.

    Parameters
    • omega (float or 1D array_like) – A list, tuple, array, or scalar value of frequencies
      in radians/sec at which the system will be evaluated.
    • squeeze (bool, optional) – If squeeze=True, remove single-dimensional entries from
      the shape of the output even if the system is not SISO. If squeeze=False, keep all indices
      (output, input and, if omega is array_like, frequency) even if the system is SISO. The default
      value can be set using config.defaults['control.squeeze_frequency_response'].

    Returns
    response – Frequency response data object representing the frequency response. This object
    can be assigned to a tuple using
        mag, phase, omega = response
    where mag is the magnitude (absolute value, not dB or log10) of the system frequency re-
    sponse, phase is the wrapped phase in radians of the system frequency response, and omega
    is the (sorted) frequencies at which the response was evaluated. If the system is SISO and
    squeeze is not True, magnitude and phase are 1D, indexed by frequency. If the system is
    not SISO or squeeze is False, the array is 3D, indexed by the output, input, and frequency. If
    squeeze is True then single-dimensional axes are removed.

    Return type
    FrequencyResponseData

property inputs
    Deprecated attribute; use ninputs instead.

The inputs attribute was used to store the number of system inputs. It is no longer used. If you need
access to the number of inputs for an LTI system, use ninputs.

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

    Parameters
    • sys (Named I/O 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

**ninputs** *

Number of system inputs.

**noutputs** *

Number of system outputs.

**nstates** *

Number of system states.

**property outputs** *

Deprecated attribute; use **noutputs** instead.

The outputs attribute was used to store the number of system outputs. It is no longer used. If you need access to the number of outputs for an LTI system, use **noutputs**.

**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\((\text{states}, \text{prefix}='x')\)

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 control.TimeResponseData

**class control.TimeResponseData** *(time, outputs, states=None, inputs=None, issiso=None,*

, output_labels=None, state_labels=None, input_labels=None, transpose=False, return_x=False, squeeze=None, multi_trace=False)"

Bases: object

A class for returning time responses.

This class maintains and manipulates the data corresponding to the temporal response of an input/output system. It is used as the return type for time domain simulations (step response, input/output response, etc).

A time response consists of a time vector, an output vector, and optionally an input vector and/or state vector. Inputs and outputs can be 1D (scalar input/output) or 2D (vector input/output).

A time response can be stored for multiple input signals (called traces), with the output and state indexed by the trace number. This allows for input/output response matrices, which is mainly useful for impulse and step responses for linear systems. For multi-trace responses, the same time vector must be used for all traces.

Time responses are accessed through either the raw data, stored as \(t, y, x, u\), or using a set of properties \(\text{time, outputs, states, inputs}\). When accessing time responses via their properties, squeeze processing is applied so that (by default) single-input, single-output systems will have the output and input indices supressed. This behavior is set using the \(\text{squeeze}\) keyword.

#### t

Time values of the input/output response(s). This attribute is normally accessed via the \(\text{time}\) property.

**Type**

1D array

#### y

Output response data, indexed either by output index and time (for single trace responses) or output, trace, and time (for multi-trace responses). These data are normally accessed via the \(\text{outputs}\) property, which performs squeeze processing.

**Type**

2D or 3D array

#### x

State space data, indexed either by output number and time (for single trace responses) or output, trace, and time (for multi-trace responses). If no state data are present, value is \text{None}. These data are normally accessed via the \(\text{states}\) property, which performs squeeze processing.

**Type**

2D or 3D array, or \text{None}
Input signal data, indexed either by input index and time (for single trace responses) or input, trace, and time (for multi-trace responses). If no input data are present, value is `None`. These data are normally accessed via the `inputs` property, which performs squeeze processing.

**Type**

2D or 3D array, or None

**squeeze**

By default, if a system is single-input, single-output (SISO) then the outputs (and inputs) are returned as a 1D array (indexed by time) and if a system is multi-input or multi-output, then the outputs are returned as a 2D array (indexed by output and time) or a 3D array (indexed by output, trace, and time). If `squeeze=True`, access to the output response will remove single-dimensional entries from the shape of the inputs and outputs even if the system is not SISO. If `squeeze=False`, the output is returned as a 2D or 3D array (indexed by the output [if multi-input], trace [if multi-trace] and time) even if the system is SISO. The default value can be set using `config.defaults[\'control.squeeze_time_response\']`.

**Type**

bool, optional

**transpose**

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

**Type**

bool, optional

**issiso**

Set to `True` if the system generating the data is single-input, single-output. If passed as `None` (default), the input data will be used to set the value.

**Type**

bool, optional

**ninputs, noutputs, nstates**

Number of inputs, outputs, and states of the underlying system.

**Type**

int

**input_labels, output_labels, state_labels**

Names for the input, output, and state variables.

**Type**

array of str

**ntraces**

Number of independent traces represented in the input/output response. If `ntraces` is 0 then the data represents a single trace with the trace index surpressed in the data.

**Type**

int
Notes

1. For backward compatibility with earlier versions of python-control, this class has an __iter__ method that allows it to be assigned to a tuple with a variable number of elements. This allows the following patterns to work:

   t, y = step_response(sys) t, x = step_response(sys, return_x=True)

   When using this (legacy) interface, the state vector is not affected by the squeeze parameter.

2. For backward compatibility with earlier versions of python-control, this class has __getitem__ and __len__ methods that allow the return value to be indexed:

   response[0]: returns the time vector response[1]: returns the output vector response[2]: returns the state vector

   When using this (legacy) interface, the state vector is not affected by the squeeze parameter.

3. The default settings for return_x, squeeze and transpose can be changed by calling the class instance and passing new values:

   response(transpose=True).input

   See TimeResponseData.__call__() for more information.

Methods

to_pandas

__call__(**kwargs)

Change value of processing keywords.

Calling the time response object will create a copy of the object and change the values of the keywords used to control the outputs, states, and inputs properties.

Parameters

- squeeze (bool, optional) – If squeeze=True, access to the output response will remove single-dimensional entries from the shape of the inputs, outputs, and states even if the system is not SISO. If squeeze=False, keep the input as a 2D or 3D array (indexed by the input (if multi-input), trace (if single input) and time) and the output and states as a 3D array (indexed by the output/state, trace, and time) even if the system is SISO.

- transpose (bool, optional) – If True, transpose all input and output arrays (for backward compatibility with MATLAB and scipy.signal.lsim()). Default value is False.

- return_x (bool, optional) – If True, return the state vector when enumerating result by assigning to a tuple (default = False).

- input_labels (array of str) – Labels for the inputs, outputs, and states, given as a list of strings matching the appropriate signal dimension.

- output_labels (array of str) – Labels for the inputs, outputs, and states, given as a list of strings matching the appropriate signal dimension.

- state_labels (array of str) – Labels for the inputs, outputs, and states, given as a list of strings matching the appropriate signal dimension.
**Property inputs**

Time response input vector.

Input(s) to the system, indexed by input (optional), trace (optional), and time. If a 1D vector is passed, the input corresponds to a scalar-valued input. If a 2D vector is passed, then it can either represent multiple single-input traces or a single multi-input trace. The optional `multi_trace` keyword should be used to disambiguate the two. If a 3D vector is passed, then it represents a multi-trace, multi-input signal, indexed by input, trace, and time.

See `TimeResponseData.squeeze` for a description of how the dimensions of the input vector can be modified using the `squeeze` keyword.

**Type**

1D or 2D array

**Property outputs**

Time response output vector.

Output response of the system, indexed by either the output and time (if only a single input is given) or the output, trace, and time (for multiple traces). See `TimeResponseData.squeeze` for a description of how this can be modified using the `squeeze` keyword.

**Type**

1D, 2D, or 3D array

**Property states**

Time response state vector.

Time evolution of the state vector, indexed indexed by either the state and time (if only a single trace is given) or the state, trace, and time (for multiple traces). See `TimeResponseData.squeeze` for a description of how this can be modified using the `squeeze` keyword.

**Type**

2D or 3D array

**Property time**

Time vector.

Time values of the input/output response(s).

**Type**

1D array

The following figure illustrates the relationship between the classes and some of the functions that can be used to convert objects from one class to another:
4.6 Input/output system subclasses

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

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>InputOutputSystem</td>
<td>A class for representing input/output systems.</td>
</tr>
<tr>
<td>InterconnectedSystem</td>
<td>Interconnection of a set of input/output systems.</td>
</tr>
<tr>
<td>LinearICSystem</td>
<td>Interconnection of a set of linear input/output systems.</td>
</tr>
<tr>
<td>LinearIOSystem</td>
<td>Input/output representation of a linear (state space) system.</td>
</tr>
<tr>
<td>NonlinearIOSystem</td>
<td>Nonlinear I/O system.</td>
</tr>
</tbody>
</table>

---

4.6. Input/output system subclasses 121
## 4.7 Additional classes

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>DescribingFunctionNonlinearity</code></td>
<td>Base class for nonlinear systems with a describing function.</td>
</tr>
<tr>
<td><code>flatsys.BasisFamily</code></td>
<td>Base class for implementing basis functions for flat systems.</td>
</tr>
<tr>
<td><code>flatsys.FlatSystem</code></td>
<td>Base class for representing a differentially flat system.</td>
</tr>
<tr>
<td><code>flatsys.LinearFlatSystem</code></td>
<td>Base class for a linear, differentially flat system.</td>
</tr>
<tr>
<td><code>flatsys.PolyFamily</code></td>
<td>Polynomial basis functions.</td>
</tr>
<tr>
<td><code>flatsys.SystemTrajectory</code></td>
<td>Class representing a system trajectory.</td>
</tr>
<tr>
<td><code>optimal.OptimalControlProblem</code></td>
<td>Description of a finite horizon, optimal control problem.</td>
</tr>
<tr>
<td><code>optimal.OptimalControlResult</code></td>
<td>Result from solving an optimal control problem.</td>
</tr>
</tbody>
</table>
The `control.matlab` module contains a number of functions that emulate some of the functionality of MATLAB. The intent of these functions is to provide a simple interface to the python control systems library (python-control) for people who are familiar with the MATLAB Control Systems Toolbox (tm).

### 5.1 Creating linear models

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>tf(num, den[, dt])</code></td>
<td>Create a transfer function system. Can create MIMO systems. The function accepts either 1, 2, or 3 parameters:</td>
</tr>
<tr>
<td><code>ss(A, B, C, D[, dt])</code></td>
<td>Create a state space system.</td>
</tr>
<tr>
<td><code>frd(d, w)</code></td>
<td>Construct a frequency response data model</td>
</tr>
<tr>
<td><code>rss([states, outputs, inputs, strictly_proper])</code></td>
<td>Create a stable random state space object.</td>
</tr>
<tr>
<td><code>drss(*args, **kwargs)</code></td>
<td>Create a stable, discrete-time, random state space system</td>
</tr>
</tbody>
</table>

#### 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

• **inputs** *(str, or list of str, optional)* – List of strings that name the individual signals of the transformed system. If not given, the inputs and outputs are the same as the original system.

• **outputs** *(str, or list of str, optional)* – List of strings that name the individual signals of the transformed system. If not given, the inputs and outputs are the same as the original system.

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

**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**

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

```python
>>> # 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, 2, 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 state space system.

- ss(updfcn, outfucn)
  - Create a nonlinear input/output system with update function updfcn and output function outfcn. See NonlinearIOSystem for more information.

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

- ss(A, B, C, D, dt)
  - Create a discrete-time state space system from the matrices of its state and output equations:
    \[
    x[k + 1] = A \cdot x[k] + B \cdot u[k] \\
y[k] = C \cdot x[k] + D \cdot u[k]
    \]

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

- ss(args, inputs=['u1', ..., 'up'], outputs=['y1', ..., 'yq'],
  states=['x1', ..., 'xn'])
  - Create a system with named input, output, and state signals.

Parameters
- sys (StateSpace or TransferFunction) – A linear system.
- A (array_like or string) – System, control, output, and feed forward matrices.
- B (array_like or string) – System, control, output, and feed forward matrices.
- C (array_like or string) – System, control, output, and feed forward matrices.
- D (array_like or string) – System, control, output, and feed forward matrices.
- dt (None, True or float, optional) – System timebase. 0 (default) indicates continuous time, True indicates discrete time with unspecified sampling time, positive number is discrete time with specified sampling time, None indicates unspecified timebase (either continuous or discrete time).

5.1. Creating linear models
• **inputs** *(str, or list of str, optional)* – List of strings that name the individual signals. If this parameter is not given or given as *None*, the signal names will be of the form *s[i]* (where *s* is one of *u*, *y*, or *x*). See *InputOutputSystem* for more information.

• **outputs** *(str, or list of str, optional)* – List of strings that name the individual signals. If this parameter is not given or given as *None*, the signal names will be of the form *s[i]* (where *s* is one of *u*, *y*, or *x*). See *InputOutputSystem* for more information.

• **states** *(str, or list of str, optional)* – List of strings that name the individual signals. If this parameter is not given or given as *None*, the signal names will be of the form *s[i]* (where *s* is one of *u*, *y*, or *x*). See *InputOutputSystem* for more information.

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

Returns

* out – Linear input/output system.

Return type

* LinearIOSystem

Raises

* ValueError – If matrix sizes are not self-consistent.

See also:

tf, ss2tf, tf2ss

Examples

```python
>>> # Create a Linear I/O system object from from for matrices
>>> sys1 = ss([[1, -2], [3, -4]], [[5], [7]], [[6, 8]], [[9]])

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

5.1.3 **control.matlab.frd**

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

Construct a frequency response data model

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

This function can be called in different ways:

frd(response, freqs)

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

frd(sys, freqs)

Convert an LTI system into an frd model with data at frequencies freqs.

Parameters

• **response** *(array_like, or list)* – complex vector with the system response
• **freq** ([array_like or list]) – 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**

```python
(control.matlab.rss(states=1, outputs=1, inputs=1, strictly_proper=False, **kwargs))
```

Create a stable random state space object.

**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 \(u, y, \text{or } x\)).

- **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.

- **strictly_proper** (bool, optional) – If set to ‘True’, returns a proper system (no direct term).

- **dt** (None, True or float, optional) – System timebase. 0 (default) indicates continuous time, True indicates discrete time with unspecified sampling time, positive number is discrete time with specified sampling time, None indicates unspecified timebase (either continuous or discrete 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**

- sys – The randomly created linear system

**Return type**

- StateSpace

**Raises**

- **ValueError** – if any input is not a positive integer
Notes

If the number of states, inputs, or outputs is not specified, then the missing numbers are assumed to be 1. If dt is not specified or is given as 0 or None, the poles of the returned system will always have a negative real part. If dt is True or a positive float, the poles of the returned system will have magnitude less than 1.

5.1.5 control.matlab.drss

control.matlab.drss(*args, **kwargs)

Create a stable, discrete-time, random state space system

Create a stable discrete time random state space object. This function calls rss() using either the dt keyword provided by the user or dt=True if not specified.

5.2 Utility functions and conversions

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

5.2.1 control.matlab.mag2db

control.matlab.mag2db(mag)

Convert a magnitude to decibels (dB)

If A is magnitude,

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

Parameters

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

Returns

db – corresponding values in decibels

Return type

float or ndarray
5.2.2 control.matlab.db2mag

control.matlab.db2mag(db)

Convert a gain in decibels (dB) to a magnitude

If A is magnitude,

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

**Parameters**

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

**Returns**

- `mag` – corresponding magnitudes

**Return type**

- float or ndarray

5.2.3 control.matlab.c2d

control.matlab.c2d(sysc, Ts, method='zoh', prewarp_frequency=None)

Convert a continuous time system to discrete time by sampling

**Parameters**

- `sysc` (**LTI (StateSpace or TransferFunction)**) – Continuous time system to be converted
- `Ts` (**float > 0**) – Sampling period
- `method` (**string**) – Method to use for conversion, e.g. ‘bilinear’, ‘zoh’ (default)
- `prewarp_frequency` (**real within [0, infinity])** – The frequency [rad/s] at which to match with the input continuous-time system’s magnitude and phase (only valid for `method='bilinear'`)

**Returns**

- `sysd` – Discrete time system, with sampling rate `Ts`

**Return type**

- LTI of the same class

**Notes**

See `StateSpace.sample()` or `TransferFunction.sample()` for further details.

**Examples**

```python
>>> sysc = TransferFunction([1], [1, 2, 1])
>>> sysd = sample_system(sysc, 1, method='bilinear')
```
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 from state space into transfer function form. Always creates a new system.

ss2tf(A, B, C, D)

Create a transfer function system from the matrices of its state and output equations.

For details see: tf()

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
- **inputs** (*str*, or *list of str*, *optional*) – List of strings that name the individual signals of the transformed system. If not given, the inputs and outputs are the same as the original system.
- **outputs** (*str*, or *list of str*, *optional*) – List of strings that name the individual signals of the transformed system. If not given, the inputs and outputs are the same as the original system.
- **name** (*string*, *optional*) – System name. If unspecified, a generic name `<sys[id]>` is generated with a unique integer id.

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

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

```python
>>> 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 space space form. Always creates a new system, even if sys is already a StateSpace object.

**tf2ss(num, den)**

Create a state space 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
- **inputs** (`str`, or list of `str`, optional) – List of strings that name the individual signals of the transformed system. If not given, the inputs and outputs are the same as the original system.
- **outputs** (`str`, or list of `str`, optional) – List of strings that name the individual signals of the transformed system. If not given, the inputs and outputs are the same as the original system.
- **name** (`string`, optional) – System name. If unspecified, a generic name `<sys[id]>` is generated with a unique integer id.

**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

5.2. Utility functions and conversions 131
• **TypeError** – if `num` or `den` are of incorrect type, or if `sys` is not a `TransferFunction` object

See also:

`ss, tf, ss2tf`

### Examples

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

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

### 5.2.6 control.matlab.tfdata

`control.matlab.tfdata(sys)`

Return transfer function data objects for a system

**Parameters**

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

**Returns**

- `(num, den)` – Transfer function coefficients (SISO only)

**Return type**

numerator and denominator arrays

### 5.3 System interconnections

- `series(sys1, sys2, [..., sysn])` — Return the series connection *(sysn * ... * sys2 * sys1).*
- `parallel(sys1, sys2, [..., sysn])` — Return the parallel connection `sys1 + sys2 (+ ... + sysn).`
- `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, sys2[, ..., 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.ninputs does not equal sys1.noutputs if sys1.dt is not compatible with sys2.dt

See also:
parallel, feedback

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

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

Examples

```python
>>> sys3 = series(sys1, sys2) # Same as sys3 = sys2 * sys1
>>> sys5 = series(sys1, sys2, sys3, sys4) # More systems
```

5.3.2 control.matlab.parallel

color.matlab.parallel(sys1, sys2[, ..., 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

5.3. System interconnections
Notes

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

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

Examples

```python
>>> sys3 = parallel(sys1, sys2) # Same as sys3 = sys1 + sys2
>>> 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 \textit{sys1} is a TransferFunction object, and StateSpace.feedback if \textit{sys1} is a StateSpace object. If \textit{sys1} is a scalar, then it is converted to \textit{sys2}'s type, and the corresponding feedback function is used. If \textit{sys1} and \textit{sys2} are both scalars, then TransferFunction.feedback is used.

5.3.4 \texttt{control.matlab.negate}

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

Return the negative of a system.

**Parameters**

- \texttt{sys} (\texttt{StateSpace}, \texttt{TransferFunction} or \texttt{FRD}) –

**Returns**

- \texttt{out}

**Return type**

\texttt{StateSpace} or \texttt{TransferFunction}

Notes

This function is a wrapper for the \texttt{\_\_neg\_\_} function in the StateSpace and TransferFunction classes. The output type is the same as the input type.

**Examples**

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

5.3.5 \texttt{control.matlab.connect}

\texttt{control.matlab.connect(sys, Q, inputv, outputv)}

Index-based interconnection of an LTI system.

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

**Parameters**

- \texttt{sys} (\texttt{StateSpace} or \texttt{TransferFunction}) – System to be connected
- \texttt{Q} (\texttt{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.
- \texttt{inputv} (\texttt{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**

```python
>>> 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])
```

**Notes**

The `interconnect()` function in the `input/output systems` module allows the use of named signals and provides an alternative method for interconnecting multiple systems.

### 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

```python
>>> 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

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

5.4.1 control.matlab.dcgain

`control.matlab.dcgain(*args)`

Compute the gain of the system in steady state.

The function takes either 1, 2, 3, or 4 parameters:

**Parameters**

- `A (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

See also:

zeros, TransferFunction.poles, StateSpace.poles

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

See also:

poles, StateSpace.zeros, TransferFunction.zeros

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, \( wn = abs(poles) \ Z = -\text{real}(poles)/poles \).
• If the system is discrete, the discrete poles are mapped to their equivalent location in the s-plane via
  \( s = \log_{10}(poles)/dt \)
  and \( wn = \text{abs}(s) \ Z = -\text{real}(s)/wn \).

See also:

pole

5.4.5 control.matlab.pzmap

collection.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

• poles (array) – The systems poles
• zeros (array) – The system’s zeros.

Notes

The pzmap function calls matplotlib.pyplot.axis(‘equal’), which means that trying to reset the axis limits may not behave as expected. To change the axis limits, use matplotlib.pyplot.gca().axis(‘auto’) and then set the axis limits to the desired values.

5.5 Time-domain analysis

<table>
<thead>
<tr>
<th>function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>step(sys[, T, X0, input, output, return_x])</td>
<td>Step response of a linear system</td>
</tr>
<tr>
<td>impulse(sys[, T, X0, input, output, return_x])</td>
<td>Impulse response of a linear system</td>
</tr>
<tr>
<td>initial(sys[, T, X0, input, output, return_x])</td>
<td>Initial condition response of a linear system</td>
</tr>
<tr>
<td>lsim(sys[, U, T, X0])</td>
<td>Simulate the output of a linear system.</td>
</tr>
</tbody>
</table>
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

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

5.5.2 control.matlab.impulse

control.matlab.impulse(sys, T=None, X0=0.0, input=0, output=None, return_x=False)

Impulse response of a linear system

If the system has multiple inputs or outputs (MIMO), one input has to be selected for the simulation. Optionally, one output may be selected. If no selection is made for the output, all outputs are given. The parameters `input` and `output` do this. All other inputs are set to 0, all other outputs are ignored.

Parameters

- `sys` (StateSpace, TransferFunction) – LTI system to simulate
- `T` (array-like or number, optional) – Time vector, or simulation time duration if a number (time vector is autocomputed if not given)

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**

```python
>>> 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

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

### 5.5.4 control.matlab.lsim

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

Simulate the output of a linear system.

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

**Parameters**

- `sys (LTI (StateSpace, or TransferFunction))` – LTI system to simulate
- `U (array-like or number, optional)` – Input array giving input at each time `T` (default = 0).
  
  If `U` is `None` or `0`, a special algorithm is used. This special algorithm is faster than the general algorithm, which is used otherwise.
- `T (array-like, 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**

```python
>>> 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])**

Nyquist plot of the frequency response

**nichols(sys_list[, omega, grid])**

Nichols plot for a system

**margin(sysdata)**

Calculate gain and phase margins and associated crossover frequencies

**freqresp(sys, omega[, squeeze])**

Frequency response of an LTI system at multiple angular frequencies.

**evalfr(sys, x[, squeeze])**

Evaluate the transfer function of an LTI system for complex frequency $x$. 
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

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

Todo: Document these use cases

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

5.6.2 control.matlab.nyquist

control.matlab.nyquist(syslist[, omega])

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

Parameters

- **sys1** *(list of LTI)* – List of linear input/output systems (single system is OK).
- **...** *(list of LTI)* – List of linear input/output systems (single system is OK).
• **sysn** *(list of LTI)* – List of linear input/output systems (single system is OK).

• **omega** *(array_like)* – Set of frequencies to be evaluated, in rad/sec.

**Returns**

• **real** *(ndarray (or list of ndarray if len(syslist) > 1))* – real part of the frequency response array

• **imag** *(ndarray (or list of ndarray if len(syslist) > 1))* – imaginary part of the frequency response array

• **omega** *(ndarray (or list of ndarray if len(syslist) > 1))* – frequencies in rad/s

### 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.

**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 representing the loop transfer function

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

**Returns**

• **gm** *(float)* – Gain margin

• **pm** *(float)* – Phase margin (in degrees)

• **wcg** *(float or array_like)* – Crossover frequency associated with gain margin (phase crossover frequency), where phase crosses below -180 degrees.

• **wcp** *(float or array_like)* – Crossover frequency associated with phase margin (gain crossover frequency), where gain crosses below 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

```python
>>> sys = tf(1, [1, 2, 1, 0])
>>> gm, pm, wcg, wcp = margin(sys)
```

5.6.5 `control.matlab.freqresp`

`control.matlab.freqresp(sys, omega, squeeze=None)`

Frequency response of an LTI system at multiple angular frequencies.

In general the system may be multiple input, multiple output (MIMO), where `m = sys.ninputs` number of inputs and `p = sys.noutputs` number of outputs.

Parameters

- `sys` *(StateSpace or TransferFunction)* – Linear system
- `omega` *(float or 1D 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.
- `squeeze` *(bool, optional)* – If squeeze=True, remove single-dimensional entries from the shape of the output even if the system is not SISO. If squeeze=False, keep all indices (output, input and, if omega is array_like, frequency) even if the system is SISO. The default value can be set using config.defaults['control.squeeze_frequency_response'].

Returns

- `response` – Frequency response data object representing the frequency response. This object can be assigned to a tuple using
  ```
  mag, phase, omega = response
  ```
  where `mag` is the magnitude (absolute value, not dB or log10) of the system frequency response, `phase` is the wrapped phase in radians of the system frequency response, and `omega` is the (sorted) frequencies at which the response was evaluated. If the system is SISO and squeeze is not True, `magnitude` and `phase` are 1D, indexed by frequency. If the system is not SISO or squeeze is False, the array is 3D, indexed by the output, input, and frequency. If `squeeze` is True then single-dimensional axes are removed.

Return type

`FrequencyResponseData`

See also:

`evalfr, bode`
Notes

This function is a wrapper for `StateSpace.frequency_response()` and `TransferFunction.frequency_response()`.

Examples

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

Todo: Add example with MIMO system

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

5.6.6 control.matlab.evalfr

`control.matlab.evalfr(sys, x, squeeze=None)`

Evaluate the transfer function of an LTI system for complex frequency `x`.

Returns the complex frequency response `sys(x)` where `x` is `s` for continuous-time systems and `z` for discrete-time systems, with `m = sys.ninputs` number of inputs and `p = sys.noutputs` number of outputs.

To evaluate at a frequency omega in radians per second, enter `x = omega * 1j` for continuous-time systems, or `x = exp(1j * omega * dt)` for discrete-time systems, or use `freqresp(sys, omega)`.

Parameters

- `sys` (`StateSpace` or `TransferFunction`) – Linear system
- `x` (`complex scalar` or `1D array_like`) – Complex frequency(s)
- `squeeze` (`bool`, optional (`default=True`)) – If `squeeze=True`, remove single-dimensional entries from the shape of the output even if the system is not SISO. If `squeeze=False`, keep all indices (output, input and, if omega is array_like, frequency) even if the system is SISO. The default value can be set using `config.defaults['control.squeeze_frequency_response']`.

Returns

- `fresp` – The frequency response of the system. If the system is SISO and `squeeze` is not True, the shape of the array matches the shape of omega. If the system is not SISO or `squeeze` is False, the first two dimensions of the array are indices for the output and input and the remaining dimensions match omega. If `squeeze` is True then single-dimensional axes are removed.

Return type

- complex ndarray
See also:

freqresp, bode

Notes

This function is a wrapper for StateSpace.__call__() and TransferFunction.__call__().

Examples

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

Todo: Add example with MIMO system

5.7 Compensator design

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rlocus(sys[, kvect, xlim, ylim, plotstr, ...])</td>
<td>Root locus plot</td>
</tr>
<tr>
<td>sisotool(sys[, kvect, xlim_rlocus, ...])</td>
<td>Sisotool style collection of plots inspired by MATLAB’s sisotool.</td>
</tr>
<tr>
<td>place(A, B, p)</td>
<td>Place closed loop eigenvalues</td>
</tr>
<tr>
<td>lqr(A, B, Q, R[, N])</td>
<td>Linear quadratic regulator design</td>
</tr>
</tbody>
</table>

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.

**Notes**

The root_locus function calls matplotlib.pyplot.axis('equal'), which means that trying to reset the axis limits may not behave as expected. To change the axis limits, use matplotlib.pyplot.gca().axis('auto') and then set the axis limits to the desired values.

### 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)

Sisotoolstyle collection of plots inspired by MATLAB’s sisotool. The left twoplots 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. If sys is SISO, use the same system for the root locus and step response. If it is desired to see a different step response than feedback(K*loop,1), sys can be provided as a two-input, two-output system (e.g. by using bdgalg.connect' or :func:`iosys.interconnect`). Sisotool inserts the negative of the selected gain K between the first output and first input and uses the second input and output for computing the step response. This allows you to see the step responses of more complex systems, for example, systems with a feedforward path into the plant or in which the gain appears in the feedback path.

• **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 tu-

• **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- or z-plane grid.

• **omega**(array_like) – List of frequencies in rad/sec to be used for 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** *(array_like of two values)* – Limits of the to generate frequency vector. If Hz=True the limits are in Hz otherwise in rad/s. Ignored if omega is provided, and auto-generated if omitted.
• **omega_num** *(int)* – Number of samples to plot. Defaults to config.defaults[‘freqplot.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

```python
>>> 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_like)* – Dynamics matrix
• **B** *(2D array_like)* – Input matrix
• **p** *(1D array_like)* – 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

```python
>>> 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 $u = -K x$ 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:

- `K, S, E = lqr(sys, Q, R)`
- `K, S, E = lqr(sys, Q, R, N)`
- `K, S, E = lqr(A, B, Q, R)`
- `K, S, E = 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_like)` – Dynamics and input matrices
- `B (2D array_like)` – Dynamics and input matrices
- `sys (LTI StateSpace system)` – Linear system
- `Q (2D array)` – State and input weight matrices
- `R (2D array)` – State and input weight matrices
- `N (2D array, optional)` – Cross weight matrix
- `integral_action (ndarray, optional)` – If this keyword is specified, the controller includes integral action in addition to state feedback. The value of the `integral_action` keyword should be an ndarray that will be multiplied by the current to generate the error for the internal integrator states of the control law. The number of outputs that are to be integrated must match the number of additional rows and columns in the `Q` matrix.
**method**(str, optional) – Set the method used for computing the result. Current methods are ‘slycot’ and ‘scipy’. If set to None (default), try ‘slycot’ first and then ‘scipy’.

**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, dlqr, dlqe

**Notes**

1. If the first argument is an LTI object, then this object will be used to define the dynamics and input matrices. Furthermore, if the LTI object corresponds to a discrete time system, the dlqr() function will be called.

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

**Examples**

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

## 5.8 State-space (SS) models

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rss([states, outputs, inputs, strictly_proper])</td>
<td>Create a stable random state space object.</td>
</tr>
<tr>
<td>drss(*args, **kwargs)</td>
<td>Create a stable, discrete-time, random state space system</td>
</tr>
<tr>
<td>ctrb(A, B)</td>
<td>Controllability matrix</td>
</tr>
<tr>
<td>obsv(A, C)</td>
<td>Observability matrix</td>
</tr>
<tr>
<td>gram(sys, type)</td>
<td>Gramian (controllability or observability)</td>
</tr>
</tbody>
</table>

### 5.8.1 control.matlab ctrb

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

Controllability 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

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

5.8.2 control.matlab.obsv

控制.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

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

5.8.3 control.matlab.gram

控制.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)
- **ControlSlycot** – 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

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

5.9 Model simplification

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

5.9.1 control.matlab.minreal

`control.matlab.minreal(sys, tol=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
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.


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

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

5.9.4 control.matlab.modred

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

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

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

Returns
rsys – 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

```python
>>> rsys = modred(sys, ELIM, method='truncate')
```
5.9.5 control.matlab.era

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

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

Note: This function is not implemented yet.

Parameters
- **YY** (array) – nout 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

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

5.9.6 control.matlab.markov

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

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 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*. Default value is False.

**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**

```python
>>> 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`
5.11 Matrix equation solvers and linear algebra

\begin{align*}
\text{lyap}(A, Q[, C, E, method]) & \quad X = \text{lyap}(A, Q) \text{ solves the continuous-time Lyapunov equation} \\
\text{dlyap}(A, Q[, C, E, method]) & \quad \text{dlyap}(A, Q) \text{ solves the discrete-time Lyapunov equation} \\
\text{care}(A, B, Q[, R, S, E, stabilizing, ...]) & \quad X, L, G = \text{care}(A, B, Q, R=None) \text{ solves the continuous-time algebraic Riccati equation} \\
\text{dare}(A, B, Q, R[, S, E, stabilizing, ...]) & \quad X, L, G = \text{dare}(A, B, Q, R) \text{ solves the discrete-time algebraic Riccati equation}
\end{align*}

5.11.1 control.matlab.lyap

\text{control.matlab.lyap}(A, Q, C=None, E=None, method=None)

\begin{align*}
X & = \text{lyap}(A, Q) \text{ solves the continuous-time Lyapunov equation} \\
AX + XA^T + Q & = 0 \\
\text{where A and Q are square matrices of the same dimension. Q must be symmetric.}
\end{align*}

\begin{align*}
X & = \text{lyap}(A, Q, C) \text{ solves the Sylvester equation} \\
AX + XQ + C & = 0 \\
\text{where A and Q are square matrices.}
\end{align*}

\begin{align*}
X & = \text{lyap}(A, Q, None, E) \text{ solves the generalized continuous-time Lyapunov equation} \\
AXE^T + EXA^T + Q & = 0 \\
\text{where Q is a symmetric matrix and A, Q and E are square matrices of the same dimension.}
\end{align*}

Parameters

- A (2D array_like) – Input matrices for the Lyapunov or Sylvester equation
- Q (2D array_like) – Input matrices for the Lyapunov or Sylvester equation
- C (2D array_like, optional) – If present, solve the Sylvester equation
- E (2D array_like, optional) – If present, solve the generalized Lyapunov equation
- method (str, optional) – Set the method used for computing the result. Current methods are ‘slycot’ and ‘scipy’. If set to None (default), try ‘slycot’ first and then ‘scipy’.

Returns

- X – Solution to the Lyapunov or Sylvester equation

Return type

- 2D array (or matrix)
5.11.2 control.matlab.dlyap

control.matlab.dlyap(A, Q, C=None, E=None, method=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.

Parameters
- A (2D array_like) – Input matrices for the Lyapunov or Sylvester equation
- Q (2D array_like) – Input matrices for the Lyapunov or Sylvester equation
- C (2D array_like, optional) – If present, solve the Sylvester equation
- E (2D array_like, optional) – If present, solve the generalized Lyapunov equation
- method (str, optional) – Set the method used for computing the result. Current methods are `slycot` and `scipy`. If set to None (default), try `slycot` first and then `scipy`.

Returns
- X – 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().
5.11.3 control.matlab.care

control.matlab.care(A, B, Q, R=None, S=None, E=None, stabilizing=True, method=None, A_s='A', B_s='B', Q_s='Q', R_s='R', S_s='S', E_s='E')

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

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

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

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

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

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

Parameters

- A (2D array_like) – Input matrices for the Riccati equation
- B (2D array_like) – Input matrices for the Riccati equation
- Q (2D array_like) – Input matrices for the Riccati equation
- R (2D array_like, optional) – Input matrices for generalized Riccati equation
- S (2D array_like, optional) – Input matrices for generalized Riccati equation
- E (2D array_like, optional) – Input matrices for generalized Riccati equation
- method (str, optional) – Set the method used for computing the result. Current methods are ‘slycot’ and ‘scipy’. If set to None (default), try ‘slycot’ first and then ‘scipy’.

Returns

- X (2D array (or matrix)) – Solution to the Riccati 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, method=None, A_s='A', B_s='B', Q_s='Q', R_s='R', S_s='S', E_s='E')

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

\[ A^T X A - X - A^T X B (B^T X B + R)^{-1} B^T X A + Q = 0 \]
where A and Q are square matrices of the same dimension. Further, Q is a symmetric matrix. The function returns the solution X, the gain matrix G = (B^T X B + R)^{-1} B^T X A and the closed loop eigenvalues L, i.e., the eigenvalues of A - B G.

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

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

where A, Q and E are square matrices of the same dimension. Further, Q and R are symmetric matrices. If R is None, it is set to the identity matrix. 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 (generalized) eigenvalues of A - B G (with respect to E, if specified).

**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
- **method (str, optional)** – Set the method used for computing the result. Current methods are ‘slycot’ and ‘scipy’. If set to None (default), try ‘slycot’ first and then ‘scipy’.

**Returns**

- **X (2D array (or matrix))** – Solution to the Riccati 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

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>gangof4(P, C[, omega])</code></td>
<td>Plot the “Gang of 4” transfer functions for a system</td>
</tr>
<tr>
<td><code>unwrap(angle[, period])</code></td>
<td>Unwrap a phase angle to give a continuous curve</td>
</tr>
</tbody>
</table>

5.12. Additional functions 161
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)

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

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>linspace</code></td>
<td>Return evenly spaced numbers over a specified interval.</td>
</tr>
<tr>
<td><code>logspace</code></td>
<td>Return numbers spaced evenly on a log scale.</td>
</tr>
<tr>
<td><code>ss2zpk</code></td>
<td>State-space representation to zero-pole-gain representation.</td>
</tr>
<tr>
<td><code>tf2zpk</code></td>
<td>Return zero, pole, gain (z, p, k) representation from a numerator, denominator representation of a linear filter.</td>
</tr>
<tr>
<td><code>zpk2ss</code></td>
<td>Zero-pole-gain representation to state-space representation.</td>
</tr>
<tr>
<td><code>zpk2tf</code></td>
<td>Return polynomial transfer function representation from zeros and poles.</td>
</tr>
</tbody>
</table>
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

\[
\dot{x} = f(x, u), \quad x \in \mathbb{R}^n, u \in \mathbb{R}^m
\]

is differentially flat if there exists a function \( \alpha \) 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

\[
x = \beta(z, \dot{z}, \ldots, z^{(q)})
\]

\[
u = \gamma(z, \dot{z}, \ldots, z^{(q)}).
\]

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

\[
\dot{x} = f(x, u), \quad x(0) = x_0, x(T) = x_f.
\]

If the system is differentially flat then

\[
x(0) = \beta(z(0), \dot{z}(0), \ldots, z^{(q)}(0)) = x_0,
\]

\[
x(T) = \gamma(z(T), \dot{z}(T), \ldots, z^{(q)}(T)) = x_f,
\]

and we see that the initial and final condition in the full state space depends on just the output \( z \) and its derivatives at the initial and final times. Thus any trajectory for \( z \) that satisfies these boundary conditions will be a feasible trajectory for the system, using equation (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 $\psi_i(t)$:

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

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

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

$$\vdots$$

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

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

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

This equation is a linear equation of the form

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

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) $\alpha$ 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:

$$z\text{flag} = \text{sys.forward}(x, u)$$

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

$$x, u = \text{sys.reverse}(z\text{flag})$$

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

$$z\text{flag}[i][j] = z_i^{(j)}$$
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:

```python
sys = control.flatsys.LinearFlatSystem(linsys)
```

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

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

In addition to the flat system description, 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 $1, t, t^2, \ldots$ can be computed using the `PolyBasis` class, which is initialized by passing the desired order of the polynomial basis set:

```python
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:

```python
traj = control.flatsys.point_to_point(
    sys, Tf, x0, u0, xf, uf, 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:

```python
xd, ud = traj.eval(T)
```

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

The `point_to_point()` function also allows the specification of a cost function and/or constraints, in the same format as `solve_ocp()`.

### 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 in *Feedback Systems* by Astrom and Murray, Example 3.11.

```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[1] * np.sin(x[2])  # dy/dt
```

(continues on next page)
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])
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 \(\delta\) 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, Tf, x0, u0, xf, uf, basis=poly)

# Create the trajectory
t = np.linspace(0, Tf, 100)
\[ x, u = \text{traj.eval}(t) \]

### 6.4 Module classes and functions

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BasisFamily(N)</td>
<td>Base class for implementing basis functions for flat systems.</td>
</tr>
<tr>
<td>BezierFamily(N[, T])</td>
<td>Bezier curve basis functions.</td>
</tr>
<tr>
<td>FlatSystem(forward, reverse[, updfcn, ...])</td>
<td>Base class for representing a differentially flat system.</td>
</tr>
<tr>
<td>LinearFlatSystem(linsys[, inputs, outputs, ...])</td>
<td>Base class for a linear, differentially flat system.</td>
</tr>
<tr>
<td>PolyFamily(N)</td>
<td>Polynomial basis functions.</td>
</tr>
<tr>
<td>SystemTrajectory(sys, basis[, coeffs, ...])</td>
<td>Class representing a system trajectory.</td>
</tr>
</tbody>
</table>

#### 6.4.1 control.flatsys.BasisFamily

class control.flatsys.BasisFamily(N)

    Bases: object

    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:

    \[ z_i^{(q)}(t) = \text{basis.eval_deriv}(self, i, j, t) \]

    **Parameters**

    *N* (int) – Order of the basis set.

    **Methods**

    | eval_deriv |
    |------------|
    __call__(i, t) | Evaluate the ith basis function at a point in time |

#### 6.4.2 control.flatsys.BezierFamily

class control.flatsys.BezierFamily(N, T=1)

    Bases: BasisFamily

    Bezier curve basis functions.

    This class represents the family of polynomials of the form

    \[ \phi_i(t) = \sum_{i=0}^{n} \binom{n}{i} \left( \frac{t}{T_f} \right)^{n-i} \left( 1 - \frac{t}{T_f} \right)^i \]
Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>eval_deriv</code></td>
<td>Evaluate the kth derivative of the ith basis function at time t.</td>
</tr>
<tr>
<td><code>__call__</code></td>
<td>Evaluate the ith basis function at a point in time</td>
</tr>
<tr>
<td><code>eval_deriv</code></td>
<td>Evaluate the kth derivative of the ith basis function at time t.</td>
</tr>
</tbody>
</table>

6.4.3 control.flatsys.FlatSystem

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

Bases: NonlinearIOSystem

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 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 current 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)

**Notes**

The class must implement two functions:

\[ z_{flag} = \text{flatsys.forward}(x, u, \text{params}) \]

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).

\[ x, u = \text{flatsys.reverse}(z_{flag}, \text{params}) \]

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.

**Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>copy</strong></td>
<td>Make a copy of an input/output system</td>
</tr>
<tr>
<td><strong>dynamics</strong></td>
<td>Compute the dynamics of a differential or difference equation.</td>
</tr>
<tr>
<td><strong>feedback</strong></td>
<td>Feedback interconnection between two input/output systems</td>
</tr>
<tr>
<td><strong>find_input</strong></td>
<td>Find the index for an input given its name ( (None \text{ if not found}) )</td>
</tr>
<tr>
<td><strong>find_output</strong></td>
<td>Find the index for an output given its name ( (None \text{ if not found}) )</td>
</tr>
<tr>
<td><strong>find_state</strong></td>
<td>Find the index for a state given its name ( (None \text{ if not found}) )</td>
</tr>
<tr>
<td><strong>forward</strong></td>
<td>Compute the flat flag given the states and input.</td>
</tr>
<tr>
<td><strong>isctime</strong></td>
<td>Check to see if a system is a continuous-time system</td>
</tr>
<tr>
<td><strong>isdtime</strong></td>
<td>Check to see if a system is a discrete-time system</td>
</tr>
<tr>
<td><strong>issiso</strong></td>
<td>Check to see if a system is single input, single output.</td>
</tr>
<tr>
<td><strong>linearize</strong></td>
<td>Linearize an input/output system at a given state and input.</td>
</tr>
<tr>
<td><strong>output</strong></td>
<td>Compute the output of the system</td>
</tr>
<tr>
<td><strong>reverse</strong></td>
<td>Compute the states and input given the flat flag.</td>
</tr>
<tr>
<td><strong>set_inputs</strong></td>
<td>Set the number/names of the system inputs.</td>
</tr>
<tr>
<td><strong>set_outputs</strong></td>
<td>Set the number/names of the system outputs.</td>
</tr>
<tr>
<td><strong>set_states</strong></td>
<td>Set the number/names of the system states.</td>
</tr>
</tbody>
</table>

\[ __add__(\text{sys2}) \]

Add two input/output systems (parallel interconnection)
__call__(u, params=None, squeeze=None)

Evaluate a (static) nonlinearity at a given input value

If a nonlinear I/O system has no internal state, then evaluating the system at an input \( u \) gives the output \( y = F(u) \), determined by the output function.

Parameters

- **params** *(dict, optional)* – Parameter values for the system. Passed to the evaluation function for the system as default values, overriding internal defaults.
- **squeeze** *(bool, optional)* – If True and if the system has a single output, return the system output as a 1D array rather than a 2D array. If False, return the system output as a 2D array even if the system is SISO. Default value set by config.defaults[‘control.squeeze_time_response’].

__mul__(sys1)

Multiply two input/output systems (series interconnection)

__neg__()

Negate an input/output systems (rescale)

__radd__(sys2)

Parallel addition of input/output system to a compatible object.

__rmul__(sys2)

Pre-multiply an input/output systems by a scalar/matrix

__rsub__(sys2)

Parallel subtraction of I/O system to a compatible object.

__sub__(sys2)

Subtract two input/output systems (parallel interconnection)

copy(name=None, use_prefix_suffix=True)

Make a copy of an input/output system

A copy of the system is made, with a new name. The name keyword can be used to specify a specific name for the system. If no name is given and use_prefix_suffix is True, the name is constructed by prepending config.defaults[‘iosys.duplicate_system_name_prefix’] and appending config.defaults[‘iosys.duplicate_system_name_suffix’]. Otherwise, a generic system name of the form \( \text{sys[<id>]} \) is used, where \( <id> \) is based on an internal counter.

dynamics(t, x, u)

Compute the dynamics of a differential or difference equation.

Given time \( t \), input \( u \) and state \( x \), returns the value of the right hand side of the dynamical system. If the system is continuous, returns the time derivative

\[
\frac{dx}{dt} = f(t, x, u)
\]

where \( f \) is the system’s (possibly nonlinear) dynamics function. If the system is discrete-time, returns the next value of \( x \):

\[
x[t+dt] = f(t, x[t], u[t])
\]

Where \( t \) is a scalar.

The inputs \( x \) and \( u \) must be of the correct length.

Parameters
• t (float) – the time at which to evaluate
• x (array_like) – current state
• u (array_like) – input

Returns
dx/dt or x[t+dt]

Return type
ndarray

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

Parameters
   • sys (Named I/O system) – System to be checked
   • strict (bool, optional) – If strict is True, make sure that timebase is not None. Default is False.

isdtme(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

linearize(x0, u0, t=0, params=(), eps=1e-06, name=None, copy=False, **kwargs)
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.

ninputs
Number of system inputs.

noutputs
Number of system outputs.

nstates
Number of system states.

output(t, x, u)
Compute the output of the system

Given time t, input u and state x, returns the output of the system:

    y = g(t, x, u)

The inputs x and u must be of the correct length.

Parameters
   • t (float) – the time at which to evaluate
   • x (array_like) – current state
   • u (array_like) – input

Returns
   y

Return type
   ndarray
reverse\( (zflag, \text{params}={} ) \)

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.
- **\text{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 \( \prefix[i] \).
6.4.4 control.flatsys.LinearFlatSystem

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

Bases: FlatSystem, LinearIOSystem

Base class for a linear, differentially flat system.

This class is used to create a differentially flat system representation from a linear system.

Parameters

• **linsys** ([StateSpace](https://docs.python-control.github.io/ pulp/control/state_space.html)) – LTI StateSpace system to be converted

• **inputs** ([int, list of str or None, optional](https://docs.python-control.github.io/pulp/control/io_system.html)) – 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](https://docs.python-control.github.io/pulp/control/io_system.html)) – Description of the system outputs. Same format as `inputs`.

• **states** ([int, list of str or None, optional](https://docs.python-control.github.io/pulp/control/io_system.html)) – Description of the system states. Same format as `inputs`.

• **dt** ([None, True or float, optional](https://docs.python-control.github.io/pulp/control/io_system.html)) – 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](https://docs.python-control.github.io/pulp/control/io_system.html)) – Parameter values for the systems. Passed to the evaluation functions for the system as default values, overriding internal defaults.

• **name** ([string, optional](https://docs.python-control.github.io/pulp/control/io_system.html)) – System name (used for specifying signals)

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>append</strong></td>
<td>Append a second model to the present model.</td>
</tr>
<tr>
<td><strong>copy</strong></td>
<td>Make a copy of an input/output system.</td>
</tr>
<tr>
<td><strong>damp</strong></td>
<td>Natural frequency, damping ratio of system poles.</td>
</tr>
<tr>
<td><strong>dcgain</strong></td>
<td>Return the zero-frequency gain.</td>
</tr>
<tr>
<td><strong>dynamics</strong></td>
<td>Compute the dynamics of a differential or difference equation.</td>
</tr>
<tr>
<td><strong>feedback</strong></td>
<td>Feedback interconnection between two input/output systems.</td>
</tr>
<tr>
<td><strong>find_input</strong></td>
<td>Find the index for an input given its name (<code>None</code> if not found).</td>
</tr>
<tr>
<td><strong>find_output</strong></td>
<td>Find the index for an output given its name (<code>None</code> if not found).</td>
</tr>
<tr>
<td><strong>find_state</strong></td>
<td>Find the index for a state given its name (<code>None</code> if not found).</td>
</tr>
<tr>
<td><strong>forward</strong></td>
<td>Compute the flat flag given the states and input.</td>
</tr>
<tr>
<td><strong>freqresp</strong></td>
<td>(deprecated) Evaluate transfer function at complex frequencies.</td>
</tr>
<tr>
<td><strong>frequency_response</strong></td>
<td>Evaluate the linear time-invariant system at an array of angular frequencies.</td>
</tr>
</tbody>
</table>
Table 1 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>horner</td>
<td>Evaluate system's transfer function at complex frequency using Laub's or Horner's method.</td>
</tr>
<tr>
<td>istime</td>
<td>Check to see if a system is a continuous-time system</td>
</tr>
<tr>
<td>istime</td>
<td>Check to see if a system is a discrete-time system</td>
</tr>
<tr>
<td>issiso</td>
<td>Check to see if a system is single input, single output</td>
</tr>
<tr>
<td>lift</td>
<td>Return the Linear Fractional Transformation.</td>
</tr>
<tr>
<td>linearize</td>
<td>Linearize an input/output system at a given state and input.</td>
</tr>
<tr>
<td>minreal</td>
<td>Calculate a minimal realization, removes unobservable and uncontrollable states</td>
</tr>
<tr>
<td>output</td>
<td>Compute the output of the system</td>
</tr>
<tr>
<td>pole</td>
<td></td>
</tr>
<tr>
<td>poles</td>
<td>Compute the poles of a state space system.</td>
</tr>
<tr>
<td>returnScipySignalLI</td>
<td>Return a list of a list of scipy.signal.lti objects.</td>
</tr>
<tr>
<td>reverse</td>
<td>Compute the states and input given the flat flag.</td>
</tr>
<tr>
<td>sample</td>
<td>Convert a continuous time system to discrete time</td>
</tr>
<tr>
<td>set_inputs</td>
<td>Set the number/names of the system inputs.</td>
</tr>
<tr>
<td>set_outputs</td>
<td>Set the number/names of the system outputs.</td>
</tr>
<tr>
<td>set_states</td>
<td>Set the number/names of the system states.</td>
</tr>
<tr>
<td>slycot_laub</td>
<td>Evaluate system's transfer function at complex frequency using Laub's method from Slycot.</td>
</tr>
<tr>
<td>zero</td>
<td></td>
</tr>
<tr>
<td>zeros</td>
<td>Compute the zeros of a state space system.</td>
</tr>
</tbody>
</table>

A

Dynamics matrix.

B

Input matrix.

C

Output matrix.

D

Direct term.

__add__ (sys2)

Add two input/output systems (parallel interconnection)

__call__ (u, params=None, squeeze=None)

Evaluate a (static) nonlinearity at a given input value

If a nonlinear I/O system has no internal state, then evaluating the system at an input $u$ gives the output $y = F(u)$, determined by the output function.

Parameters
• **params** (*dict, optional*) – Parameter values for the system. Passed to the evaluation function for the system as default values, overriding internal defaults.

• **squeeze** (*bool, optional*) – If True and if the system has a single output, return the system output as a 1D array rather than a 2D array. If False, return the system output as a 2D array even if the system is SISO. Default value set by config.defaults[‘control.squeeze_time_response’].

```python
__div__(other)
Divide two LTI systems.
```

```python
__getitem__(indices)
Array style access
```

```python
__mul__(sys1)
Multiply two input/output systems (series interconnection)
```

```python
__neg__()
Negate an input/output systems (rescale)
```

```python
__radd__(sys2)
Parallel addition of input/output system to a compatible object.
```

```python
__rdiv__(other)
Right divide two LTI systems.
```

```python
__rmul__(sys2)
Pre-multiply an input/output systems by a scalar/matrix
```

```python
__rsub__(sys2)
Parallel subtraction of I/O system to a compatible object.
```

```python
__sub__(sys2)
Subtract two input/output systems (parallel interconnection)
```

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

```python
copy(name=None, use_prefix_suffix=True)
Make a copy of an input/output system
A copy of the system is made, with a new name. The name keyword can be used to specify a specific name for the system. If no name is given and use_prefix_suffix is True, the name is constructed by prepending config.defaults[‘iosys.duplicate_system_name_prefix’] and appending config.defaults[‘iosys.duplicate_system_name_suffix’]. Otherwise, a generic system name of the form sys[<id>] is used, where <id> is based on an internal counter.
```

```python
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**(*warn_infinite=False*)

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:

**Parameters**

- **warn_infinite** (bool, optional) – By default, don’t issue a warning message if the zero-frequency gain is infinite. Setting *warn_infinite* to generate the warning message.

**Returns**

- **gain** – Array or scalar value for SISO systems, depending on config.defaults['control.squeeze_frequency_response']. The value of the array elements or the scalar is either the zero-frequency (or DC) gain, or *inf*, if the frequency response is singular.

  For real valued systems, the empty imaginary part of the complex zero-frequency response is discarded and a real array or scalar is returned.

**Return type**

(noutputs, ninputs) ndarray or scalar

**dynamics**(*t, x, u*)

Compute the dynamics of a differential or difference equation.

Given time *t*, input *u* and state *x*, returns the value of the right hand side of the dynamical system. If the system is continuous, returns the time derivative

\[ \frac{dx}{dt} = f(t, x, u) \]

where *f* is the system’s (possibly nonlinear) dynamics function. If the system is discrete-time, returns the next value of *x*:

\[ x[t+dt] = f(t, x[t], u[t]) \]

Where *t* is a scalar.

The inputs *x* and *u* must be of the correct length.

**Parameters**

- **t** (float) – the time at which to evaluate
- **x** (array_like) – current state
- **u** (array_like) – input

**Returns**

- **dx/dt or x[t+dt]**

**Return type**

ndarray

**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.

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

`freqresp(omega)`

(deprecated) Evaluate transfer function at complex frequencies.

`frequency_response(omega, squeeze=None)`

Evaluate the linear time-invariant system at an array of angular frequencies.

Reports the frequency response of the system,

\[
G(j*\omega) = \text{mag} \times \exp(j*\text{phase})
\]

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

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

In general the system may be multiple input, multiple output (MIMO), where \( m = self.ninputs \) number of inputs and \( p = self.noutputs \) number of outputs.

Parameters

• `omega (float or 1D array_like)` – A list, tuple, array, or scalar value of frequencies in radians/sec at which the system will be evaluated.

• `squeeze (bool, optional)` – If `squeeze=True`, remove single-dimensional entries from the shape of the output even if the system is not SISO. If `squeeze=False`, keep all indices (output, input and, if `omega` is array_like, frequency) even if the system is SISO. The default value can be set using `config.defaults['control.squeeze_frequency_response']`.

Returns

`response` – Frequency response data object representing the frequency response. This object can be assigned to a tuple using

\[
\text{mag, phase, omega} = \text{response}
\]
where \( \text{mag} \) is the magnitude (absolute value, not dB or log10) of the system frequency response, \( \text{phase} \) is the wrapped phase in radians of the system frequency response, and \( \omega \) is the (sorted) frequencies at which the response was evaluated. If the system is SISO and squeeze is not True, \( \text{magnitude} \) and \( \text{phase} \) are 1D, indexed by frequency. If the system is not SISO or squeeze is False, the array is 3D, indexed by the output, input, and frequency. If squeeze is True then single-dimensional axes are removed.

**Return type**

FrequencyResponseData

**horner**(*x*, **warn_infinite=True**)

Evaluate system’s transfer function at complex frequency using Laub’s or Horner’s method. Evaluates \( \text{sys}(x) \) where \( x \) is \( s \) for continuous-time systems and \( z \) for discrete-time systems. Expects inputs and outputs to be formatted correctly. Use \( \text{sys}(x) \) for a more user-friendly interface.

**Parameters**

- \( x \) (complex array_like or complex) – Complex frequencies

**Returns**

- **output** – Frequency response

**Return type**

(self.noutputs, self.ninputs, len(x)) complex ndarray

**Notes**

Attempts to use Laub’s method from Slycot library, with a fall-back to python code.

**property inputs**

Deprecated attribute; use \( \text{ninputs} \) instead.

The \( \text{inputs} \) attribute was used to store the number of system inputs. It is no longer used. If you need access to the number of inputs for an LTI system, use \( \text{ninputs} \).

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

**Parameters**

- **sys** (Named I/O 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, name=None, copy=False, **kwargs*)

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

**ninputs**

Number of system inputs.

**noutputs**

Number of system outputs.

**nstates**

Number of system states.

**output**(*t, x, u*)

Compute the output of the system

Given time *t*, input *u* and state *x*, returns the output of the system:

\[ y = g(t, x, u) \]

The inputs *x* and *u* must be of the correct length.

Parameters

- **t** (float) – the time at which to evaluate
- **x** (array_like) – current state
- **u** (array_like) – input

Returns

\[ y \]

Return type

ndarray

**property outputs**

Deprecated attribute; use **noutputs** instead.

The **outputs** attribute was used to store the number of system outputs. It is no longer used. If you need access to the number of outputs for an LTI system, use **noutputs**.

**poles()**

Compute the poles of a state space system.
returnScipySignalLTI(strict=True)

Return a list of a list of \texttt{scipy.signal.lti} objects.

For instance,

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

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

**Parameters**

strict (bool, optional) –

\textbf{True} (default):

The timebase \texttt{ssobject.dt} cannot be None; it must be continuous (0) or discrete (True or > 0).

\textbf{False}:

If \texttt{ssobject.dt} is None, continuous time \texttt{scipy.signal.lti} objects are returned.

**Returns**

\texttt{out} – continuous time (inheriting from \texttt{scipy.signal.lti}) or discrete time (inheriting from \texttt{scipy.signal.dlti}) SISO objects

**Return type**

list of list of \texttt{scipy.signal.StateSpace}

reverse(zflag, params)

Compute the states and input given the flat flag.

See \texttt{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

```python
>>> 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]`.

**slycot_laub** *(x)*

Evaluate system’s transfer function at complex frequency using Laub’s method from Slycot.

Expects inputs and outputs to be formatted correctly. Use `sys(x)` for a more user-friendly interface.
Parameters

\[ x \text{ (complex array_like or complex)} \] – Complex frequency

Returns

\[ \text{output} \] – Frequency response

Return type

\( \text{(number_outputs, number_inputs, len(x)) complex ndarray} \)

property states

\[ \text{zeros}() \]

Compute the zeros of a state space system.

6.4.5 control.flatsys.PolyFamily

class control.flatsys.PolyFamily(N)

Bases: BasisFamily

Polynomial basis functions.

This class represents the family of polynomials of the form

\[ \phi_i(t) = t^i \]

Methods

\[ \text{eval_deriv} \]

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

\[ \text{__call__}(i, t) \]

Evaluate the ith basis function at a point in time

\[ \text{eval_deriv}(i, k, t) \]

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

6.4.6 control.flatsys.SystemTrajectory

class control.flatsys.SystemTrajectory(sys, basis, coeffs=[], flaglen=[], params=None)

Bases: object

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.

Parameters

- \[ \text{sys (FlatSystem)} \] – Flat system object associated with this trajectory.
- \[ \text{basis (BasisFamily)} \] – Family of basis vectors to use to represent the trajectory.
- \[ \text{coeffs (list of 1D arrays, optional)} \] – For each flat output, define the coefficients of the basis functions used to represent the trajectory. Defaults to an empty list.
- \[ \text{flaglen (list of ints, optional)} \] – For each flat output, the number of derivatives of the flat output used to define the trajectory. Defaults to an empty list.
# Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>eval</code></td>
<td>Return the state and input for a trajectory at a list of times.</td>
</tr>
<tr>
<td><code>response</code></td>
<td>Return the trajectory of a system as a TimeResponseData object</td>
</tr>
</tbody>
</table>

## `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:

\[
x, u = \text{traj.eval}(tlist)
\]

**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.

## `response(tlist, transpose=False, return_x=False, squeeze=None)`

Return the trajectory of a system as a TimeResponseData object

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

\[
\text{response} = \text{traj.response}(tlist) \\
\text{time, yd, ud} = \text{response}.\text{time}, \text{response}.\text{outputs}, \text{response}.\text{inputs}
\]

**Parameters**

- **tlist** (*1D array*) – List of times to evaluate the trajectory.
- **transpose** (*bool*, *optional*) – If True, transpose all input and output arrays (for backward compatibility with MATLAB and `scipy.signal.lsim()`). Default value is False.
- **return_x** (*bool*, *optional*) – If True, return the state vector when assigning to a tuple (default = False). See `forced_response()` for more details.
- **squeeze** (*bool*, *optional*) – By default, if a system is single-input, single-output (SISO) then the output response is returned as a 1D array (indexed by time). If `squeeze=True`, remove single-dimensional entries from the shape of the output even if the system is not SISO. If `squeeze=False`, keep the output as a 3D array (indexed by the output, input, and time) even if the system is SISO. The default value can be set using `config.defaults[‘control.squeeze_time_response’]`.

**Returns**

**results** – Time response represented as a TimeResponseData object containing the following properties:

- **time** (array): Time values of the output.
- **outputs** (array): Response of the system. If the system is SISO and `squeeze` is not True, the array is 1D (indexed by time). If the system is not SISO or `squeeze` is False, the array is 3D (indexed by the output, trace, and time).
- **states** (array): Time evolution of the state vector, represented as either a 2D array indexed by state and time (if SISO) or a 3D array indexed by state, trace, and time. Not affected by `squeeze`. 

• inputs (array): Input(s) to the system, indexed in the same manner as outputs.

The return value of the system can also be accessed by assigning the function to a tuple of length 2 (time, output) or of length 3 (time, output, state) if return_x is True.

**Return type**

*TimeResponseData*

<table>
<thead>
<tr>
<th>function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>point_to_point(sys, timepts[, x0, u0, xf, ...])</td>
<td>Compute trajectory between an initial and final conditions.</td>
</tr>
</tbody>
</table>
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:

```python
t, y = ct.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:

```python
x eq, ueq = ct.find_eqpt(io_sys, X0, U0)
ss_sys = ct.linearize(io_sys, x eq, ueq)
```

Input/output systems are automatically created for state space LTI systems when using the `ss()` function. 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 an output function (computes the outputs from the state):

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

More complex input/output systems can be constructed by using the `interconnect()` function, 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:

```python
steering = ct.interconnect(
    [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

```python
import control as ct
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:

```python
io_predprey = ct.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:

```python
X0 = [25, 20]            # Initial H, L
T = np.linspace(0, 70, 500) # Simulation 70 years of time

# Simulate the system
t, y = ct.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'])
```

(continues on next page)
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.

```python
plt.show(block=False)
```

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):

```python
eqpt = ct.find_eqpt(io_predprey, X0, 0)
xeq = eqpt[0]  # choose the nonzero equilibrium point
lin_predprey = ct.linearize(io_predprey, xeq, 0)
```

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:

```python
io_controller = ct.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` using the `connect()` function:

```python
io_closed = ct.interconnect(
    [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:

```python
# Simulate the system
t, y = ct.input_output_response(io_closed, T, 30, [15, 20])
```

```python
# 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 Additional features

The I/O systems module has a number of other features that can be used to simplify the creation of interconnected input/output systems.

7.3.1 Summing junction

The `summing_junction()` function can be used to create an input/output system that takes the sum of an arbitrary number of inputs. For example, to create an input/output system that takes the sum of three inputs, use the command

```python
sumblk = ct.summing_junction(3)
```

By default, the name of the inputs will be of the form u[i] and the output will be y. This can be changed by giving an explicit list of names:

```python
sumblk = ct.summing_junction(inputs=['a', 'b', 'c'], output='d')
```

A more typical usage would be to define an input/output system that compares a reference signal to the output of the process and computes the error:

```python
sumblk = ct.summing_junction(inputs=['r', '-y'], output='e')
```

Note the use of the minus sign as a means of setting the sign of the input ‘y’ to be negative instead of positive.

It is also possible to define “vector” summing blocks that take multi-dimensional inputs and produce a multi-dimensional output. For example, the command

```python
sumblk = ct.summing_junction(inputs=['r', '-y'], output='e', dimension=2)
```

will produce an input/output block that implements e[0] = r[0] - y[0] and e[1] = r[1] - y[1].

7.3.2 Automatic connections using signal names

The `interconnect()` function allows the interconnection of multiple systems by using signal names of the form `sys.signal`. In many situations, it can be cumbersome to explicitly connect all of the appropriate inputs and outputs. As an alternative, if the `connections` keyword is omitted, the `interconnect()` function will connect all signals of the same name to each other. This can allow for simplified methods of interconnecting systems, especially when combined with the `summing_junction()` function. For example, the following code will create a unity gain, negative feedback system:

```python
P = ct.tf2io([1], [1, 0], inputs='u', outputs='y')
C = ct.tf2io([10], [1, 1], inputs='e', outputs='u')
sumblk = ct.summing_junction(inputs=['r', '-y'], output='e')
T = ct.interconnect([P, C, sumblk], inplist='r', outlist='y')
```
If a signal name appears in multiple outputs then that signal will be summed when it is interconnected. Similarly, if a signal name appears in multiple inputs then all systems using that signal name will receive the same input. The `interconnect()` function will generate an error if an signal listed in `inplist` or `outlist` (corresponding to the inputs and outputs of the interconnected system) is not found, but inputs and outputs of individual systems that are not connected to other systems are left unconnected (so be careful!).

### 7.4 Module classes and functions

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>InputOutputSystem([params])</code></td>
<td>A class for representing input/output systems.</td>
</tr>
<tr>
<td><code>InterconnectedSystem(syslist[, connections,...])</code></td>
<td>Interconnection of a set of input/output systems.</td>
</tr>
<tr>
<td><code>LinearICSystem(io_sys[, ss_sys])</code></td>
<td>Interconnection of a set of linear input/output systems.</td>
</tr>
<tr>
<td><code>LinearIOSystem(linsys, **kwargs)</code></td>
<td>Input/output representation of a linear (state space) system.</td>
</tr>
<tr>
<td><code>NonlinearIOSystem(updfcn[, outfcn, params])</code></td>
<td>Nonlinear I/O system.</td>
</tr>
</tbody>
</table>

#### 7.4.1 `control.InterconnectedSystem`  

**class** `control.InterconnectedSystem(syslist, connections=[], inplist=[], outlist=[], params={}, warn_duplicate=None, **kwargs)`

**Bases:** `InputOutputSystem`

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.

See `interconnect()` for a list of parameters.
Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>check_unused_signals</code></td>
<td>Check for unused subsystem inputs and outputs</td>
</tr>
<tr>
<td><code>copy</code></td>
<td>Make a copy of an input/output system</td>
</tr>
<tr>
<td><code>dynamics</code></td>
<td>Compute the dynamics of a differential or difference equation.</td>
</tr>
<tr>
<td><code>feedback</code></td>
<td>Feedback interconnection between two input/output systems</td>
</tr>
<tr>
<td><code>find_input</code></td>
<td>Find the index for an input given its name (None if not found)</td>
</tr>
<tr>
<td><code>find_output</code></td>
<td>Find the index for an output given its name (None if not found)</td>
</tr>
<tr>
<td><code>find_state</code></td>
<td>Find the index for a state given its name (None if not found)</td>
</tr>
<tr>
<td><code>isctime</code></td>
<td>Check to see if a system is a continuous-time system</td>
</tr>
<tr>
<td><code>isdtime</code></td>
<td>Check to see if a system is a discrete-time system</td>
</tr>
<tr>
<td><code>issiso</code></td>
<td>Check to see if a system is single input, single output</td>
</tr>
<tr>
<td><code>linearize</code></td>
<td>Linearize an input/output system at a given state and input.</td>
</tr>
<tr>
<td><code>output</code></td>
<td>Compute the output of the system</td>
</tr>
<tr>
<td><code>set_connect_map</code></td>
<td>Set the connection map for an interconnected I/O system.</td>
</tr>
<tr>
<td><code>set_input_map</code></td>
<td>Set the input map for an interconnected I/O system.</td>
</tr>
<tr>
<td><code>set_inputs</code></td>
<td>Set the number/names of the system inputs.</td>
</tr>
<tr>
<td><code>set_output_map</code></td>
<td>Set the output map for an interconnected I/O system.</td>
</tr>
<tr>
<td><code>set_outputs</code></td>
<td>Set the number/names of the system outputs.</td>
</tr>
<tr>
<td><code>set_states</code></td>
<td>Set the number/names of the system states.</td>
</tr>
<tr>
<td><code>unused_signals</code></td>
<td>Find unused subsystem inputs and outputs</td>
</tr>
</tbody>
</table>

__add__(sys2)

Add two input/output systems (parallel interconnection)

__mul__(sys1)

Multiply two input/output systems (series interconnection)

__neg__()

Negate an input/output systems (rescale)

__radd__(sys2)

Parallel addition of input/output system to a compatible object.

__rmul__(sys2)

Pre-multiply an input/output systems by a scalar/matrix

__rsub__(sys2)

Parallel subtraction of I/O system to a compatible object.

__sub__(sys2)

Subtract two input/output systems (parallel interconnection)

```
check_unused_signals(ignore_inputs=None, ignore_outputs=None)
```

Check for unused subsystem inputs and outputs

If any unused inputs or outputs are found, emit a warning.
**Parameters**

- **ignore_inputs (list of input-spec)** –
  Subsystem inputs known to be unused. input-spec can be any of:
  'sig', 'sys.sig', (isys, isig), ('sys', isig)
  If the 'sig' form is used, all subsystem inputs with that name are considered ignored.

- **ignore_outputs (list of output-spec)** –
  Subsystem outputs known to be unused. output-spec can be any of:
  'sig', 'sys.sig', (isys, isig), ('sys', isig)
  If the 'sig' form is used, all subsystem outputs with that name are considered ignored.

**copy (name=None, use_prefix_suffix=True)**

Make a copy of an input/output system

A copy of the system is made, with a new name. The name keyword can be used to specify
a specific name for the system. If no name is given and use_prefix_suffix is True, the name is
constructed by prepending config.defaults['iosys.duplicate_system_name_prefix'] and appending con-
fig.defaults['iosys.duplicate_system_name_suffix']. Otherwise, a generic system name of the form
`sys[<id>]` is used, where `<id>` is based on an internal counter.

**dynamics (t, x, u)**

Compute the dynamics of a differential or difference equation.

Given time `t`, input `u` and state `x`, returns the value of the right hand side of the dynamical system. If the
system is continuous, returns the time derivative

\[
dx/dt = f(t, x, u)\]

where \( f \) is the system’s (possibly nonlinear) dynamics function. If the system is discrete-time, returns the
next value of `x`:

\[
x[t+dt] = f(t, x[t], u[t])
\]

Where \( t \) is a scalar.

The inputs `x` and `u` must be of the correct length.

**Parameters**

- **t (float)** – the time at which to evaluate
- **x (array_like)** – current state
- **u (array_like)** – input

**Returns**

\( dx/dt \) or \( x[t+dt] \)

**Return type**

ndarray

**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)

`isctime(strict=False)`

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

**Parameters**

• `sys (Named I/O 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

`linearize(x0, u0, t=0, params={}, eps=1e-06, name=None, copy=False, **kwargs)`

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.

`ninputs`

Number of system inputs.

`noutputs`

Number of system outputs.

`nstates`

Number of system states.
output \((t, x, u)\)

Compute the output of the system

Given time \(t\), input \(u\) and state \(x\), returns the output of the system:

\[ y = g(t, x, u) \]

The inputs \(x\) and \(u\) must be of the correct length.

**Parameters**

- \(t\) (float) – the time at which to evaluate
- \(x\) (array_like) – current state
- \(u\) (array_like) – input

**Returns**

- \(y\)

**Return type**

ndarray

**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]*.

**unused_signals**()

Find unused subsystem inputs and outputs

**Returns**

• **unused_inputs** (*dict*) – A mapping from tuple of indices (isys, isig) to string ‘{sys}.{sig}’, for all unused subsystem inputs.

• **unused_outputs** (*dict*) – A mapping from tuple of indices (isys, isig) to string ‘{sys}.{sig}’, for all unused subsystem outputs.

### 7.4.2 control.LinearICSystem

class control.LinearICSystem(*io_sys, ss_sys=None*)

Bases: InterconnectedSystem, LinearIOSystem

Interconnection of a set of linear input/output systems.

This class is used to implement a system that is an interconnection of linear input/output systems. It has all of the structure of an InterconnectedSystem, but also maintains the requirement elements of LinearIOSystem, including the StateSpace class structure, allowing it to be passed to functions that expect a StateSpace system.

This class is generated using interconnect() and not called directly.

**Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>append</td>
<td>Append a second model to the present model.</td>
</tr>
<tr>
<td>check_unused_signals</td>
<td>Check for unused subsystem inputs and outputs</td>
</tr>
<tr>
<td>copy</td>
<td>Make a copy of an input/output system</td>
</tr>
<tr>
<td>damp</td>
<td>Natural frequency, damping ratio of system poles</td>
</tr>
<tr>
<td>dcgain</td>
<td>Return the zero-frequency gain</td>
</tr>
<tr>
<td>dynamics</td>
<td>Compute the dynamics of a differential or difference equation.</td>
</tr>
<tr>
<td>feedback</td>
<td>Feedback interconnection between two input/output systems</td>
</tr>
</tbody>
</table>

continues on next page
### Table 1 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>find_input</code></td>
<td>Find the index for an input given its name (None if not found)</td>
</tr>
<tr>
<td><code>find_output</code></td>
<td>Find the index for an output given its name (None if not found)</td>
</tr>
<tr>
<td><code>find_state</code></td>
<td>Find the index for a state given its name (None if not found)</td>
</tr>
<tr>
<td><code>freqresp</code></td>
<td>(deprecated) Evaluate transfer function at complex frequencies.</td>
</tr>
<tr>
<td><code>frequency_response</code></td>
<td>Evaluate the linear time-invariant system at an array of angular frequencies.</td>
</tr>
<tr>
<td><code>horner</code></td>
<td>Evaluate system's transfer function at complex frequency using Laub's or Horner's method.</td>
</tr>
<tr>
<td><code>isctime</code></td>
<td>Check to see if a system is a continuous-time system</td>
</tr>
<tr>
<td><code>isdtime</code></td>
<td>Check to see if a system is a discrete-time system</td>
</tr>
<tr>
<td><code>issiso</code></td>
<td>Check to see if a system is single input, single output</td>
</tr>
<tr>
<td><code>lft</code></td>
<td>Return the Linear Fractional Transformation.</td>
</tr>
<tr>
<td><code>linearize</code></td>
<td>Linearize an input/output system at a given state and input.</td>
</tr>
<tr>
<td><code>minreal</code></td>
<td>Calculate a minimal realization, removes unobservable and uncontrollable states</td>
</tr>
<tr>
<td><code>output</code></td>
<td>Compute the output of the system</td>
</tr>
<tr>
<td><code>pole</code></td>
<td>Compute the poles of a state space system.</td>
</tr>
<tr>
<td><code>poles</code></td>
<td>Compute the poles of a state space system.</td>
</tr>
<tr>
<td><code>returnScipySignalLTI</code></td>
<td>Return a list of a list of <code>scipy.signal.lti</code> objects.</td>
</tr>
<tr>
<td><code>sample</code></td>
<td>Convert a continuous time system to discrete time</td>
</tr>
<tr>
<td><code>set_connect_map</code></td>
<td>Set the connection map for an interconnected I/O system.</td>
</tr>
<tr>
<td><code>set_input_map</code></td>
<td>Set the input map for an interconnected I/O system.</td>
</tr>
<tr>
<td><code>set_inputs</code></td>
<td>Set the number/names of the system inputs.</td>
</tr>
<tr>
<td><code>set_output_map</code></td>
<td>Set the output map for an interconnected I/O system.</td>
</tr>
<tr>
<td><code>set_outputs</code></td>
<td>Set the number/names of the system outputs.</td>
</tr>
<tr>
<td><code>set_states</code></td>
<td>Set the number/names of the system states.</td>
</tr>
<tr>
<td><code>slycot_laub</code></td>
<td>Evaluate system's transfer function at complex frequency using Laub's method from Slycot.</td>
</tr>
<tr>
<td><code>unused_signals</code></td>
<td>Find unused subsystem inputs and outputs</td>
</tr>
<tr>
<td><code>zero</code></td>
<td>Compute the zeros of a state space system.</td>
</tr>
</tbody>
</table>

**A**

Dynamics matrix.

**B**

Input matrix.

**C**

Output matrix.
D

Direct term.

__add__(sys2)
Add two input/output systems (parallel interconnection)

__call__(x, squeeze=None, warn_infinite=True)
Evaluate system’s transfer function at complex frequency.

Returns the complex frequency response \( \text{sys}(x) \) where \( x \) is \( s \) for continuous-time systems and \( z \) for discrete-time systems.

To evaluate at a frequency \( \omega \) in radians per second, enter \( x = \omega \times 1j \), for continuous-time systems, or \( x = \exp(1j \times \omega \times dt) \) for discrete-time systems. Or use StateSpace.

Parameters

- \( x \) (complex or complex 1D array_like) – Complex frequencies
- \( \text{squeeze} \) (bool, optional) – If \( \text{squeeze} = \text{True} \), remove single-dimensional entries from the shape of the output even if the system is not SISO. If \( \text{squeeze} = \text{False} \), keep all indices (output, input and, if \( \omega \) is array_like, frequency) even if the system is SISO. The default value can be set using config.defaults[‘control.squeeze_frequency_response’].
- \( \text{warn_infinite} \) (bool, optional) – If set to \( \text{False} \), don’t warn if frequency response is infinite.

Returns

\( \text{fresp} \) – The frequency response of the system. If the system is SISO and \( \text{squeeze} \) is not \( \text{True} \), the shape of the array matches the shape of \( \omega \). If the system is not SISO or \( \text{squeeze} \) is \( \text{False} \), the first two dimensions of the array are indices for the output and input and the remaining dimensions match \( \omega \). If \( \text{squeeze} \) is \( \text{True} \) then single-dimensional axes are removed.

Return type

complex ndarray

__div__(other)
Divide two LTI systems.

__getitem__(indices)
Array style access

__mul__(sys1)
Multiply two input/output systems (series interconnection)

__neg__()
Negate an input/output systems (rescale)

__radd__(sys2)
Parallel addition of input/output system to a compatible object.

__rdiv__(other)
Right divide two LTI systems.

__rmul__(sys2)
Pre-multiply an input/output systems by a scalar/matrix
Parallel subtraction of I/O system to a compatible object.

Subtract two input/output systems (parallel interconnection)

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

Check for unused subsystem inputs and outputs

If any unused inputs or outputs are found, emit a warning.

Make a copy of an input/output system

A copy of the system is made, with a new name. The name keyword can be used to specify a specific name for the system. If no name is given and use_prefix_suffix is True, the name is constructed by prepending config.defaults['iosys.duplicate_system_name_prefix'] and appending config.defaults['iosys.duplicate_system_name_suffix']. Otherwise, a generic system name of the form sys[<id>] is used, where <id> is based on an internal counter.

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

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:

Parameters

- warn_infinite (bool, optional) – By default, don’t issue a warning message if the zero-frequency gain is infinite. Setting warn_infinite to generate the warning message.
Returns

gain – Array or scalar value for SISO systems, depending on config.defaults['control.squeeze_frequency_response']. The value of the array elements or the scalar is either the zero-frequency (or DC) gain, or \( \text{inf} \), if the frequency response is singular.

For real valued systems, the empty imaginary part of the complex zero-frequency response is discarded and a real array or scalar is returned.

Return type
(noutputs, ninputs) ndarray or scalar

dynamics\((t, x, u)\)

Compute the dynamics of a differential or difference equation.

Given time \( t \), input \( u \) and state \( x \), returns the value of the right hand side of the dynamical system. If the system is continuous, returns the time derivative

\[
dx/dt = f(t, x, u)
\]

where \( f \) is the system’s (possibly nonlinear) dynamics function. If the system is discrete-time, returns the next value of \( x \):

\[
x[t+dt] = f(t, x[t], u[t])
\]

Where \( t \) is a scalar.

The inputs \( x \) and \( u \) must be of the correct length.

Parameters

• \( t \) (float) – the time at which to evaluate
• \( x \) (array_like) – current state
• \( u \) (array_like) – input

Returns

\( dx/dt \) or \( x[t+dt] \)

Return type

ndarray

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

Feedback interconnection between two input/output systems

Parameters

• \( \text{sys1} \) (InputOutputSystem) – The primary process.
• \( \text{sys2} \) (InputOutputSystem) – The feedback process (often a feedback controller).
• \( \text{sign} \) (scalar, optional) – The sign of feedback. \( \text{sign} = -1 \) indicates negative feedback, and \( \text{sign} = 1 \) indicates positive feedback. \( \text{sign} \) 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)*

*(deprecated)* Evaluate transfer function at complex frequencies.

**frequency_response** *(omega, squeeze=None)*

Evaluate the linear time-invariant system at an array of angular frequencies.

Reports the frequency response of the system,

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

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

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

In general the system may be multiple input, multiple output (MIMO), where \( m = \text{self.ninputs} \) number of inputs and \( p = \text{self.noutputs} \) number of outputs.

**Parameters**

- **omega** *(float or 1D array_like)* – A list, tuple, array, or scalar value of frequencies in radians/sec at which the system will be evaluated.

- **squeeze** *(bool, optional)* – If squeeze=True, remove single-dimensional entries from the shape of the output even if the system is not SISO. If squeeze=False, keep all indices (output, input and, if omega is array_like, frequency) even if the system is SISO. The default value can be set using config.defaults[‘control.squeeze_frequency_response’].

**Returns**

**response** – Frequency response data object representing the frequency response. This object can be assigned to a tuple using

\[ \text{mag}, \text{phase}, \omega = \text{response} \]

where \( \text{mag} \) is the magnitude (absolute value, not dB or log10) of the system frequency response, \( \text{phase} \) is the wrapped phase in radians of the system frequency response, and \( \omega \) is the (sorted) frequencies at which the response was evaluated. If the system is SISO and squeeze is not True, magnitude and phase are 1D, indexed by frequency. If the system is not SISO or squeeze is False, the array is 3D, indexed by the output, input, and frequency. If squeeze is True then single-dimensional axes are removed.

**Return type**

FrequencyReponseData

**horner** *(x, warn_infinite=True)*

Evaluate system’s transfer function at complex frequency using Laub’s or Horner’s method.

Evaluates \( \text{sys}(x) \) where \( x \) is \( s \) for continuous-time systems and \( z \) for discrete-time systems.

Expects inputs and outputs to be formatted correctly. Use \( \text{sys}(x) \) for a more user-friendly interface.

**Parameters**

- **x** *(complex array_like or complex)* – Complex frequencies
Returns

    output – Frequency response

Return type

    (self.noutputs, self.ninputs, len(x)) complex ndarray

Notes

Attempts to use Laub’s method from Slycot library, with a fall-back to python code.

property inputs

    Deprecated attribute; use ninputs instead.

The inputs attribute was used to store the number of system inputs. It is no longer used. If you need access to the number of inputs for an LTI system, use ninputs.

isctime(strict=False)

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

Parameters

    • sys (Named I/O system) – System to be checked
    • strict (bool, optional) – If strict is True, make sure that timebase is not None. Default is False.

isdtme(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, name=None, copy=False, **kwargs)

    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
ninputs
Number of system inputs.

noutputs
Number of system outputs.

nstates
Number of system states.

output(t, x, u)
Compute the output of the system
Given time t, input u and state x, returns the output of the system:
y = g(t, x, u)
The inputs x and u must be of the correct length.

Parameters
• t (float) – the time at which to evaluate
• x (array_like) – current state
• u (array_like) – input

Returns
y

Return type
ndarray

property outputs
Deprecated attribute; use noutputs instead.
The outputs attribute was used to store the number of system outputs. It is no longer used. If you need access to the number of outputs for an LTI system, use noutputs.

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

returnScipySignalLTI(strict=True)
Return a list of a list of scipy.signal.lti objects.
For instance,

```python
>>> 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.

Parameters
strict (bool, optional) –
True (default):
The timebase ssobject.dt cannot be None; it must be continuous (0) or discrete (True or > 0).
False:
If ssobject.dt is None, continuous time scipy.signal.lti objects are returned.

Returns
out – continuous time (inheriting from scipy.signal.lti) or discrete time (inheriting from scipy.signal.dlti) SISO objects

Return type
list of list of scipy.signal.StateSpace

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

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

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 $prefix[i]$.

`slycot_laub(x)`

Evaluate system’s transfer function at complex frequency using Laub’s method from Slycot.

Expects inputs and outputs to be formatted correctly. Use `sys(x)` for a more user-friendly interface.

**Parameters**

- `x` *(complex array_like or complex)* – Complex frequency

**Returns**

- `output` – Frequency response

**Return type**

(number_outputs, number_inputs, len(x)) complex ndarray

`property states`  

Deprecated attribute; use `nstates` instead.

The state attribute was used to store the number of states for : a state space system. It is no longer used. If you need to access the number of states, use `nstates`.

`unused_signals()`  

Find unused subsystem inputs and outputs

**Returns**

- **unused_inputs** *(dict)* – A mapping from tuple of indices (isys, isig) to string ‘{sys}.{sig}’, for all unused subsystem inputs.

- **unused_outputs** *(dict)* – A mapping from tuple of indices (isys, isig) to string ‘{sys}.{sig}’, for all unused subsystem outputs.

`zeros()`  

Compute the zeros of a state space system.

### 7.4.3 control.LinearIOSystem

**class control.LinearIOSystem(linsys, **kwargs)**

Bases: `InputOutputSystem`, `StateSpace`

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

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

**Parameters**

- **linsys** *(StateSpace or TransferFunction)* – LTI 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. 0 (default) indicates continuous time, True indicates discrete time with unspecified sampling time, positive number is discrete time with specified sampling time, None indicates unspecified timebase (either continuous or discrete time).

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

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

*ninputs, noutputs, nstates, dt, etc*

See *InputOutputSystem* for inherited attributes.

*A, B, C, D*

See *StateSpace* for inherited attributes.
## Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>append</strong></td>
<td>Append a second model to the present model.</td>
</tr>
<tr>
<td><strong>copy</strong></td>
<td>Make a copy of an input/output system</td>
</tr>
<tr>
<td><strong>damp</strong></td>
<td>Natural frequency, damping ratio of system poles</td>
</tr>
<tr>
<td><strong>dcgain</strong></td>
<td>Return the zero-frequency gain</td>
</tr>
<tr>
<td><strong>dynamics</strong></td>
<td>Compute the dynamics of a differential or difference equation.</td>
</tr>
<tr>
<td><strong>feedback</strong></td>
<td>Feedback interconnection between two input/output systems</td>
</tr>
<tr>
<td><strong>find_input</strong></td>
<td>Find the index for an input given its name (None if not found)</td>
</tr>
<tr>
<td><strong>find_output</strong></td>
<td>Find the index for an output given its name (None if not found)</td>
</tr>
<tr>
<td><strong>find_state</strong></td>
<td>Find the index for a state given its name (None if not found)</td>
</tr>
<tr>
<td><strong>freqresp</strong></td>
<td>(deprecated) Evaluate transfer function at complex frequencies.</td>
</tr>
<tr>
<td><strong>frequency_response</strong></td>
<td>Evaluate the linear time-invariant system at an array of angular frequencies.</td>
</tr>
<tr>
<td><strong>horner</strong></td>
<td>Evaluate system's transfer function at complex frequency using Laub's or Horner's method.</td>
</tr>
<tr>
<td><strong>isctime</strong></td>
<td>Check to see if a system is a continuous-time system</td>
</tr>
<tr>
<td><strong>isdtime</strong></td>
<td>Check to see if a system is a discrete-time system</td>
</tr>
<tr>
<td><strong>issiso</strong></td>
<td>Check to see if a system is single input, single output</td>
</tr>
<tr>
<td><strong>lft</strong></td>
<td>Return the Linear Fractional Transformation.</td>
</tr>
<tr>
<td><strong>linearize</strong></td>
<td>Linearize an input/output system at a given state and input.</td>
</tr>
<tr>
<td><strong>minreal</strong></td>
<td>Calculate a minimal realization, removes unobservable and uncontrollable states</td>
</tr>
<tr>
<td><strong>output</strong></td>
<td>Compute the output of the system</td>
</tr>
<tr>
<td><strong>pole</strong></td>
<td></td>
</tr>
<tr>
<td><strong>poles</strong></td>
<td>Compute the poles of a state space system.</td>
</tr>
<tr>
<td><strong>returnScipySignalLTI</strong></td>
<td>Return a list of a list of scipy.signal.lti objects.</td>
</tr>
<tr>
<td><strong>sample</strong></td>
<td>Convert a continuous time system to discrete time</td>
</tr>
<tr>
<td><strong>set_inputs</strong></td>
<td>Set the number/names of the system inputs.</td>
</tr>
<tr>
<td><strong>set_outputs</strong></td>
<td>Set the number/names of the system outputs.</td>
</tr>
<tr>
<td><strong>set_states</strong></td>
<td>Set the number/names of the system states.</td>
</tr>
<tr>
<td><strong>slycot_laub</strong></td>
<td>Evaluate system's transfer function at complex frequency using Laub's method from Slycot.</td>
</tr>
<tr>
<td><strong>zero</strong></td>
<td></td>
</tr>
<tr>
<td><strong>zeros</strong></td>
<td>Compute the zeros of a state space system.</td>
</tr>
</tbody>
</table>

### Symbols

- **A**
  - Dynamics matrix.

- **B**
  - Input matrix.
C
Output matrix.

D
Direct term.

`__add__`(sys2)
Add two input/output systems (parallel interconnection)

`__call__`(x, squeeze=None, warn_infinite=True)
Evaluate system’s transfer function at complex frequency.

Returns the complex frequency response \( \text{sys}(x) \) where \( x \) is \( s \) for continuous-time systems and \( z \) for discrete-time systems.

To evaluate at a frequency \( \omega \) in radians per second, enter \( x = \omega \cdot 1j \), for continuous-time systems, or \( x = \exp(1j \cdot \omega \cdot dt) \) for discrete-time systems. Or use `StateSpace.frequency_response()`.

Parameters

- **x** *(complex or complex 1D array_like)* – Complex frequencies
- **squeeze** *(bool, optional)* – If `squeeze=True`, remove single-dimensional entries from the shape of the output even if the system is not SISO. If `squeeze=False`, keep all indices (output, input and, if omega is array_like, frequency) even if the system is SISO. The default value can be set using config.defaults[‘control.squeeze_frequency_response’].
- **warn_infinite** *(bool, optional)* – If set to `False`, don’t warn if frequency response is infinite.

Returns

- **fresp** – The frequency response of the system. If the system is SISO and squeeze is not True, the shape of the array matches the shape of \( \omega \). If the system is not SISO or squeeze is False, the first two dimensions of the array are indices for the output and input and the remaining dimensions match \( \omega \). If `squeeze` is True then single-dimensional axes are removed.

Return type

- **complex ndarray**

`__div__`(other)
Divide two LTI systems.

`__getitem__`(indices)
Array style access

`__mul__`(sys1)
Multiply two input/output systems (series interconnection)

`__neg__`()  
Negate an input/output systems (rescale)

`__radd__`(sys2)
Parallel addition of input/output system to a compatible object.
__rdiv__(other)
Right divide two LTI systems.

__rmul__(sys2)
Pre-multiply an input/output systems by a scalar/matrix

__rsub__(sys2)
Parallel subtraction of I/O system to a compatible object.

__sub__(sys2)
Subtract two input/output systems (parallel interconnection)

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(name=None, use_prefix_suffix=True)
Make a copy of an input/output system

A copy of the system is made, with a new name. The name keyword can be used to specify a specific name for the system. If no name is given and use_prefix_suffix is True, the name is constructed by prepending config.defaults['iosys.duplicate_system_name_prefix'] and appending config.defaults['iosys.duplicate_system_name_suffix']. Otherwise, a generic system name of the form sys[<id>] is used, where <id> is based on an internal counter.

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(warn_infinite=False)
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:

Parameters

- warn_infinite (bool, optional) – By default, don’t issue a warning message if the zero-frequency gain is infinite. Setting warn_infinite to generate the warning message.

Returns

- gain – Array or scalar value for SISO systems, depending on config.defaults['control.squeeze_frequency_response']. The value of the array elements or the scalar is either the zero-frequency (or DC) gain, or inf, if the frequency response is singular.

For real valued systems, the empty imaginary part of the complex zero-frequency response is discarded and a real array or scalar is returned.

Return type
(noutputs, ninputs) ndarray or scalar
dynamics(t, x, u)

Compute the dynamics of a differential or difference equation.

Given time t, input u and state x, returns the value of the right hand side of the dynamical system. If the system is continuous, returns the time derivative

\[ \frac{dx}{dt} = f(t, x, u) \]

where \( f \) is the system’s (possibly nonlinear) dynamics function. If the system is discrete-time, returns the next value of x:

\[ x[t+dt] = f(t, x[t], u[t]) \]

Where t is a scalar.

The inputs x and u must be of the correct length.

Parameters

- t (float) – the time at which to evaluate
- x (array_like) – current state
- u (array_like) – input

Returns

dx/dt or x[t+dt]

Return type

ndarray

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)

(deprecated) Evaluate transfer function at complex frequencies.

7.4. Module classes and functions 213
**frequency_response**(*omega, squeeze=None*)

Evaluate the linear time-invariant system at an array of angular frequencies.

Reports the frequency response of the system,

\[ G(j\omega) = \text{mag} \times \exp(j\text{phase}) \]

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

\[ G(\exp(j\omega dt)) = \text{mag} \times \exp(j\text{phase}). \]

In general the system may be multiple input, multiple output (MIMO), where \( m = \text{self.ninputs} \) number of inputs and \( p = \text{self.noutputs} \) number of outputs.

**Parameters**

- *omega* (*float or 1D array_like*) – A list, tuple, array, or scalar value of frequencies in radians/sec at which the system will be evaluated.

- *squeeze* (*bool, optional*) – If squeeze=True, remove single-dimensional entries from the shape of the output even if the system is not SISO. If squeeze=False, keep all indices (output, input, and if omega is array_like, frequency) even if the system is SISO. The default value can be set using config.defaults['control.squeeze_frequency_response'].

**Returns**

- *response* – Frequency response data object representing the frequency response. This object can be assigned to a tuple using

  \[ \text{mag}, \text{phase}, \omega = \text{response} \]

  where \( \text{mag} \) is the magnitude (absolute value, not dB or log10) of the system frequency response, \( \text{phase} \) is the wrapped phase in radians of the system frequency response, and \( \omega \) is the (sorted) frequencies at which the response was evaluated. If the system is SISO and squeeze is not True, \( \text{magnitude} \) and \( \text{phase} \) are 1D, indexed by frequency. If the system is not SISO or squeeze is False, the array is 3D, indexed by the output, input, and frequency. If squeeze is True then single-dimensional axes are removed.

**Return type**

- *FrequencyResponseData*

**horner**(*x, warn_infinite=True*)

Evaluate system’s transfer function at complex frequency using Laub’s or Horner’s method.

Evaluates \( \text{sys}(x) \) where \( x \) is \( s \) for continuous-time systems and \( z \) for discrete-time systems.

Expects inputs and outputs to be formatted correctly. Use \( \text{sys}(x) \) for a more user-friendly interface.

**Parameters**

- *x* (*complex array_like or complex*) – Complex frequencies

**Returns**

- *output* – Frequency response

**Return type**

- *(self.noutputs, self.ninputs, len(x)) complex ndarray*
Notes

Attempts to use Laub’s method from Slycot library, with a fall-back to python code.

**property inputs**
Deprecate attribute; use `ninputs` instead.

The `inputs` attribute was used to store the number of system inputs. It is no longer used. If you need access to the number of inputs for an LTI system, use `ninputs`.

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

**Parameters**
- `sys` *(Named I/O system)* – System to be checked
- `strict` *(bool, optional)* – If strict is True, make sure that timebase is not None. Default is False.

**isdtme**(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
- `nu` *(int, optional)* – Dimension of (plant) control input.
- `ny` *(int, optional)* – Dimension of (plant) measurement output.

**linearize**(x0, u0, t=0, params={}, eps=1e-06, name=None, copy=False, **kwargs)
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

**ninputs**
Number of system inputs.
noutputs

Number of system outputs.

nstates

Number of system states.

output \((t, x, u)\)

Compute the output of the system

Given time \(t\), input \(u\) and state \(x\), returns the output of the system:

\[ y = g(t, x, u) \]

The inputs \(x\) and \(u\) must be of the correct length.

Parameters

- \(t\) (float) – the time at which to evaluate
- \(x\) (array_like) – current state
- \(u\) (array_like) – input

Returns

- \(y\)

Return type

ndarray

property outputs

Deprecated attribute; use noutputs instead.

The outputs attribute was used to store the number of system outputs. It is no longer used. If you need access to the number of outputs for an LTI system, use noutputs.

poles()

Compute the poles of a state space system.

returnScipySignalLTI (strict=True)

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.

Parameters

- strict (bool, optional) –

  True (default):
  - The timebase ssobject.dt cannot be None; it must be continuous (0) or discrete (True or > 0).

  False:
  - If ssobject.dt is None, continuous time scipy.signal.lti objects are returned.
Returns

out – continuous time (inheriting from `scipy.signal.lti`) or discrete time (inheriting from `scipy.signal.dlti`) SISO objects

Return type

list of list of `scipy.signal.StateSpace`

`sampate(Ts, method='zoh', alpha=None, prewarip_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

```python
>>> 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]`.

**slycot_laub** (*x*)

Evaluate system’s transfer function at complex frequency using Laub’s method from Slycot.

Expects inputs and outputs to be formatted correctly. Use `sys(x)` for a more user-friendly interface.

**Parameters**

• **x** (*complex array_like or complex*) – Complex frequency

**Returns**

• **output** – Frequency response

**Return type**

*(number_outputs, number_inputs, len(x)) complex ndarray*

**property states**

Deprecated attribute; use `nstates` instead.

The `state` attribute was used to store the number of states for a state space system. It is no longer used. If you need to access the number of states, use `nstates`.

**zeros** ()

Compute the zeros of a state space system.
7.4.4 control.NonlinearIOSystem

class control.NonlinearIOSystem(updfcn, outfcn=None, params={}, **kwargs)
    Bases: InputOutputSystem

Nonlinear I/O system.

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 are not yet supported by most functions.)

Parameters

- **updfcn** (callable) – Function returning the state update function
  
  \[
  \text{updfcn}(t, x, u, \text{params}) \to \text{array}
  \]

  where \(x\) is a 1-D array with shape \((\text{nstates},)\), \(u\) is a 1-D array with shape \((\text{ninputs},)\), \(t\) is a float representing the current time, and \(\text{params}\) is a dict containing the values of parameters used by the function.

- **outfcn** (callable) – Function returning the output at the given state
  
  \[
  \text{outfcn}(t, x, u, \text{params}) \to \text{array}
  \]

  where the arguments are the same as for \(\text{upfcn}\).

- **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 \(\text{None}\), 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 \(\text{inputs}\).

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

- **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:
  - \(\text{dt} = 0\): continuous time system (default)
  - \(\text{dt} > 0\): discrete time system with sampling period ‘\(\text{dt}\)’
  - \(\text{dt} = \text{True}\): discrete time with unspecified sampling period
  - \(\text{dt} = \text{None}\): no timebase specified

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

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

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>copy</td>
<td>Make a copy of an input/output system</td>
</tr>
<tr>
<td>dynamics</td>
<td>Compute the dynamics of a differential or difference equation.</td>
</tr>
<tr>
<td>feedback</td>
<td>Feedback interconnection between two input/output systems</td>
</tr>
<tr>
<td>find_input</td>
<td>Find the index for an input given its name (None if not found)</td>
</tr>
<tr>
<td>find_output</td>
<td>Find the index for an output given its name (None if not found)</td>
</tr>
<tr>
<td>find_state</td>
<td>Find the index for a state given its name (None if not found)</td>
</tr>
<tr>
<td>isctime</td>
<td>Check to see if a system is a continuous-time system</td>
</tr>
<tr>
<td>isdtime</td>
<td>Check to see if a system is a discrete-time system</td>
</tr>
<tr>
<td>issiso</td>
<td>Check to see if a system is single input, single output</td>
</tr>
<tr>
<td>linearize</td>
<td>Linearize an input/output system at a given state and input.</td>
</tr>
<tr>
<td>output</td>
<td>Compute the output of the system</td>
</tr>
<tr>
<td>set_inputs</td>
<td>Set the number/names of the system inputs.</td>
</tr>
<tr>
<td>set_outputs</td>
<td>Set the number/names of the system outputs.</td>
</tr>
<tr>
<td>set_states</td>
<td>Set the number/names of the system states.</td>
</tr>
</tbody>
</table>

__add__(sys2)
Add two input/output systems (parallel interconnection)

__call__(u, params=None, squeeze=None)
Evaluate a (static) nonlinearity at a given input value
If a nonlinear I/O system has no internal state, then evaluating the system at an input u gives the output y = F(u), determined by the output function.

Parameters

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

- **squeeze** (bool, optional) – If True and if the system has a single output, return the system output as a 1D array rather than a 2D array. If False, return the system output as a 2D array even if the system is SISO. Default value set by config.defaults['control.squeeze_time_response'].

__mul__(sys1)
Multiply two input/output systems (series interconnection)

__neg__()
Negate an input/output systems (rescale)

__radd__(sys2)
Parallel addition of input/output system to a compatible object.

__rmul__(sys2)
Pre-multiply an input/output systems by a scalar/matrix

__rsub__(sys2)
Parallel subtraction of I/O system to a compatible object.
__sub__ (sys2)
Subtract two input/output systems (parallel interconnection)

copy (name=None, use_prefix_suffix=True)
Make a copy of an input/output system

A copy of the system is made, with a new name. The name keyword can be used to specify
a specific name for the system. If no name is given and use_prefix_suffix is True, the name is
constructed by prepending config.defaults[‘iosys.duplicate_system_name_prefix’] and appending con-
fig.defaults[‘iosys.duplicate_system_name_suffix’]. Otherwise, a generic system name of the form
sys[<id>] is used, where <id> is based on an internal counter.

dynamics (t, x, u)
Compute the dynamics of a differential or difference equation.
Given time t, input u and state x, returns the value of the right hand side of the dynamical system. If the
system is continuous, returns the time derivative

\[ \frac{dx}{dt} = f(t, x, u) \]

where \( f \) is the system’s (possibly nonlinear) dynamics function. If the system is discrete-time, returns the
next value of \( x \):

\[ x[t+dt] = f(t, x[t], u[t]) \]

Where \( t \) is a scalar.

The inputs \( x \) and \( u \) must be of the correct length.

Parameters

- \( t \) (float) – the time at which to evaluate
- \( x \) (array_like) – current state
- \( u \) (array_like) – input

Returns

\( \frac{dx}{dt} \) or \( x[t+dt] \)

Return type

ndarray

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)

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

**Parameters**

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

**isdtune**(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

**linearize**(x0, u0, t=0, params={}, eps=1e-06, name=None, copy=False, **kwargs)
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.

**ninputs**
Number of system inputs.

**noutputs**
Number of system outputs.

**nstates**
Number of system states.

**output**(t, x, u)
Compute the output of the system

Given time t, input u and state x, returns the output of the system:

\[ y = g(t, x, u) \]

The inputs x and u must be of the correct length.

**Parameters**

- **t** *(float)* – the time at which to evaluate
• \( x \) (array_like) – current state
• \( u \) (array_like) – input

Returns

\( y \)

Return type

ndarray

\textbf{set_inputs}(inputs, prefix='u')

Set the number/names of the system inputs.

Parameters

• \texttt{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).

• \texttt{prefix} (string, optional) – If \texttt{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 \texttt{prefix}[i].

\textbf{set_outputs}(outputs, prefix='y')

Set the number/names of the system outputs.

Parameters

• \texttt{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).

• \texttt{prefix} (string, optional) – If \texttt{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 \texttt{prefix}[i].

\textbf{set_states}(states, prefix='x')

Set the number/names of the system states.

Parameters

• \texttt{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).

• \texttt{prefix} (string, optional) – If \texttt{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 \texttt{prefix}[i].

\begin{tabular}{ll}
\texttt{find_eqpt}(sys, x0[, u0, y0, t, params, iu, ...]) & Find the equilibrium point for an input/output system. \\
\texttt{linearize}(sys, xeq[, ueq, t, params]) & Linearize an input/output system at a given state and input. \\
\texttt{input_output_response}(sys, T[, U, X0, ...]) & Compute the output response of a system to a given input. \\
\texttt{interconnect}(syslist[, connections, ...]) & Interconnect a set of input/output systems. \\
\texttt{ss2io}(\ast\texttt{args, **kwargs}) & Create an I/O system from a state space linear system. \\
\texttt{summing_junction}([inputs, output, ...]) & Create a summing junction as an input/output system. \\
\texttt{tf2io}(sys) & Convert a transfer function into an I/O system. \\
\end{tabular}
For nonlinear systems consisting of a feedback connection between a linear system and a static nonlinearity, it is possible to obtain a generalization of Nyquist’s stability criterion based on the idea of describing functions. The basic concept involves approximating the response of a static nonlinearity to an input $u = Ae^{j\omega t}$ as an output $y = N(A)(Ae^{j\omega t})$, where $N(A) \in \mathbb{C}$ represents the (amplitude-dependent) gain and phase associated with the nonlinearity.

Stability analysis of a linear system $H(s)$ with a feedback nonlinearity $F(x)$ is done by looking for amplitudes $A$ and frequencies $\omega$ such that

$$H(j\omega)N(A) = -1$$

If such an intersection exists, it indicates that there may be a limit cycle of amplitude $A$ with frequency $\omega$.

Describing function analysis is a simple method, but it is approximate because it assumes that higher harmonics can be neglected.

### 8.1 Module usage

The function `describing_function()` can be used to compute the describing function of a nonlinear function:

```python
N = ct.describing_function(F, A)
```

Stability analysis using describing functions is done by looking for amplitudes $a$ and frequencies `$\omega$` such that

$$H(j\omega) = \frac{-1}{N(A)}$$

These points can be determined by generating a Nyquist plot in which the transfer function $H(j\omega)$ intersects the negative reciprocal of the describing function $N(A)$. The `describing_function_plot()` function generates this plot and returns the amplitude and frequency of any points of intersection:

```python
ct.describing_function_plot(H, F, amp_range[, omega_range])
```
8.2 Pre-defined nonlinearities

To facilitate the use of common describing functions, the following nonlinearity constructors are predefined:

- `friction_backlash_nonlinearity(b)`  # backlash nonlinearity with width b
- `relay_hysteresis_nonlinearity(b, c)`  # relay output of amplitude b with
  # hysteresis of half-width c
- `saturation_nonlinearity(ub[, lb])`  # saturation nonlinearity with upper
  # bound and (optional) lower bound

Calling these functions will create an object $F$ that can be used for describing function analysis. For example, to create a saturation nonlinearity:

```python
F = ct.saturation_nonlinearity(1)
```

These functions use the `DescribingFunctionNonlinearity`, which allows an analytical description of the describing function.

8.3 Module classes and functions

<table>
<thead>
<tr>
<th><code>DescribingFunctionNonlinearity()</code></th>
<th>Base class for nonlinear systems with a describing function.</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>friction_backlash_nonlinearity(b)</code></td>
<td>Backlash nonlinearity for describing function analysis.</td>
</tr>
<tr>
<td><code>relay_hysteresis_nonlinearity(b, c)</code></td>
<td>Relay w/ hysteresis nonlinearity for describing function analysis.</td>
</tr>
<tr>
<td><code>saturation_nonlinearity([ub, lb])</code></td>
<td>Create saturation nonlinearity for use in describing function analysis.</td>
</tr>
</tbody>
</table>

8.3.1 `control.DescribingFunctionNonlinearity`

class `control.DescribingFunctionNonlinearity`

Bases: `object`

Base class for nonlinear systems with a describing function.

This class is intended to be used as a base class for nonlinear functions that have an analytically defined describing function. Subclasses should override the `__call__` and `describing_function` methods and (optionally) the `_isstatic` method (should be `False` if `__call__` updates the instance state).

Methods

- `describing_function`  
  Return the describing function for a nonlinearity.

- `__call__(A)`  
  Evaluate the nonlinearity at a (scalar) input value.
describing_function(A)
    Return the describing function for a nonlinearity.

    This method is used to allow analytical representations of the describing function for a nonlinearity. It turns
the (complex) value of the describing function for sinusoidal input of amplitude A.

### 8.3.2 control.friction_backlash_nonlinearity

**class control.friction_backlash_nonlinearity(b)**

Bases: DescribingFunctionNonlinearity

Backlash nonlinearity for describing function analysis.

This class creates a nonlinear function representing a friction-dominated backlash nonlinearity, including the
describing function for the nonlinearity. The following call creates a nonlinear function suitable for describing
function analysis:

F = friction_backlash_nonlinearity(b)

This function maintains an internal state representing the ‘center’ of a mechanism with backlash. If the new input
is within \( b/2 \) of the current center, the output is unchanged. Otherwise, the output is given by the input shifted
by \( b/2 \).

**Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>call</strong>(x)</td>
<td>Evaluate the nonlinearity at a (scalar) input value.</td>
</tr>
<tr>
<td>describing_function(A)</td>
<td>Return the describing function for a nonlinearity.</td>
</tr>
</tbody>
</table>

    This method is used to allow analytical representations of the describing function for a nonlinearity. It turns
the (complex) value of the describing function for sinusoidal input of amplitude A.

### 8.3.3 control.relay_hysteresis_nonlinearity

**class control.relay_hysteresis_nonlinearity(b, c)**

Bases: DescribingFunctionNonlinearity

Relay w/ hysteresis nonlinearity for describing function analysis.

This class creates a nonlinear function representing a relay with symmetric upper and lower bounds of magnitude
\( b \) and a hysteretic region of width \( c \) (using the notation from [FBS2e](https://fbsbook.org), Example 10.12,
including the describing function for the nonlinearity. The following call creates a nonlinear function suitable
for describing function analysis:

F = relay_hysteresis_nonlinearity(b, c)

The output of this function is \( b \) if \( x > c \) and \(-b\) if \( x < -c \). For \(-c <= x <= c \), the value depends on the branch of
the hysteresis loop (as illustrated in Figure 10.20 of FBS2e).
Methods

```
describing_function
Return the describing function for a nonlinearity.
```

```
__call__(x)
Evaluate the nonlinearity at a (scalar) input value.
```

```
describing_function(A)
Return the describing function for a nonlinearity.

This method is used to allow analytical representations of the describing function for a nonlinearity. It turns
the (complex) value of the describing function for sinusoidal input of amplitude A.
```

### 8.3.4 control.saturation_nonlinearity

class control.saturation_nonlinearity(ub=1, lb=None)

Bases: DescribingFunctionNonlinearity

Create saturation nonlinearity for use in describing function analysis.

This class creates a nonlinear function representing a saturation with given upper and lower bounds, including the
describing function for the nonlinearity. The following call creates a nonlinear function suitable for describing
function analysis:

```
F = saturation_nonlinearity(ub[, lb])
```

By default, the lower bound is set to the negative of the upper bound. Asymmetric saturation functions can be
created, but note that these functions will not have zero bias and hence care must be taken in using the nonlinearity
for analysis.

Methods

```
describing_function
Return the describing function for a nonlinearity.
```

```
__call__(x)
Evaluate the nonlinearity at a (scalar) input value.
```

```
describing_function(A)
Return the describing function for a nonlinearity.

This method is used to allow analytical representations of the describing function for a nonlinearity. It turns
the (complex) value of the describing function for sinusoidal input of amplitude A.
The *optimal* module provides support for optimization-based controllers for nonlinear systems with state and input constraints.

### 9.1 Problem setup

Consider the optimal control problem:

\[
\min_{u(t)} \int_0^T L(x, u) \, dt + V(x(T))
\]

subject to the constraint

\[
\dot{x} = f(x, u), \quad x \in \mathbb{R}^n, \, u \in \mathbb{R}^m.
\]

Abstractly, this is a constrained optimization problem where we seek a feasible trajectory \((x(t), u(t))\) that minimizes the cost function

\[
J(x, u) = \int_0^T L(x, u) \, dt + V(x(T)).
\]

More formally, this problem is equivalent to the “standard” problem of minimizing a cost function \(J(x, u)\) where \((x, u) \in L_2[0, T]\) (the set of square integrable functions) and \(h(z) = \dot{x}(t) - f(x(t), u(t)) = 0\) models the dynamics. The term \(L(x, u)\) is referred to as the integral (or trajectory) cost and \(V(x(T))\) is the final (or terminal) cost.

It is often convenient to ask that the final value of the trajectory, denoted \(x_f\), be specified. We can do this by requiring that \(x(T) = x_f\) or by using a more general form of constraint:

\[
\psi_i(x(T)) = 0, \quad i = 1, \ldots, q.
\]

The fully constrained case is obtained by setting \(q = n\) and defining \(\psi_i(x(T)) = x_i(T) - x_{i,f}\). For a control problem with a full set of terminal constraints, \(V(x(T))\) can be omitted (since its value is fixed).

Finally, we may wish to consider optimizations in which either the state or the inputs are constrained by a set of nonlinear functions of the form

\[
l_i \leq g_i(x, u) \leq u_i, \quad i = 1, \ldots, k.
\]

where \(l_i\) and \(u_i\) represent lower and upper bounds on the constraint function \(g_i\). Note that these constraints can be on the input, the state, or combinations of input and state, depending on the form of \(g_i\). Furthermore, these constraints are intended to hold at all instants in time along the trajectory.

A common use of optimization-based control techniques is the implementation of model predictive control (also called receding horizon control). In model predictive control, a finite horizon optimal control problem is solved, generating
open-loop state and control trajectories. The resulting control trajectory is applied to the system for a fraction of the horizon length. This process is then repeated, resulting in a sampled data feedback law. This approach is illustrated in the following figure:

Every $\Delta T$ seconds, an optimal control problem is solved over a $T$ second horizon, starting from the current state. The first $\Delta T$ seconds of the optimal control $u_T(\cdot; x(t))$ is then applied to the system. If we let $x_T(\cdot; x(t))$ represent the optimal trajectory starting from $x(t)$ then the system state evolves from $x(t)$ at current time $t$ to $x_T^{*}(\delta T, x(t))$ at the next sample time $t + \Delta T$, assuming no model uncertainty.

In reality, the system will not follow the predicted path exactly, so that the red (computed) and blue (actual) trajectories will diverge. We thus recompute the optimal path from the new state at time $t + \Delta T$, extending our horizon by an additional $\Delta T$ units of time. This approach can be shown to generate stabilizing control laws under suitable conditions (see, for example, the FBS2e supplement on Optimization-Based Control.

### 9.2 Module usage

The optimal control module provides a means of computing optimal trajectories for nonlinear systems and implementing optimization-based controllers, including model predictive control. It follows the basic problem setup described above, but carries out all computations in discrete time (so that integrals become sums) and over a finite horizon. To local the optimal control modules, import `control.optimal`:

```python
import control.optimal as obc
```

To describe an optimal control problem we need an input/output system, a time horizon, a cost function, and (optionally) a set of constraints on the state and/or input, either along the trajectory and at the terminal time. The optimal control module operates by converting the optimal control problem into a standard optimization problem that can be solved by `scipy.optimize.minimize()`. The optimal control problem can be solved by using the `solve_ocp()` function:

```python
res = obc.solve_ocp(sys, horizon, X0, cost, constraints)
```

The `sys` parameter should be an `InputOutputSystem` and the `horizon` parameter should represent a time vector that gives the list of times at which the cost and constraints should be evaluated.

The `cost` function has call signature `cost(t, x, u)` and should return the (incremental) cost at the given time, state, and input. It will be evaluated at each point in the `horizon` vector. The `terminal_cost` parameter can be used to specify a cost function for the final point in the trajectory.

The `constraints` parameter is a list of constraints similar to that used by the `scipy.optimize.minimize()` function. Each constraint is a tuple of one of the following forms:
For a linear constraint, the 2D array $A$ is multiplied by a vector consisting of the current state $x$ and current input $u$ stacked vertically, then compared with the upper and lower bound. This constrain is satisfied if

$$lb \leq A \otimes \text{np.hstack([}x, u\text{])} \leq ub$$

A nonlinear constraint is satisfied if

$$lb \leq f(x, u) \leq ub$$

By default, constraints are taken to be trajectory constraints holding at all points on the trajectory. The \texttt{terminal\_constraint} parameter can be used to specify a constraint that only holds at the final point of the trajectory.

The return value for \texttt{solve_ocp()} is a bundle object that has the following elements:

- \texttt{res.success: True} if the optimization was successfully solved
- \texttt{res.inputs: optimal input}
- \texttt{res.states: state trajectory (if return\_x was True)}
- \texttt{res.time: copy of the time horizon vector}

In addition, the results from \texttt{scipy.optimize.minimize()} are also available.

To simplify the specification of cost functions and constraints, the \texttt{ios} module defines a number of utility functions:

- \texttt{quadratic\_cost(sys, Q, R[, x0, u0])} Create quadratic cost function
- \texttt{input\_poly\_constraint(sys, A, b)} Create input constraint from polytope
- \texttt{input\_range\_constraint(sys, lb, ub)} Create input constraint from polytope
- \texttt{output\_poly\_constraint(sys, A, b)} Create output constraint from polytope
- \texttt{output\_range\_constraint(sys, lb, ub)} Create output constraint from range
- \texttt{state\_poly\_constraint(sys, A, b)} Create state constraint from polytope
- \texttt{state\_range\_constraint(sys, lb, ub)} Create state constraint from polytope

### 9.3 Example

Consider the vehicle steering example described in FBS2e. The dynamics of the system can be defined as a nonlinear input/output system using the following code:

```python
import numpy as np
import control as ct
import control.optimal as opt
import matplotlib.pyplot as plt

def vehicle_update(t, x, u, params):
    # 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)
```

(continues on next page)
# Return the derivative of the state

```python
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)
```

```python
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

```python
vehicle = ct.NonlinearIOSystem(
    vehicle_update, vehicle_output, states=3, name='vehicle',
    inputs=('v', 'phi'), outputs=('x', 'y', 'theta'))
```

We consider an optimal control problem that consists of “changing lanes” by moving from the point \( x = 0 \text{ m}, y = -2 \text{ m}, \theta = 0 \) to the point \( x = 100 \text{ m}, y = 2 \text{ m}, \theta = 0 \) over a period of 10 seconds and with a with a starting and ending velocity of 10 m/s:

- \( x_0 = [0., -2., 0.] \)
- \( u_0 = [10., 0.] \)
- \( x_f = [100., 2., 0.] \)
- \( u_f = [10., 0.] \)
- \( T_f = 10 \)

To set up the optimal control problem we design a cost function that penalizes the state and input using quadratic cost functions:

```python
Q = np.diag([0, 0, 0.1])  # don't turn too sharply
R = np.diag([1, 1])  # keep inputs small
P = np.diag([1000, 1000, 1000])  # get close to final point
traj_cost = opt.quadratic_cost(vehicle, Q, R, x0=x_f, u0=u_f)
term_cost = opt.quadratic_cost(vehicle, P, 0, x0=x_f)
```

We also constraint the maximum turning rate to 0.1 radians (about 6 degrees) and constrain the velocity to be in the range of 9 m/s to 11 m/s:

```python
constraints = [opt.input_range_constraint(vehicle, [8, -0.1], [12, 0.1])]
```

Finally, we solve for the optimal inputs:

```python
horizon = np.linspace(0, T_f, 3, endpoint=True)
result = opt.solve_ocp(
    vehicle, horizon, x0, traj_cost, constraints,
    terminal_cost=term_cost, initial_guess=u0)
```

Plotting the results:

```python
# Simulate the system dynamics (open loop)
resp = ct.input_output_response(
    vehicle, horizon, result.inputs, x0,
    t_eval=np.linspace(0, T_f, 100))
t, y, u = resp.time, resp.outputs, resp.inputs
```
plt.subplot(3, 1, 1)
plt.plot(y[0], y[1])
plt.plot(x0[0], x0[1], 'ro', xf[0], xf[1], 'ro')
plt.xlabel('x [m]')
plt.ylabel('y [m]')

plt.subplot(3, 1, 2)
plt.plot(t, u[0])
plt.axis([0, 10, 9.9, 10.1])
plt.xlabel('t [sec]')
plt.ylabel('u1 [m/s]')

plt.subplot(3, 1, 3)
plt.plot(t, u[1])
plt.axis([0, 10, -0.01, 0.01])
plt.xlabel('t [sec]')
plt.ylabel('u2 [rad/s]')

plt.suptitle('Lane change maneuver')
plt.tight_layout()
plt.show()

yields
The python-control optimization module makes use of the SciPy optimization toolbox and it can sometimes be tricky to get the optimization to converge. If you are getting errors when solving optimal control problems or your solutions do not seem close to optimal, here are a few things to try:

- **Less is more**: try using a smaller number of time points in your optimization. The default optimal control problem formulation uses the value of the inputs at each time point as a free variable and this can generate a large number of parameters quickly. Often you can find very good solutions with a small number of free variables (the example above uses 3 time points for 2 inputs, so a total of 6 optimization variables). Note that you can “resample” the optimal trajectory by running a simulation of the system and using the `t_eval` keyword in `input_output_response` (as done above).

- **Use a smooth basis**: as an alternative to parameterizing the optimal control inputs using the value of the control at the listed time points, you can specify a set of basis functions using the `basis` keyword in `solve_ocp()` and then parameterize the controller by linear combination of the basis functions. The `control.flatsys` module defines several sets of basis functions that can be used.

- **Tweak the optimizer**: by using the `minimize_method`, `minimize_options`, and `minimize_kwargs` keywords in `solve_ocp()`, you can choose the SciPy optimization function that you use and set many parameters. See `scipy.optimize.minimize()` for more information on the optimizers that are available and the options and keywords that they accept.

- **Walk before you run**: try setting up a simpler version of the optimization, remove constraints or simplifying the
cost to get a simple version of the problem working and then add complexity. Sometimes this can help you find the right set of options or identify situations in which you are being too aggressive in what your are trying to get the system to do.

See *Optimal control for vehicle steering (lane change)* for some examples of different problem formulations.

## 9.5 Module classes and functions

### 9.5.1 control.optimal.OptimalControlProblem

```python
class control.optimal.OptimalControlProblem(sys, timepts, integral_cost, trajectory_constraints=[], terminal_cost=None, terminal_constraints=[], initial_guess=None, basis=None, log=False, **kwargs)
```

**Bases:** object

Description of a finite horizon, optimal control problem.

The `OptimalControlProblem` class holds all of the information required to specify an optimal control problem: the system dynamics, cost function, and constraints. As much as possible, the information used to specify an optimal control problem matches the notation and terminology of the SciPy `optimize.minimize` module, with the hope that this makes it easier to remember how to describe a problem.

**Parameters**

- `sys` (*InputOutputSystem*) – I/O system for which the optimal input will be computed.
- `timepts` (*1D array_like*) – List of times at which the optimal input should be computed.
- `integral_cost` (*callable*) – Function that returns the integral cost given the current state and input. Called as `integral_cost(x, u)`.
- `trajectory_constraints` (*list of tuples, optional*) – List of constraints that should hold at each point in the time vector. Each element of the list should consist of a tuple with first element given by `LinearConstraint()` or `NonlinearConstraint()` and the remaining elements of the tuple are the arguments that would be passed to those functions. The constraints will be applied at each time point along the trajectory.
- `terminal_cost` (*callable, optional*) – Function that returns the terminal cost given the current state and input. Called as `terminal_cost(x, u)`.
- `initial_guess` (*1D or 2D array_like*) – Initial inputs to use as a guess for the optimal input. The inputs should either be a 2D vector of shape (ninputs, horizon) or a 1D input of shape (ninputs,) that will be broadcast by extension of the time axis.
- `log` (*bool, optional*) – If `True`, turn on logging messages (using Python logging module). Use `logging.basicConfig` to enable logging output (e.g., to a file).
- `**kwargs` (*dict, optional*) – Additional parameters (passed to `scipy.optimize.minimize()`).

**Returns**

- `ocp` (*OptimalControlProblem*) – Optimal control problem object, to be used in computing optimal controllers.
• Additional parameters
  
  • `solve_ivp_method` *(str, optional)*  – Set the method used by `scipy.integrate.solve_ivp()`.
  
  • `solve_ivp_kwargs` *(str, optional)*  – Pass additional keywords to `scipy.integrate.solve_ivp()`.
  
  • `minimize_method` *(str, optional)*  – Set the method used by `scipy.optimize.minimize()`.
  
  • `minimize_options` *(str, optional)*  – Set the options keyword used by `scipy.optimize.minimize()`.
  
  • `minimize_kwargs` *(str, optional)*  – Pass additional keywords to `scipy.optimize.minimize()`.

**Notes**

To describe an optimal control problem we need an input/output system, a time horizon, a cost function, and (optionally) a set of constraints on the state and/or input, either along the trajectory and at the terminal time. This class sets up an optimization over the inputs at each point in time, using the integral and terminal costs as well as the trajectory and terminal constraints. The `compute_trajectory` method sets up an optimization problem that can be solved using `scipy.optimize.minimize()`.

The `_cost_function` method takes the information computes the cost of the trajectory generated by the proposed input. It does this by calling a user-defined function for the integral_cost given the current states and inputs at each point along the trajectory and then adding the value of a user-defined terminal cost at the final point in the trajectory.

The `_constraint_function` method evaluates the constraint functions along the trajectory generated by the proposed input. As in the case of the cost function, the constraints are evaluated at the state and input along each point on the trajectory. This information is compared against the constraint upper and lower bounds. The constraint function is processed in the class initializer, so that it only needs to be computed once.

If `basis` is specified, then the optimization is done over coefficients of the basis elements. Otherwise, the optimization is performed over the values of the input at the specified times (using linear interpolation for continuous systems).

The default values for `minimize_method`, `minimize_options`, `minimize_kwargs`, `solve_ivp_method`, and `solve_ivp_options` can be set using `config.defaults['optimal.<keyword>']`.

**Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>compute_mpc</code></td>
<td>Compute the optimal input at state x</td>
</tr>
<tr>
<td><code>compute_trajectory</code></td>
<td>Compute the optimal input at state x</td>
</tr>
<tr>
<td><code>create_mpc_iosystem</code></td>
<td>Create an I/O system implementing an MPC controller</td>
</tr>
</tbody>
</table>

**compute_mpc(x, squeeze=None)**

Compute the optimal input at state x

This function calls the `compute_trajectory()` method and returns the input at the first time point.

**Parameters**
• **x** (*array-like or number, optional*) – Initial state for the system.

• **squeeze** (*bool, optional*) – If True and if the system has a single output, return the system output as a 1D array rather than a 2D array. If False, return the system output as a 2D array even if the system is SISO. Default value set by config.defaults[‘control.squeeze_time_response’].

**Returns**

  • **input** – Optimal input for the system at the current time. If the system is SISO and squeeze is not True, the array is 1D (indexed by time). If the system is not SISO or squeeze is False, the array is 2D (indexed by the output number and time). Set to **None** if the optimization failed.

**Return type**

array

**compute_trajectory**(*x*, *squeeze=None*, *transpose=None*, *return_states=None*, *initial_guess=None*, *print_summary=True*, **kwargs)

Compute the optimal input at state **x**

**Parameters**

• **x** (*array-like or number, optional*) – Initial state for the system.

• **return_states** (*bool, optional*) – If True, return the values of the state at each time (default = False).

• **squeeze** (*bool, optional*) – If True and if the system has a single output, return the system output as a 1D array rather than a 2D array. If False, return the system output as a 2D array even if the system is SISO. Default value set by config.defaults[‘control.squeeze_time_response’].

• **transpose** (*bool, optional*) – If True, assume that 2D input arrays are transposed from the standard format. Used to convert MATLAB-style inputs to our format.

**Returns**

• **res** (*OptimalControlResult*) – Bundle object with the results of the optimal control problem.

• **res.success** (*bool*) – Boolean flag indicating whether the optimization was successful.

• **res.time** (*array*) – Time values of the input.

• **res.inputs** (*array*) – Optimal inputs for the system. If the system is SISO and squeeze is not True, the array is 1D (indexed by time). If the system is not SISO or squeeze is False, the array is 2D (indexed by the output number and time).

• **res.states** (*array*) – Time evolution of the state vector (if return_states=True).

**create_mpc_iosystem**()  
Create an I/O system implementing an MPC controller
9.5.2 control.optimal.OptimalControlResult

class control.optimal.OptimalControlResult(ocp, res, return_states=False, print_summary=False, transpose=None, squeeze=None)

Bases: OptimizeResult

Result from solving an optimal control problem.

This class is a subclass of scipy.optimize.OptimizeResult with additional attributes associated with solving optimal control problems.

inputs

The optimal inputs associated with the optimal control problem.

Type
ndarray

states

If return_states was set to true, stores the state trajectory associated with the optimal input.

Type
ndarray

success

Whether or not the optimizer exited successful.

Type
bool

problem

Optimal control problem that generated this solution.

Type
OptimalControlProblem

cost

Final cost of the return solution.

Type
float

system_simulations, {cost, constraint, eqconst}_evaluations

Number of system simulations and evaluations of the cost function, (inequality) constraint function, and equality constraint function performed during the optimization.

Type
int

{cost, constraint, eqconst}_process_time

If logging was enabled, the amount of time spent evaluating the cost and constraint functions.

Type
float
**Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>clear</code></td>
<td>-</td>
</tr>
<tr>
<td><code>copy</code></td>
<td>-</td>
</tr>
<tr>
<td><code>fromkeys</code></td>
<td>Create a new dictionary with keys from iterable and values set to value.</td>
</tr>
<tr>
<td><code>get</code></td>
<td>Return the value for key if key is in the dictionary, else default.</td>
</tr>
<tr>
<td><code>items</code></td>
<td>-</td>
</tr>
<tr>
<td><code>keys</code></td>
<td>-</td>
</tr>
<tr>
<td><code>pop</code></td>
<td>If key is not found, d is returned if given, otherwise <code>KeyError</code> is raised</td>
</tr>
<tr>
<td><code>popitem</code></td>
<td>2-tuple; but raise <code>KeyError</code> if D is empty.</td>
</tr>
<tr>
<td><code>setdefault</code></td>
<td>Insert key with a value of default if key is not in the dictionary.</td>
</tr>
<tr>
<td><code>update</code></td>
<td>If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]</td>
</tr>
<tr>
<td><code>values</code></td>
<td>-</td>
</tr>
</tbody>
</table>

**Special Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>__contains__</code>(key, /)</td>
<td>True if the dictionary has the specified key, else False.</td>
</tr>
<tr>
<td><code>__delattr__</code>(key, /)</td>
<td>Delete self[key].</td>
</tr>
<tr>
<td><code>__delitem__</code>(key, /)</td>
<td>Delete self[key].</td>
</tr>
<tr>
<td><code>__dir__</code>()</td>
<td>Default dir() implementation.</td>
</tr>
<tr>
<td><code>__eq__</code>(value, /)</td>
<td>Return self==value.</td>
</tr>
<tr>
<td><code>__ge__</code>(value, /)</td>
<td>Return self&gt;=value.</td>
</tr>
<tr>
<td><code>__getattribute__</code>(name, /)</td>
<td>Return getattr(self, name).</td>
</tr>
<tr>
<td><code>__getitem__</code>()</td>
<td>x.<strong>getitem</strong>(y) &lt;==&gt; x[y]</td>
</tr>
<tr>
<td><code>__gt__</code>(value, /)</td>
<td>Return self&gt;value.</td>
</tr>
<tr>
<td><code>__hash__</code> = None</td>
<td></td>
</tr>
</tbody>
</table>

9.5. Module classes and functions
__iter__()  
Implement iter(self).

__le__(value,/)  
Return self<=value.

__len__()  
Return len(self).

__lt__(value,/)  
Return self<value.

__ne__(value,/)  
Return self!=value.

__new__(**kwargs)  

__setattr__(key, value,/)  
Set self[key] to value.

__setitem__(key, value,/)  
Set self[key] to value.

__sizeof__() → size of D in memory, in bytes

clear() → None. Remove all items from D.

copy() → a shallow copy of D

fromkeys(value=None,/)  
Create a new dictionary with keys from iterable and values set to value.

get(key, default= None,/)  
Return the value for key if key is in the dictionary, else default.

items() → a set-like object providing a view on D's items

keys() → a set-like object providing a view on D's keys

pop(k[, d]) → v, remove specified key and return the corresponding value.  
If key is not found, d is returned if given, otherwise KeyError is raised

popitem() → (k, v), remove and return some (key, value) pair as a 2-tuple; but raise KeyError if D is empty.

setdefault(key, default= None,/)  
Insert key with a value of default if key is not in the dictionary.  
Return the value for key if key is in the dictionary, else default.

update([E], **F) → None. Update D from dict/iterable E and F.  
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values
9.5.3 control.optimal.solve_ocp

control.optimal.solve_ocp(sys, horizon, X0, cost, trajectory_constraints=None, terminal_cost=None, terminal_constraints=[], initial_guess=None, basis=None, squeeze=None, transpose=None, return_states=False, log=False, **kwargs)

Compute the solution to an optimal control problem

Parameters

- **sys** (*InputOutputSystem*) – I/O system for which the optimal input will be computed.
- **horizon** (*1D array_like*) – List of times at which the optimal input should be computed.
- **X0** (*array-like or number, optional*) – Initial condition (default = 0).
- **cost** (*callable*) – Function that returns the integral cost given the current state and input. Called as `cost(x, u)`.
- **trajectory_constraints** (*list of tuples, optional*) – List of constraints that should hold at each point in the time vector. Each element of the list should consist of a tuple with first element given by `scipy.optimize.LinearConstraint()` or `scipy.optimize.NonlinearConstraint()` and the remaining elements of the tuple are the arguments that would be passed to those functions. The following tuples are supported:
  - (LinearConstraint, A, lb, ub): The matrix A is multiplied by stacked vector of the state and input at each point on the trajectory for comparison against the upper and lower bounds.
  - (NonlinearConstraint, fun, lb, ub): A user-specific constraint function `fun(x, u)` is called at each point along the trajectory and compared against the upper and lower bounds.

The constraints are applied at each time point along the trajectory.
- **terminal_cost** (*callable, optional*) – Function that returns the terminal cost given the current state and input. Called as `terminal_cost(x, u)`.
- **terminal_constraints** (*list of tuples, optional*) – List of constraints that should hold at the end of the trajectory. Same format as `constraints`.
- **initial_guess** (*1D or 2D array_like*) – Initial inputs to use as a guess for the optimal input. The inputs should either be a 2D vector of shape (ninputs, horizon) or a 1D input of shape (ninputs,) that will be broadcast by extension of the time axis.
- **log** (*bool, optional*) – If `True`, turn on logging messages (using Python logging module).
- **return_states** (*bool, optional*) – If `True`, return the values of the state at each time (default = `False`).
- **squeeze** (*bool, optional*) – If `True` and if the system has a single output, return the system output as a 1D array rather than a 2D array. If `False`, return the
system output as a 2D array even if the system is SISO. Default value set by con-
fig.defaults['control.squeeze_time_response'].

- **transpose** *(bool, optional)* – If True, assume that 2D input arrays are transposed from
  the standard format. Used to convert MATLAB-style inputs to our format.

- **kwargs** *(dict, optional)* – Additional parameters (passed to scipy.optimal.
  minimize()).

Returns

- **res** *(OptimalControlResult)* – Bundle object with the results of the optimal control problem.
- **res.success** *(bool)* – Boolean flag indicating whether the optimization was successful.
- **res.time** *(array)* – Time values of the input.
- **res.inputs** *(array)* – Optimal inputs for the system. If the system is SISO and squeeze is not
  True, the array is 1D (indexed by time). If the system is not SISO or squeeze is False, the
  array is 2D (indexed by the output number and time).
- **res.states** *(array)* – Time evolution of the state vector (if return_states=True).

Notes

Additional keyword parameters can be used to fine tune the behavior of the underlying optimization and integra-
tions functions. See `OptimalControlProblem()` for more information.

### 9.5.4 control.optimal.create_mpc_iosystem

control.optimal.create_mpc_iosystem*(sys, horizon, cost, constraints=[], terminal_cost=None,
  terminal_constraints=[], log=False, **kwargs)*

Create a model predictive I/O control system

This function creates an input/output system that implements a model predictive control for a system given the
time horizon, cost function and constraints that define the finite-horizon optimization that should be carried out
at each state.

Parameters

- **sys** *(InputOutputSystem)* – I/O system for which the optimal input will be computed.
- **horizon** *(1D array_like)* – List of times at which the optimal input should be computed.
- **cost** *(callable)* – Function that returns the integral cost given the current state and input.
  Called as cost(x, u).
- **constraints** *(list of tuples, optional)* – List of constraints that should hold at
  each point in the time vector. See `solve_ocp()` for more details.
- **terminal_cost** *(callable, optional)* – Function that returns the terminal cost given
  the current state and input. Called as terminal_cost(x, u).
- **terminal_constraints** *(list of tuples, optional)* – List of constraints that
  should hold at the end of the trajectory. Same format as `constraints`.
- **kwargs** *(dict, optional)* – Additional parameters (passed to scipy.optimal.
  minimize()).
Returns
ctrl – An I/O system taking the current state of the model system and returning the current input to be applied that minimizes the cost function while satisfying the constraints.

Return type
InputOutputSystem

Notes
Additional keyword parameters can be used to fine tune the behavior of the underlying optimization and integrations functions. See OptimalControlProblem() for more information.

9.5.5 control.optimal.input_poly_constraint

control.optimal.input_poly_constraint(sys, A, b)
Create input constraint from polytope

Creates a linear constraint on the system input of the form $A u \leq b$ that can be used as an optimal control constraint (trajectory or terminal).

Parameters
- sys (InputOutputSystem) – I/O system for which the constraint is being defined.
- A (2D array) – Constraint matrix
- b (1D array) – Upper bound for the constraint

Returns
constraint – A tuple consisting of the constraint type and parameter values.

Return type
tuple

9.5.6 control.optimal.input_range_constraint

control.optimal.input_range_constraint(sys, lb, ub)
Create input constraint from polytope

Creates a linear constraint on the system input that bounds the range of the individual states to be between $lb$ and $ub$. The upper and lower bounds can be set of inf and -inf to indicate there is no constraint or to the same value to describe an equality constraint.

Parameters
- sys (InputOutputSystem) – I/O system for which the constraint is being defined.
- lb (1D array) – Lower bound for each of the inputs.
- ub (1D array) – Upper bound for each of the inputs.

Returns
constraint – A tuple consisting of the constraint type and parameter values.

Return type
tuple
9.5.7 control.optimal.output_poly_constraint

control.optimal.output_poly_constraint(sys, A, b)

Create output constraint from polytope

Creates a linear constraint on the system output of the form \( A \ y \leq b \) that can be used as an optimal control constraint (trajectory or terminal).

Parameters

- **sys** (InputOutputSystem) – I/O system for which the constraint is being defined.
- **A** (2D array) – Constraint matrix
- **b** (1D array) – Upper bound for the constraint

Returns

- **constraint** – A tuple consisting of the constraint type and parameter values.

Return type
tuple

9.5.8 control.optimal.output_range_constraint

control.optimal.output_range_constraint(sys, lb, ub)

Create output constraint from range

Creates a linear constraint on the system output that bounds the range of the individual states to be between \( lb \) and \( ub \). The upper and lower bounds can be set of \( \text{inf} \) and \( -\text{inf} \) to indicate there is no constraint or to the same value to describe an equality constraint.

Parameters

- **sys** (InputOutputSystem) – I/O system for which the constraint is being defined.
- **lb** (1D array) – Lower bound for each of the outputs.
- **ub** (1D array) – Upper bound for each of the outputs.

Returns

- **constraint** – A tuple consisting of the constraint type and parameter values.

Return type
tuple

9.5.9 control.optimal.state_poly_constraint

control.optimal.state_poly_constraint(sys, A, b)

Create state constraint from polytope

Creates a linear constraint on the system state of the form \( A \ x \leq b \) that can be used as an optimal control constraint (trajectory or terminal).

Parameters

- **sys** (InputOutputSystem) – I/O system for which the constraint is being defined.
- **A** (2D array) – Constraint matrix
- **b** (1D array) – Upper bound for the constraint
Returns
constraint – A tuple consisting of the constraint type and parameter values.

Return type
tuple

9.5.10 control.optimal.state_range_constraint

control.optimal.state_range_constraint(sys, lb, ub)

Create state constraint from polytope

Creates a linear constraint on the system state that bounds the range of the individual states to be between lb and ub. The upper and lower bounds can be set of inf and -inf to indicate there is no constraint or to the same value to describe an equality constraint.

Parameters

- sys (InputOutputSystem) – I/O system for which the constraint is being defined.
- lb (1D array) – Lower bound for each of the states.
- ub (1D array) – Upper bound for each of the states.

Returns
constraint – A tuple consisting of the constraint type and parameter values.

Return type
tuple
The source code for the examples below are available in the `examples/` subdirectory 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).

### 10.1 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.

#### 10.1.1 Second order system (MATLAB module example)

This example computes time and frequency responses for a second-order system using the MATLAB compatibility module.

**Code**

```python
# 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)
```

(continues on next page)
```python
plt.plot(T.T, yout.T)
plt.show(block=False)

# 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)
plt.show(block=False)

# Root locus plot for the system
rlocus(sys)
```

Notes

1. The environment variable `PYCONTROL_TEST_EXAMPLES` is used for testing to turn off plotting of the outputs.

10.1.2 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

```python
# 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.
#
import os
import matplotlib.pyplot as plt  # MATLAB-like plotting functions
import control as ct
import numpy as np

# System parameters
m = 4                      # mass of aircraft
J = 0.0475                 # inertia around pitch axis
```
```python
r = 0.25  # distance to center of force
g = 9.8  # gravitational constant
c = 0.05  # damping factor (estimated)

# Transfer functions for dynamics
Pi = ct.tf([r], [1, 0, 0])  # inner loop (roll)
Po = ct.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 * ct.tf([1, a], [1, b])  # lead compensator
Li = Pi * Ci

# Bode plot for the open loop process
plt.figure(1)
ct.bode_plot(Pi)

# Bode plot for the loop transfer function, with margins
plt.figure(2)
ct.bode_plot(Li)

# Compute out the gain and phase margins
gm, pm, wcg, wcp = ct.margin(Li)

# Compute the sensitivity and complementary sensitivity functions
Si = ct.feedback(1, Li)
Ti = Li * Si

# Check to make sure that the specification is met
plt.figure(3)
ct.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 = ct.parallel(ct.feedback(Ci, Pi), -m * g *ct.feedback(Ci * Pi, 1))

plt.figure(4)
plt.clf()
plt.subplot(221)
ct.bode_plot(Hi)

# Now design the lateral control system
a, b, K = 0.02, 5, 2
Co = -K * ct.tf([1, 0.3], [1, 10])  # another lead compensator
```

(continues on next page)
Lo = -m*g*Po*Co

plt.figure(5)
ct.bode_plot(Lo)  # margin(Lo)

# Finally compute the real outer-loop loop gain + responses
L = Co * Hi * Po
S = ct.feedback(1, L)
T = ct.feedback(L, 1)

# Compute stability margins
gm, pm, wgc, wpc = ct.margin(L)
print("Gain margin: %g at %g" % (gm, wgc))
print("Phase margin: %g at %g" % (pm, wpc))

plt.figure(6)
plt.clf()
ct.bode_plot(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], [-180, -180], 'k-')
ax.semilogx(w, np.squeeze(phase), 'b-')
ax.axis([1e-4, 1e3, -180, 0])
plt.xlabel('Frequency [deg]')

mag, phase, w = ct.freqresp(L, np.logspace(-4, 3))
phase = phase - 360

# Recreate the frequency response and shift the phase
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]')
```python
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()
ct.nyquist_plot(L, (0.0001, 1000))

# Add a box in the region we are going to expand
plt.plot([-2, -2, 1, 1, -2], [-4, 4, 4, -4, -4], 'r-')

# Expanded region
plt.figure(8)
plt.clf()
ct.nyquist_plot(L)
plt.axis([-2, 1, -4, 4])

# set up the color
color = 'b'

# Add arrows to the plot
# H1 = L.evalfr(0.4); H2 = L.evalfr(0.41);
# arrow([real(H1), imag(H1)], [real(H2), imag(H2)], AM_normal_arrowsize,
# 'EdgeColor', color, 'FaceColor', color);

# H1 = freqresp(L, 0.35); H2 = freqresp(L, 0.36);
# arrow([real(H2), -imag(H2)], [real(H1), -imag(H1)], AM_normal_arrowsize,
# 'EdgeColor', color, 'FaceColor', color);

plt.figure(9)
Tvec, Yvec = ct.step_response(T, np.linspace(0, 20))
plt.plot(Tvec.T, Yvec.T)
Tvec, Yvec = ct.step_response(Co*S, np.linspace(0, 20))
plt.plot(Tvec.T, Yvec.T)

plt.figure(10)
plt.clf()
P, Z = ct.pzmap(T, plot=True, grid=True)
print("Closed loop poles and zeros: ", P, Z)

# Gang of Four
plt.figure(11)
plt.clf()
ct.gangof4_plot(Hi * Po, Co)

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.

### 10.1.3 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-like plotting functions
import control as ct

# 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.
A = np.array([
    [0, 0, 0, 1, 0],
    [0, 0, 0, 0, 1],
    [0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0],
])
```

(continues on next page)
[[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.array(
    [[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.array([[1, 0, 0, 0, 0, 0], [0, 1, 0, 0, 0, 0]]
D = np.array([[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.array([[1], [0], [0], [0], [0], [0]])
yd = np.array([[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

\[
\begin{align*}
A_x &= A[np.ix_(\text{lat}, \text{lat})] \\
B_x &= B[np.ix_(\text{lat}, [0])] \\
C_x &= C[np.ix_([0], \text{lat})] \\
D_x &= D[np.ix_([0], [0])]
\end{align*}
\]

\[
\begin{align*}
A_y &= A[np.ix_(\text{alt}, \text{alt})] \\
B_y &= B[np.ix_(\text{alt}, [1])] \\
C_y &= C[np.ix_([1], \text{alt})] \\
D_y &= D[np.ix_([1], [1])]
\end{align*}
\]

# Label the plot

```python
plt.clf()
plt.suptitle("LQR controllers for vectored thrust aircraft (pvtol-lqr)")
```

# LQR design

# Start with a diagonal weighting

```python
Qx1 = np.diag([1, 1, 1, 1, 1, 1])
Qu1a = np.diag([1, 1])
K1a, X, E = ct.lqr(A, B, Qx1, Qu1a)
```

# Close the loop: \( \dot{x} = Ax - B K (x-xd) \)

```python
# Note: python-control requires we do this 1 input at a time
# H1a = ss(A-B*Kla, B*Kla*concatenate((xd, yd), axis=1), C, D);
# (T, Y) = step_response(H1a, T=np.linspace(0,10,100));
```

# Step response for the first input

```python
H1ax = ct.ss(Ax - Bx @ K1a[np.ix_(0], lat)],
          Bx @ K1a[np.ix_(0], lat]) @ xd[lat, :], Cx, Dx)
Tx, Yx = ct.step_response(H1ax, T=np.linspace(0, 10, 100))
```

# Step response for the second input

```python
H1ay = ct.ss(Ay - By @ K1a[np.ix_(1], alt)],
          By @ K1a[np.ix_(1], alt]) @ yd[alt, :], Cy, Dy)
Ty, Yy = ct.step_response(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-')
plt.axis([0, 10, -0.1, 1.4])
plt.ylabel('position')
plt.legend(('x', 'y'), loc='lower right')
```

(continues on next page)
# Look at different input weightings

```python
Qua = np.diag([1, 1])
K1a, X, E = ct.lqr(A, B, Qx1, Qua)
Hlax = ct.ss(Ax - Bx @ K1a[np.ix_([0], lat)],
            Bx @ K1a[np.ix_([0], lat)] @ xd[lat, :], Cx, Dx)
```

```python
Quib = (40 ** 2)*np.diag([1, 1])
Kib, X, E = ct.lqr(A, B, Qx1, Quib)
Hlbx = ct.ss(Ax - Bx @ Kib[np.ix_([0], lat)],
            Bx @ Kib[np.ix_([0], lat)] @ xd[lat, :], Cx, Dx)
```

```python
Quic = (200 ** 2)*np.diag([1, 1])
Kic, X, E = ct.lqr(A, B, Qx1, Quic)
Hlcx = ct.ss(Ax - Bx @ Kic[np.ix_([0], lat)],
            Bx @ Kic[np.ix_([0], lat)] @ xd[lat, :], Cx, Dx)
```

```python
T1, Y1 = ct.step_response(Hlax, T=np.linspace(0, 10, 100))
T2, Y2 = ct.step_response(Hlbx, T=np.linspace(0, 10, 100))
T3, Y3 = ct.step_response(Hlcx, T=np.linspace(0, 10, 100))
```

```python
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])
plt.text(5.3, 0.4, r'$\rho$')
```

```python
# Output weighting - change Qx to use outputs
Qx2 = C.T @ C
Qu2 = 0.1 * np.diag([1, 1])
K2, X, E = ct.lqr(A, B, Qx2, Qu2)
```

```python
H2x = ct.ss(Ax - Bx @ K2[np.ix_([0], lat)],
            Bx @ K2[np.ix_([0], lat)] @ xd[lat, :], Cx, Dx)
H2y = ct.ss(Ay - By @ K2[np.ix_([1], alt)],
            By @ K2[np.ix_([1], alt)] @ yd[alt, :], Cy, Dy)
```

```python
plt.subplot(223)
plt.title("Output weighting")
plt.plot(T2x.T, Y2x.T, T2y.T, Y2y.T)
plt.ylabel('position')
plt.xlabel('time')
plt.legend(('x', 'y'), loc='lower right')
```

(continues on next page)
# 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
# due to loss in efficiency.

Qx3 = np.diag([100, 10, 2*np.pi/5, 0, 0, 0])
Qu3 = 0.1*np.diag([1, 10])
K3, X, E = ct.lqr(A, B, Qx3, Qu3)

H3x = ct.ss(Ax - Bx @ K3[np.ix_([0], lat)],
            Bx @ K3[np.ix_([0], lat)] @ xd[lat, :], Cx, Dx)
H3y = ct.ss(Ay - By @ K3[np.ix_([1], alt)],
            By @ K3[np.ix_([1], alt)] @ yd[alt, :], Cy, Dy)
plt.subplot(224)
# step_response(H3x, H3y, 10)
T3x, Y3x = ct.step_response(H3x, T=np.linspace(0, 10, 100))
T3y, Y3y = ct.step_response(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')
plt.tight_layout()

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.

10.1.4 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.],
               [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.

10.1.5 Phase plot examples

Code

```python
# 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

# 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)
```

(continues on next page)
# Systems of ODEs: damped oscillator example (simulation + phase portrait)

```python
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
```python
plt.figure()
plt.clf()
phase_plot(oscillator_ode, [-1, 1, 10], [-1, 1, 10], 0.15)
# plt.plot([0], [0], 'k.') # MarkerSize, AM_data_markersize*3)
plt.xlabel('$x_1$')
plt.ylabel('$x_2$')
plt.title('Damped oscillator, vector field')
```

# Generate a phase plot for the damped oscillator
```python
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], [0.75, 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.

```python
def saddle_ode(x, t):
    return x[0] - 3*x[1], -3*x[0] + x[1]
```

# Asy stable
m = 1
\[ b = 1 \quad \# \text{default values} \]

```
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],
    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)
# plt.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()
```
149 plt.clf()
150 plt.axis([-1, 1, -1, 1])  # set(gca, 'DataAspectRatio', [1 1 1]);
151 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)  
)
159 plt.plot([0], [0], 'k.')  # 'MarkerSize', AM_data_markersize*3)
161 plt.xlabel('$x_1$')
162 plt.ylabel('$x_2$')
163 plt.title('Undamped system

Lyapunov stable, not asympt. stable')
165 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.

10.1.6 SISO robust control example (SP96, Example 2.1)

Code

```python
"""robust_siso.py

Demonstrate mixed-sensitivity H-infinity design for a SISO plant.

Based on Example 2.11 from Multivariable Feedback Control, Skogestad
"""

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

\[ M = 1.5 \]
\[ wb = 10 \]
\[ A = 1 \times 10^{-4} \]
\[
ws1 = \frac{s/M + wb}{s + wb^2 A}
\]
# KS weighting

\[ wu = tf(1, 1) \]

\[
k1, cl1, info1 = mixsyn(g, ws1, wu)
\]

# sensitivity (S) and complementary sensitivity (T) functions for
# design 1

\[
s1 = feedback(1, g \cdot k1)
\]
\[
t1 = feedback(g \cdot k1, 1)
\]

# 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.

\[
ws2 = \frac{(s/M)^{0.5} + wb}{(s + wb^2 A)^2}
\]
# the KS weighting is the same as for the first design

\[
k2, cl2, info2 = mixsyn(g, ws2, wu)
\]

# S and T for design 2

\[
s2 = feedback(1, g \cdot k2)
\]
\[
t2 = feedback(g \cdot k2, 1)
\]

# frequency response

\[
\text{omega = np.logspace(-2, 2, 101)}
\]
\[
ws1mag, _, _ = ws1.frequency_response(omega)
\]
\[
slmag, _, _ = sl1.frequency_response(omega)
\]
\[
ws2mag, _, _ = ws2.frequency_response(omega)
\]
\[
s2mag, _, _ = s2.frequency_response(omega)
\]

plt.figure(1)
# text uses log-scaled absolute, but dB are probably more familiar to most control...
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

(continues on next page)
```python
import numpy as np
import matplotlib.pyplot as plt

# Define time vector
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')
```

Notes

1. The environment variable `PYCONTROL_TEST_EXAMPLES` is used for testing to turn off plotting of the outputs.

10.1.7 MIMO robust control example (SP96, Example 3.8)

Code

```python
"""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])
    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.frequency_response(w)
    sjw = (m*np.exp(1j*p)).transpose(2, 0, 1)
    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)
    _, yul = step_response(g, t, input=0, squeeze=True)
.. code-block:: python

    # linear system, so scale and sum previous results to get the
    # [1,-1] response
    yuz = yu1 - yu2

    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()
        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):
    """response to step of [-1,1]""
    _, yu1 = step_response(g, t, input=0, squeeze=True)
    _, yu2 = step_response(g, t, input=1, squeeze=True)
    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(f'design 1 gamma {gam1:.3g} (Skogestad: 2.80)')
    print(f'design 2 gamma {gam2:.3g} (Skogestad: 2.92)')
    print(f'design 3 gamma {gam3:.3g} (Skogestad: 6.73)')

    # 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

    # 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.

10.1.8 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 explores the dynamics and control of the cruise control system,
# following the material presented 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 shown, all using the InputOutputSystem class (and subclasses).
#
import numpy as np
import matplotlib.pyplot as plt

(continues on next page)
from math import pi
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 \( F_d \) 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

    # Add force balance equations for the car body
    F = m * v * sin(theta)  # Force due to contact with road
    F_d = m * g * (cos(theta) + Cd * v * rho / (2 * A) * sin(theta))
    F厍 = m * v * cos(theta)  # Disturbance force due to gravity, friction, and drag

    # Calculate acceleration
    acceleration = (F - F_d) / m

    return acceleration
```python
# 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
#
# 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 \text{sgn}(v), where Cr is
# the coefficient of rolling friction and \text{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(
```

(continues on next page)
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
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'),
   outlist=('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(1)
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)
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.

# Figure 4.2
fig, axes = plt.subplots(1, 2, figsize=(7, 3))

(a) - single torque curve as function of omega
ax = axes[0]
omega = np.linspace(0, 700, 701)
ax.plot(omega, motor_torque(omega))
ax.set_xlabel(r'$\text{Angular velocity } \omega \text{ [rad/s]}$')
ax.set_ylabel(r'$\text{Torque } T \text{ [Nm]}$')
ax.grid(True, linestyle='dotted')

(b) - torque curves in different gears, as function of velocity
ax = axes[1]
v = np.linspace(0, 70, 71)
alpha = [40, 25, 16, 12, 10]
for gear in range(5):
    omega = alpha[gear] * v
    T = motor_torque(omega)
    plt.plot(v, T, color='#1f77b4', linestyle='solid')

# Set up the axes and style
ax.axis([0, 70, 100, 200])
ax.grid(True, linestyle='dotted')

# Add labels
plt.text(11.5, 120, '$n=1$')
a.text(24, 120, '$n=2$')
a.text(42.5, 120, '$n=3$')

(continues on next page)
ax.text(58.5, 120, '$n$=4')
ax.text(58.5, 185, '$n$=5')
ax.set_xlabel('Velocity $v$ [m/s]
ax.set_ylabel('Torque $T$ [Nm]

plt.suptitle('Torque curves for typical car engine')
plt.tight_layout()
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)
    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

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, label=None, t_hill=None, vref=20, antiwindup=False,
                 linetype='b-', subplots=None, legend=None):

  if subplots is None:
    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
  subplot_axes = list(subplots)

  # Velocity profile
  if subplot_axes[0] is None:
    subplot_axes[0] = plt.subplot(2, 1, 1)
  else:
    plt.sca(subplot_axes[0])
  plt.plot(t, y[v_ind], linetype)
  plt.plot(t, vref*np.ones(t.shape), 'k--', label='vref')
  if t_hill:
    plt.axvline(t_hill, color='k', linestyle='--', label='t hill')
  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(subplot_axes[1])
  plt.plot(t, y[u_ind], 'r--' if antiwindup else linetype, label=label)
  # Applied input profile

(continues on next page)
```python
if antiwindup:
    # TODO: plot the actual signal from the process?
    plt.plot(t, np.clip(y[u_ind], 0, 1), linetype, label='Applied')
if t_hill:
    plt.axvline(t_hill, color='k', linestyle='--')
if legend:
    plt.legend(frameon=False)
plt.axis([0, t[-1], u_min, u_max])
plt.xlabel('Time $t$ $[s]$')
plt.ylabel('Throttle $u$')

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, t_hill=5)

# Example 7.8: State space feedback with integral action
# 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
```
def sf_update(t, z, u, params={}):
    y, r = u[1], u[2]
    return y - r

def sf_output(t, z, u, params={}):
    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)

control_sf = ct.NonlinearIOSystem(
    sf_update, sf_output, name='control',
    inputs=('x', 'y', 'r'),
    outputs=('u'),
    states=('z'))

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]
ud = Y0[0]
yd = Y0[1]

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

```python
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, 'kd': kd, 'ud': ud, 'yd': yd})
subplots = cruise_plot(cruise_sf, t, y, label='Proportional', linetype='b--')
```

# Response of the system with state feedback + integral action

```python
# Example 11.5: simulate the effect of a (steeper) hill at t = 5 seconds
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, label='Commanded', t_hill=5, antiwindup=True, legend=True)
```

# Example 11.6: add anti-windup compensation

```python
# 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.
```

```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.))
```
486 cruise_plot(cruise_pi, t, y, label='Commanded', t_hill=5, antiwindup=True, legend=True)

487 # If running as a standalone program, show plots and wait before closing
488 import os
489 if __name__ == '__main__' and 'PYCONTROL_TEST_EXAMPLES' not in os.environ:
490     plt.show()
491 else:
492     plt.show(block=False)

Notes

1. The environment variable PYCONTROL_TEST_EXAMPLES is used for testing to turn off plotting of the outputs.

10.1.9 Gain scheduled control for vehicle steering (I/O system)

Code

1 # steering-gainsched.py - gain scheduled control for vehicle steering
2 # RMM, 8 May 2019
3 #
4 # This file works through Example 1.1 in the "Optimization-Based Control"
5 # course notes by Richard Murray (available at http://fbsbook.org, in the
6 # optimization-based control supplement). It is intended to demonstrate the
7 # functionality for nonlinear input/output systems in the python-control
8 # package.
9
10 import numpy as np
11 import control as ct
12 from cmath import sqrt
13 import matplotlib.pyplot as plt
14
15 #
16 # Vehicle steering dynamics
17 #
18 # The vehicle dynamics are given by a simple bicycle model. We take the state
19 # of the system as (x, y, theta) where (x, y) is the position of the vehicle
20 # in the plane and theta is the angle of the vehicle with respect to
21 # horizontal. The vehicle input is given by (v, phi) where v is the forward
22 # velocity of the vehicle and phi is the angle of the steering wheel. The
23 # model includes saturation of the vehicle steering angle.
24 #
25 # System state: x, y, theta
26 # System input: v, phi
27 # System output: x, y
28 # System parameters: wheelbase, maxsteer
29 #
30 def vehicle_update(t, x, u, params):
31     # Get the parameters for the model

(continues on next page)
```
32 l = params.get('wheelbase', 3.)  # vehicle wheelbase
33 phimax = params.get('maxsteer', 0.5)  # max steering angle (rad)
34
35 # Saturate the steering input
36 phi = np.clip(u[1], -phimax, phimax)
37
38 # Return the derivative of the state
39 return np.array([  
40     np.cos(x[2]) * u[0],  # xdot = cos(theta) v  
41     np.sin(x[2]) * u[0],  # ydot = sin(theta) v  
42     (u[0] / l) * np.tan(phi)  # thdot = v/l tan(phi)  
43     ])

46 return x  # return x, y, theta (full state)

54 # Define the vehicle steering dynamics as an input/output system
55 vehicle = ct.NonlinearIOSystem(  
56     vehicle_update, vehicle_output, states=3, name='vehicle',  
57     inputs=('v', 'phi'),  
58     outputs=('x', 'y', 'theta'))

68 def control_output(t, x, u, params):
69     longpole = params.get('longpole', -2.)
70     latpole1 = params.get('latpole1', -1/2 + sqrt(-7)/2)
71     latpole2 = params.get('latpole2', -1/2 - sqrt(-7)/2)
72     l = params.get('wheelbase', 3)
73
74     ex, ey, etheta, vd, phid = u
75
78     alpha1 = -np.real(latpole1 + latpole2)
79     alpha2 = np.real(latpole1 * latpole2)
80
83 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

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, θ).
# 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
# [ yref ] --> trajgen +++-> controller ++-> vehicle ++-> [x, y, theta]
# [ vref ] ^
# |     | |
# #
# +---------+ [-1] --+--+

(continued on next page)
# We construct the system using the InterconnectedSystem constructor and using
# signal labels to keep track of everything.

steering = ct.interconnect(
    # List of subsystems
    (trajgen, controller, vehicle), name='steering',

    # Interconnections between subsystems
    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
plt.figure();

# Plot the reference trajectory for the y position
plt.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
    plt.plot([0, 5], [vref, vref], 'k--')
# Plot the system output

```python
y_line, = plt.plot(tout, yout[y_index, :], 'r')  # lateral position
v_line, = plt.plot(tout, yout[v_index, :], 'b')  # vehicle velocity
```

# Add axis labels
```python
plt.xlabel('Time (s)')
plt.ylabel('x vel (m/s), y pos (m)')
plt.legend((v_line, y_line), ('v', 'y'), loc='center right', frameon=False)
```

## Notes

### 10.1.10 Optimal control for vehicle steering (lane change)

### Code

```python
# steering-optimal.py - optimal control for vehicle steering
# RMM, 18 Feb 2021
#
# This file works through an optimal control example for the vehicle
# steering system. It is intended to demonstrate the functionality for
# optimal control module (control.optimal) in the python-control package.

import numpy as np
import math
import control as ct
import control.optimal as opt
import matplotlib.pyplot as plt
import logging
import time
import os

# 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):
    # 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 (use min/max instead of clip for speed)
phi = max(-phimax, min(u[1], phimax))

# Return the derivative of the state
return np.array([  
    math.cos(x[2]) * u[0], # xdot = cos(theta) v  
    math.sin(x[2]) * u[0], # ydot = sin(theta) v  
    (u[0] / l) * math.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'))

# Utility function to plot the results

# def plot_results(t, y, u, figure=None, yf=None):
    # Plot the xy trajectory
    plt.figure(figure)
    plt.subplot(3, 1, 1)
    plt.plot(y[0], y[1])
    plt.xlabel('x [m]')
    plt.ylabel('y [m]')
    if yf:
        plt.plot(yf[0], yf[1], 'ro')

    # Plot the inputs as a function of time
    plt.subplot(3, 1, 2)
    plt.plot(t, u[0])
    plt.xlabel('t [sec]')
    plt.ylabel('velocity [m/s]')

    plt.subplot(3, 1, 3)
    plt.plot(t, u[1])
    plt.xlabel('t [sec]')
    plt.ylabel('steering [rad/s]')

    plt.suptitle('Lane change maneuver')
    plt.tight_layout()
    plt.show(block=False)

# # Optimal control problem
# Perform a "lane change" maneuver over the course of 10 seconds.
#
# Initial and final conditions
x0 = [0., -2., 0.]; u0 = [10., 0.]
xf = [100., 2., 0.]; uf = [10., 0.]
Tf = 10

# Approach 1: standard quadratic cost
#
# We can set up the optimal control problem as trying to minimize the
# distance from the desired final point while at the same time not
# exerting too much control effort to achieve our goal.
print("Approach 1: standard quadratic cost")

# Set up the cost functions
Q = np.diag([.1, 10, .1])  # keep lateral error low
R = np.diag([.1, 1])  # minimize applied inputs
quad_cost = opt.quadratic_cost(vehicle, Q, R, x0=xf, u0=uf)

# Define the time horizon (and spacing) for the optimization
horizon = np.linspace(0, Tf, 10, endpoint=True)

# Provide an intial guess (will be extended to entire horizon)
bend_left = [10, 0.01]  # slight left veer

# Turn on debug level logging so that we can see what the optimizer is doing
logging.basicConfig(
    level=logging.DEBUG, filename="steering-integral_cost.log",
    filemode='w', force=True)

# Compute the optimal control, setting step size for gradient calculation (eps)
start_time = time.process_time()
result1 = opt.solve_ocp(
    vehicle, horizon, x0, quad_cost, initial_guess=bend_left, log=True,
    minimize_method='trust-constr',
    minimize_options={'finite_diff_rel_step': 0.01},
)
print("* Total time = %.5g seconds\n" % (time.process_time() - start_time))

# If we are running CI tests, make sure we succeeded
if 'PYCONTROL_TEST_EXAMPLES' in os.environ:
    assert result1.success

# Extract and plot the results (+ state trajectory)
t1, u1 = result1.time, result1.inputs
t1, y1 = ct.input_output_response(vehicle, horizon, u1, x0)
plot_results(t1, y1, u1, figure=1, yf=xf[0:2])
# Approach 2: input cost, input constraints, terminal cost

# The previous solution integrates the position error for the entire horizon, and so the car changes lanes very quickly (at the cost of larger inputs). Instead, we can penalize the final state and impose a higher cost on the inputs, resulting in a more graduate lane change.

# We also set the solver explicitly.

**print("Approach 2: input cost and constraints plus terminal cost")**

# Add input constraint, input cost, terminal cost
constraints = [opt.input_range_constraint(vehicle, [8, -0.1], [12, 0.1])] traj_cost = opt.quadratic_cost(vehicle, None, np.diag([0.1, 1]), u0=uf) term_cost = opt.quadratic_cost(vehicle, np.diag([1, 10, 10]), None, x0=xf)

# Change logging to keep less information
logging.basicConfig( level=logging.INFO, filename="./steering-terminal_cost.log", filemode='w', force=True)

# Compute the optimal control
start_time = time.process_time()
result2 = opt.solve_ocp( vehicle, horizon, x0, traj_cost, constraints, terminal_cost=term_cost, initial_guess=bend_left, log=True, minimize_method='SLSQP', minimize_options={'eps': 0.01})

**print("\n* Total time = %g seconds\n" % (time.process_time() - start_time))**

# If we are running CI tests, make sure we succeeded
if 'PYCONTROL_TEST_EXAMPLES' in os.environ: assert result2.success

# Extract and plot the results (+ state trajectory)
t2, u2 = result2.time, result2.inputs
t2, y2 = ct.input_output_response(vehicle, horizon, u2, x0) plot_results(t2, y2, u2, figure=2, yf=xf[0:2])

# Approach 3: terminal constraints
# We can also remove the cost function on the state and replace it with a terminal *constraint* on the state. If a solution is found, it guarantees we get to exactly the final state.

**print("Approach 3: terminal constraints")**

# Input cost and terminal constraints
R = np.diag([1, 1])  # minimize applied inputs
cost3 = opt.quadratic_cost(vehicle, np.zeros((3,3)), R, u0=uf) constraints = [
192 \texttt{opt.input_range_constraint(vehicle, [8, -0.1], [12, 0.1])]
193 terminal = [ \texttt{opt.state_range_constraint(vehicle, xf, xf)} ]

# Reset logging to its default values
194 logging.basicConfig(
195     level=logging.DEBUG, filename="./steering-terminal_constraint.log",
196     filemode='w', force=True)

# Compute the optimal control
197 start_time = time.process_time()
198 result3 = opt.solve_ocp(  
199     vehicle, horizon, x0, cost3, constraints,  
200     terminal_constraints=terminal, initial_guess=bend_left, log=False,  
201     solve_ivp_kwargs={'atol': 1e-3, 'rtol': 1e-2},  
202     minimize_method='trust-constr',
203     )
204 print("* Total time = \%5g seconds\n" % (time.process_time() - start_time))

# If we are running CI tests, make sure we succeeded
205 if 'PYCONTROL_TEST_EXAMPLES' in os.environ:
206     assert result3.success

# Extract and plot the results (+ state trajectory)
207 t3, u3 = result3.time, result3.inputs
208 t3, y3 = ct.input_output_response(vehicle, horizon, u3, x0)
209 plot_results(t3, y3, u3, figure=3, yf=xf[0:2])

# Approach 4: terminal constraints w/ basis functions
210 # As a final example, we can use a basis function to reduce the size
211 # of the problem and get faster answers with more temporal resolution.
212 # Here we parameterize the input by a set of 4 Bezier curves but solve
213 # for a much more time resolved set of inputs.

214 print("Approach 4: Bezier basis")
215 import control.flatsys as flat

# Compute the optimal control
216 start_time = time.process_time()
217 result4 = opt.solve_ocp(  
218     vehicle, horizon, x0, quad_cost,  
219     constraints,  
220     terminal_constraints=terminal,  
221     initial_guess=bend_left,  
222     basis=flat.BezierFamily(4, T=Tf),  
223     solve_ivp_kwargs={'method': 'RK45', 'atol': 1e-2, 'rtol': 1e-2},  
224     solve_ivp_kwargs={'atol': 1e-3, 'rtol': 1e-2},  
225     minimize_method='trust-constr', minimize_options={'disp': True},
226     log=False
227     )
228 print("* Total time = \%5g seconds\n" % (time.process_time() - start_time))

(continues on next page)
# If we are running CI tests, make sure we succeeded
if 'PYCONTROL_TEST_EXAMPLES' in os.environ:
    assert result4.success

# Extract and plot the results (+ state trajectory)
t4, u4 = result4.time, result4.inputs
t4, y4 = ct.input_output_response(vehicle, horizon, u4, x0)
plot_results(t4, y4, u4, figure=4, yf=xf[0:2])

# If we are not running CI tests, display the results
if 'PYCONTROL_TEST_EXAMPLES' not in os.environ:
    plt.show()

Notes

10.1.11 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 differentially 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
import control.optimal as opt

#
# System model and utility functions
#
#
# 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

# 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

# Plot the trajectory in xy coordinates
def plot_results(t, x, ud):
    plt.subplot(4, 1, 2)
```python
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('Time t [sec]')
plt.ylabel(r'$\delta$ [rad]')
plt.tight_layout()
```

(continued from previous page)

# Approach 1: point to point solution, no cost or constraints

# Create differentially flat input/output system
vehicle_flat = fs.FlatSystem(
    vehicle_flat_forward, vehicle_flat_reverse, vehicle_update,
    inputs=('v', 'delta'), outputs=('x', 'y'),
    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, Tf, x0, u0, xf, uf, 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(1)
plt.suptitle("Open loop trajectory for kinematic car lane change")
plot_results(t, x, ud)

# Approach #2: add cost function to make lane change quicker
#
# Define timepoints for evaluation plus basis function to use
timepts = np.linspace(0, Tf, 10)
basis = fs.PolyFamily(8)
# Define the cost function (penalize lateral error and steering)
traj_cost = opt.quadratic_cost(
    vehicle_flat, np.diag([0, 0.1, 0]), np.diag([0.1, 1]), x0=xf, u0=uf)
# Solve for an optimal solution
traj = fs.point_to_point(
    vehicle_flat, timepts, x0, u0, xf, uf, cost=traj_cost, basis=basis,
) xd, ud = traj.eval(T)

plt.figure(2)
plt.suptitle("Lane change with lateral error + steering penalties")
plot_results(T, xd, ud)

# Approach #3: optimal cost with trajectory constraints
#
# Resolve the problem with constraints on the inputs
#
constraints = [
    opt.input_range_constraint(vehicle_flat, [8, -0.1], [12, 0.1])
]
# Solve for an optimal solution
traj = fs.point_to_point(
    vehicle_flat, timepts, x0, u0, xf, uf, cost=traj_cost,
    constraints=constraints, basis=basis,
) xd, ud = traj.eval(T)

plt.figure(3)
plt.suptitle("Lane change with penalty + steering constraints")
plot_results(T, xd, ud)

# Show the results unless we are running in batch mode
if 'PYCONTROL_TEST_EXAMPLES' not in os.environ:
Notes

1. The environment variable `PYCONTROL_TEST_EXAMPLES` is used for testing to turn off plotting of the outputs.

10.2 Jupyter notebooks

The examples below use `python-control` in a Jupyter notebook environment. These notebooks demonstrate the use of modeling, analysis, and design tools using running examples in FBS2e.

10.2.1 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
```

(continues on next page)
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
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: $F_{a} = \frac{1}{2} \rho C_d A |v| v$, where $\rho$ is the density of air, $C_d$ is the shape-dependent aerodynamic drag coefficient, and $A$ is the frontal area of the car.

$$F_a = \frac{1}{2} \rho C_d A |v| v$$

# Final acceleration on the car

$$F_d = F_g + F_r + F_a$$

$$dv = \frac{(F - F_d)}{m}$$

```python
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 $0 \leq u \leq 1$ that controls the throttle position. The torque also depends on engine speed $\omega$.

```python
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.

```python
fig, axes = plt.subplots(1, 2, figsize=(7, 3))

# (a) - single torque curve as function of omega
ax = axes[0]
omega = np.linspace(0, 700, 701)
ax.plot(omega, motor_torque(omega))
ax.set_xlabel(r'Angular velocity $\omega$ [rad/s]')
ax.set_ylabel('Torque $T$ [Nm]')
ax.grid(True, linestyle='dotted')

# (b) - torque curves in different gears, as function of velocity
ax = axes[1]
v = np.linspace(0, 70, 71)
alpha = [40, 25, 16, 12, 10]
for gear in range(5):
    omega = alpha[gear] * v
    T = motor_torque(omega)
    plt.plot(v, T, color='#1f77b4', linestyle='solid')
```

(continues on next page)
# Set up the axes and style
ax.axis([0, 70, 100, 200])
ax.grid(True, linestyle='dotted')

# Add labels
plt.text(11.5, 120, '\$n$=1')
ax.text(24, 120, '\$n$=2')
ax.text(42.5, 120, '\$n$=3')
ax.text(58.5, 120, '\$n$=4')
ax.text(58.5, 185, '\$n$=5')
ax.set_xlabel('Velocity $v$ [m/s]
ax.set_ylabel('Torque $T$ [Nm]
plt.suptitle('Torque curves for typical car engine')
plt.tight_layout()

---

**Input/output 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 $\theta$ as input. The output of this model is the current vehicle velocity $v$.

```python
[5]: vehicle = ct.NonlinearIOSystem(
    vehicle_update, None, name='vehicle',
    inputs=('u', 'gear', 'theta'), outputs=('v'), states=('v'))

# Define a function for creating a "standard" cruise control plot
def cruise_plot(sys, t, y, label=None, t_hill=None, vref=20, antiwindup=False,
    linetype='b-', subplots=None, legend=None):
    if subplots is None:
        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
    ```
```python
while min(y[v_ind]) < v_min: v_min -= 1

# Create arrays for return values
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-')
if t_hill:
    plt.axvline(t_hill, color='k', linestyle='--', label='t hill')
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, label=label)
# Applied input profile
if antiwindup:
    plt.plot(t, np.clip(y[u_ind], 0, 1), linetype, label='Applied')
if t_hill:
    plt.axvline(t_hill, color='k', linestyle='--')
if legend:
    plt.legend(frameon=False)
plt.axis([0, t[-1], u_min, u_max])
plt.xlabel('Time $t$ [s]')
plt.ylabel('Throttle $u$ [$^\dagger$]')

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.
```python
[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
vehicle = ct.Dynamics('vehicle',
    states=['x', 'y', 'r'],
    outputs=['u', 'v'],
    inplist=['r', 'gear', 'theta'],
    outlist=['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
linearized = ct.linearize(vehicle, Xeq, [Ueq[0], gear[0], 0])
```

10.2. Jupyter notebooks
# 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)

# 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
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_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 \( P(s) = \frac{b}{s + a} \). A simple (but not necessarily good) way to design a PI controller is to choose the parameters of the PI controller as \( k_i = a k_p \). The controller transfer function is then \( C(s) = k_p + k_i/s = k_i(s + a)/s \). It has a zero at \( s = -k_i/k_p = -a \) that cancels the process pole at \( s = -a \). We have \( P(s)C(s) = k_i/s \) giving the transfer function from reference to vehicle velocity as \( G_{yr}(s) = bk_p/(s + bk_p) \), 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 \( 1/(bk_p) \).

[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

$$s^2 + 2\zeta\omega_0 s + \omega_0^2$$

# Values of the first order transfer function $P(s) = \frac{b}{s + a}$ are set above

# Define the input that we want to track
T = np.linspace(0, 40, 101)
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

(continues on next page)
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'))

# 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',
        (continued on next page)
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)

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)

(continues on next page)
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([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]');
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
```

(continues on next page)
# PI controller

```python
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.

```text
[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, t_hill=5);
```
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.

```python
[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]
t, y = ct.input_output_response(
    cruise_pi, T, [vref, gear, theta_hill], X0,
    params={'kaw':0})
```

Car with cruise control encountering sloping road

![Graph showing car with cruise control encountering sloping road. The graph plots velocity (in m/s) and throttle over time.](image)

304 Chapter 10. Examples
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.

```python
[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, label='Commanded', t_hill=5,
            antiwindup=True, legend=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.
10.2.2 Describing function analysis

Richard M. Murray, 27 Jan 2021

This Jupyter notebook shows how to use the `descfcn` module of the Python Control Toolbox to perform describing function analysis of a nonlinear system. A brief introduction to describing functions can be found in Feedback Systems, Section 10.5 (Generalized Notions of Gain and Phase).

```python
import control as ct
import numpy as np
import matplotlib.pyplot as plt
import math
```

**Built-in describing functions**

The Python Control Toolbox has a number of built-in functions that provide describing functions for some standard nonlinearities.

**Saturation nonlinearity**

A saturation nonlinearity can be obtained using the `ct.saturation_nonlinearity` function. This function takes the saturation level as an argument.

```python
saturation = ct.saturation_nonlinearity(0.75)
x = np.linspace(-2, 2, 50)
plt.plot(x, saturation(x))
plt.xlabel("Input, x")
plt.ylabel("Output, y = sat(x)")
plt.title("Input/output map for a saturation nonlinearity");
```
Backlash nonlinearity

A friction-dominated backlash nonlinearity can be obtained using the `ct.friction_backlash_nonlinearity` function. This function takes as an argument the size of the backlash region.
amp_range = np.linspace(0, 2, 50)
N_a = ct.describing_function(backlash, amp_range)

plt.figure()
plt.plot(amp_range, abs(N_a))
plt.xlabel("Amplitude A")
plt.ylabel("Amplitude of describing function, N(A)")
plt.title("Describing function for a backlash nonlinearity")

plt.figure()
plt.plot(amp_range, np.angle(N_a))
plt.xlabel("Amplitude A")
plt.ylabel("Phase of describing function, N(A)")
plt.title("Describing function for a backlash nonlinearity");
User-defined, static nonlinearities

In addition to pre-defined nonlinearities, it is possible to computing describing functions for static nonlinearities. The describing function for any suitable nonlinear function can be computed numerically using the `describing_function` function.

```python
# Define a saturation nonlinearity as a simple function
def my_saturation(x):
    if abs(x) >= 1:
        return math.copysign(1, x)
    else:
        return x

amp_range = np.linspace(0, 2, 50)
plt.plot(amp_range, ct.describing_function(my_saturation, amp_range).real)
plt.xlabel("Amplitude A")
plt.ylabel("Describing function, N(A)")
plt.title("Describing function for a saturation nonlinearity");
```
Stability analysis using describing functions

Describing functions can be used to assess stability of closed loop systems consisting of a linear system and a static nonlinear using a Nyquist plot.

Limit cycle position for a third order system with saturation nonlinearity

Consider a nonlinear feedback system consisting of a third-order linear system with transfer function \( H(s) \) and a saturation nonlinearity having describing function \( N(a) \). Stability can be assessed by looking for points at which

\[
H(j\omega)N(a) = -1
\]

The \texttt{describing_function_plot} function plots \( H(j\omega) \) and \(-1/N(a)\) and prints out the the amplitudes and frequencies corresponding to intersections of these curves.

```python
# Linear dynamics
H_simple = ct.tf([8], [1, 2, 2, 1])
omega = np.logspace(-3, 3, 500)

# Nonlinearity
F_saturation = ct.saturation_nonlinearity(1)
amp = np.linspace(00, 5, 50)

describing_function_plot(H_simple, F_saturation, amp, omega)
```

```
[[3.343977839598768, 1.4142156916757294]]
```
The intersection occurs at amplitude 3.3 and frequency 1.4 rad/sec (= 0.2 Hz) and thus we predict a limit cycle with amplitude 3.3 and period of approximately 5 seconds.

```python
# Create an I/O system simulation to see what happens
io_saturation = ct.NonlinearIOSystem(
    None,
    lambda t, x, u, params: F_saturation(u),
    inputs=1, outputs=1
)

sys = ct.feedback(ct.tf2io(H_simple), io_saturation)
T = np.linspace(0, 30, 200)
t, y = ct.input_output_response(sys, T, 0.1, 0)
plt.plot(t, y);
```

10.2. Jupyter notebooks
Limit cycle prediction with for a time-delay system with backlash

This example demonstrates a more complicated interaction between a (non-static) nonlinearity and a higher order transfer function, resulting in multiple intersection points.

[9]:

```python
# Linear dynamics
H_simple = ct.tf([1], [1, 2, 2, 1])
H_multiple = H_simple * ct.tf("ct.pade(5, 4)") * 4
omega = np.logspace(-3, 3, 500)

# Nonlinearity
F_backlash = ct.friction_backlash_nonlinearity(1)
amp = np.linspace(0.6, 5, 50)

# Describing function plot
ct.describing_function_plot(H_multiple, F_backlash, amp, omega, mirror_style=False)
```

[9]:

```
[(0.6260158833531679, 0.31026194979692245),
 (0.8741930326842812, 1.215641094477062)]
```

10.2.3 Discrete Time Sensor Fusion

RMM, 24 Feb 2022

In this example we work through estimation of the state of a car changing lanes with two different sensors available: one with good longitudinal accuracy and the other with good lateral accuracy.

All calculations are done in discrete time, using both a Kalman filter formulation and predictor-corrector form.

[17]:

```
import numpy as np
import scipy as sp
import matplotlib.pyplot as plt
import control as ct
import control.optimal as opt
```
import control.flatsys as fs
from IPython.display import Image

# Define line styles
ebarstyle = {'elinewidth': 0.5, 'capsize': 2}
xdstyle = {'color': 'k', 'linestyle': '--', 'linewidth': 0.5,
          'marker': '+', 'markersize': 4}

System definition

We consider a bicycle model for an automobile:

Continuous time model

The dynamics are given by

\[
\begin{align*}
\dot{x} &= \cos \theta \, v, \\
\dot{y} &= \sin \theta \, v, \\
\dot{\theta} &= \frac{v}{l} \tan \phi,
\end{align*}
\]

where \((x, y, \theta)\) are the position and orientation of the vehicle, \(v\) is the forward velocity, \(\phi\) is the steering wheel angle, and \(l\) is the wheelbase.

These dynamics are included in the file vehicle.py:

```python
# Vehicle steering dynamics
# System state: x, y, theta
# System input: v, phi
# System output: x, y
# System parameters: wheelbase, maxsteer

from vehicle import vehicle, plot_lanechange
print(vehicle)
```

Object: vehicle
Inputs (2): v, delta,
Outputs (3): x, y, theta,
States (3): x, y, theta,

This system is differentially flat and so we can define a trajectory for the system using the flatsys module. We generate a motion that corresponds to changing lanes on a road:

```python
# Generate a trajectory for the vehicle
# Define the endpoints of the trajectory
x0 = [0., -2., 0.]; u0 = [10., 0.]
xf = [40., 2., 0.]; uf = [10., 0.]
Tf = 4

# Find a trajectory between the initial condition and the final condition
traj = fs.point_to_point(vehicle, Tf, x0, u0, xf, uf, basis=fs.PolyFamily(6))

# Create the desired trajectory between the initial and final condition
Ts = 0.1
# Ts = 0.5
T = np.arange(0, Tf + Ts, Ts)
xd, ud = traj.eval(T)
plot_lanechange(T, xd, ud)
```

**Discrete time system model**

For the model that we use for the Kalman filter, we take a simple discretization using the approximation that \( \dot{x} = (x[k+1] - x[k])/T_s \) where \( T_s \) is the sampling time.

```python
# Create a discrete time, linear model
#
```
# Linearize about the starting point
linsys = ct.linearize(vehicle, x0, u0)

# Create a discrete time model by hand
Ad = np.eye(linsys.nstates) + linsys.A * Ts
Bd = linsys.B * Ts
discsys = ct.LinearIOSystem(ct.ss(Ad, Bd, np.eye(linsys.nstates), 0, dt=Ts))
print(discsys)

# Sensor model

We assume that we have two sensors: one with good longitudinal accuracy and the other with good lateral accuracy. For each sensor we define the map from the state space to the sensor outputs, the covariance matrix for the measurements, and a white noise signal (now in discrete time).

# Sensor #1: longitudinal
C_lon = np.eye(2, discsys.nstates)
Rw_lon = np.diag([0.1 ** 2, 1 ** 2])
W_lon = ct.white_noise(T, Rw_lon, dt=Ts)

# Sensor #2: lateral
C_lat = np.eye(2, discsys.nstates)
Rw_lat = np.diag([1 ** 2, 0.1 ** 2])
W_lat = ct.white_noise(T, Rw_lat, dt=Ts)

# Plot the noisy signals
plt.subplot(2, 1, 1)
Linear Quadratic Estimator

To estimate the position of the vehicle, we construct an optimal estimator (Kalman filter).

```
[22]:
  # Create an estimator for the system
  #
  # Disturbance and initial condition model
  Rv = np.diag([[0.1, 0.01]]) * Ts
  # Rv = np.diag([10, 0.1]) * Ts # No input data
  P0 = np.diag([1, 1, 0.1])

  # Combine the sensors
  C = np.vstack([C_lon, C_lat])
  Rw = sp.linalg.block_diag(Rw_lon, Rw_lat)
```
Python Control Library Documentation, Release dev

(continued from previous page)

estim = ct.create_estimator_iosystem(discsys, Rv, Rw, C=C, P0=P0)
print(estim)
Object: sys[7]
Inputs (6): y[0], y[1], y[2], y[3], u[0], u[1],
Outputs (3): xhat[0], xhat[1], xhat[2],
States (12): xhat[0], xhat[1], xhat[2], P[0,0], P[0,1], P[0,2], P[1,0], P[1,1], P[1,2],␣
˓→P[2,0], P[2,1], P[2,2],
Finally, we estimate the position of the vehicle based on sensor measurements. We assume that the input to the vehicle
(velocity and steering angle) is available, though we can also explore what happens if that information is not available
(see commented out code).
We also carry out a prediction of the position of the vehicle by turning off the correction term in the Kalman filter.
[23]: # Compute the inputs to the estimator
Y = np.vstack([xd[0:2] + W_lon, xd[0:2] + W_lat])
U = np.vstack([Y, ud])
# add input to the Kalman filter
# U = np.vstack([Y, ud * 0]) # with no input information
X0 = np.hstack([xd[:, 0], P0.reshape(-1)])
# Run the estimator on the trajectory
estim_resp = ct.input_output_response(estim, T, U, X0)
# Run a prediction to see what happens next
T_predict = np.arange(T[-1], T[-1] + 4 + Ts, Ts)
U_predict = np.outer(U[:, -1], np.ones_like(T_predict))
predict_resp = ct.input_output_response(
estim, T_predict, U_predict, estim_resp.states[:, -1],
params={'correct': False})
# Plot the estimated trajectory versus the actual trajectory
plt.subplot(2, 1, 1)
plt.errorbar(
estim_resp.time, estim_resp.outputs[0],
estim_resp.states[estim.find_state('P[0,0]')], fmt='b-', **ebarstyle)
plt.errorbar(
predict_resp.time, predict_resp.outputs[0],
predict_resp.states[estim.find_state('P[0,0]')], fmt='r-', **ebarstyle)
plt.plot(T, xd[0], 'k--')
plt.ylabel("$x$ position [m]")
plt.subplot(2, 1, 2)
plt.errorbar(
estim_resp.time, estim_resp.outputs[1],
estim_resp.states[estim.find_state('P[1,1]')], fmt='b-', **ebarstyle)
plt.errorbar(
predict_resp.time, predict_resp.outputs[1],
predict_resp.states[estim.find_state('P[1,1]')], fmt='r-', **ebarstyle)
# lims = plt.axis(); plt.axis([lims[0], lims[1], -5, 5])
plt.plot(T, xd[1], 'k--');
plt.ylabel("$y$ position [m]")
plt.xlabel("Time $t$ [s]");

10.2. Jupyter notebooks

317


More insight can be obtained by focusing on the errors in prediction:

```python
[24]: # Plot the estimated errors
    plt.subplot(2, 1, 1)
    plt.errorbar(
        estim_resp.time, estim_resp.outputs[0] - xd[0],
        estim_resp.states[estim.find_state('P[0,0]')], fmt='b-', **ebarstyle)
    plt.errorbar(
        predict_resp.time, predict_resp.outputs[0] - (xd[0] + xd[0, -1]),
        predict_resp.states[estim.find_state('P[0,0]')], fmt='r-', **ebarstyle)
    lims = plt.axis(); plt.axis([lims[0], lims[1], -0.2, 0.2])

    # lims = plt.axis(); plt.axis([lims[0], lims[1], -2, 0.2])

    plt.subplot(2, 1, 2)
    plt.errorbar(
        estim_resp.time, estim_resp.outputs[1] - xd[1],
        estim_resp.states[estim.find_state('P[1,1]')], fmt='b-', **ebarstyle)
    plt.errorbar(
        predict_resp.time, predict_resp.outputs[1] - xd[1, -1],
        predict_resp.states[estim.find_state('P[1,1]')], fmt='r-', **ebarstyle)
    lims = plt.axis(); plt.axis([lims[0], lims[1], -0.2, 0.2]);
```
**Things to try**

To gain a bit more insight into sensor fusion, you can try the following: * Remove the input (and update P0) * Change the sampling rate

**Predictor-corrector form**

Instead of using `create_estimator_iopystem`, we can also compute out the estimate in a more manual fashion, done here using the predictor-corrector form:

```python

# Create an array to store the results
xhat = np.zeros((discsys.nstates, T.size))
P = np.zeros((discsys.nstates, discsys.nstates, T.size))

# Update the estimates at each time
for i, t in enumerate(T):
    # Prediction step
    if i == 0:
        # Use the initial condition
        xkkm1 = xd[:, 0]
Pkkm1 = P0
    else:
        xkkm1 = A @ xkk + B @ ud[:, i-1]
Pkkm1 = A @ Pkk @ A.T + F @ Rv @ F.T

    # Correction step
    L = Pkkm1 @ C.T @ np.linalg.inv(Rw + C @ Pkkm1 @ C.T)
xkk = xkkm1 - L @ (C @ xkkm1 - Y[:, i])
Pkk = Pkkm1 - L @ C @ Pkkm1
```

(continues on next page)
# Save the state estimate and covariance for later plotting
xhat[:, i], P[:, :, i] = xkk, Pkk
# xhat[:, i], P[:, :, i] = xkkml, Pkkml # For comparison to Kalman form

plt.subplot(2, 1, 1)
plt.errorbar(T, xhat[0], P[0, 0], fmt='b-', **ebarstyle)
plt.plot(T, xd[0], 'k--')
plt.ylabel(r"$x$ position [m]"")

plt.subplot(2, 1, 2)
plt.errorbar(T, xhat[1], P[1, 1], fmt='b-', **ebarstyle)
plt.plot(T, xd[1], 'k--')
plt.ylabel(r"$x$ position [m]"");

We can compare the results of the predictor-corrector form to the Kalman filter form used at the top of the notebook:

[26]: # Plot the estimated errors (and compare to Kalman form)
plt.subplot(2, 1, 1)
plt.errorbar(T, xhat[0] - xd[0], P[0, 0], fmt='b-', **ebarstyle)
plt.plot(estim_resps.time, estim_resps.outputs[0] - xd[0], 'r--')
lims = plt.axis(); plt.axis([lims[0], lims[1], -0.2, 0.2])

plt.subplot(2, 1, 2)
plt.errorbar(T, xhat[1] - xd[1], P[1, 1], fmt='b-', **ebarstyle)
plt.plot(estim_resps.time, estim_resps.outputs[1] - xd[1], 'r--')
lims = plt.axis(); plt.axis([lims[0], lims[1], -0.2, 0.2]);
Note that the estimates are not the same! It turns out that to get the correspondence of the two formulations, we need to define $\hat{x}_{KF}(k) = \hat{x}_{PC}(k|k-1)$ (see commented out code above).

10.2.4 Model Predictive Control: Aircraft Model

RMM, 13 Feb 2021

This example replicates the MPT3 regulation problem example.

```python
import control as ct
import numpy as np
import control.optimal as opt
import matplotlib.pyplot as plt

# model of an aircraft discretized with 0.2s sampling time
# Source: https://www.mpt3.org/UI/RegulationProblem
A = [[0.99, 0.01, 0.18, -0.09, 0],
     [0, 0.94, 0, 0.29, 0],
     [0, 0.14, 0.81, -0.9, 0],
     [0, -0.2, 0, 0.95, 0],
     [0, 0.09, 0, 0, 0.9]]
B = [[0.01, -0.02],
     [-0.14, 0],
     [0.05, -0.2],
     [0.02, 0],
     [-0.01, 0]]
C = [[0, 1, 0, 0, -1],
     [0, 0, 1, 0, 0],
     [0, 0, 0, 1, 0],
     [1, 0, 0, 0, 0]]
model = ct.ss2io(ct.ss(A, B, C, 0, 0.2))

# For the simulation we need the full state output
(continues on next page)
sys = ct.ss2io(ct.ss(A, B, np.eye(5), 0, 0.2))

# compute the steady state values for a particular value of the input
ud = np.array([0.8, -0.3])
xd = np.linalg.inv(np.eye(5) - A) @ B @ ud
yd = C @ xd

[4]:
# computed values will be used as references for the desired
# steady state which can be added using "reference" filter
# model.u.with('reference');
# model.u.reference = us;
# model.y.with('reference');
# model.y.reference = ys;

# provide constraints on the system signals
constraints = [opt.input_range_constraint(sys, [-5, -6], [5, 6])]

# provide penalties on the system signals
Q = model.C.transpose() @ np.diag([10, 10, 10, 10]) @ model.C
R = np.diag([3, 2])

cost = opt.quadratic_cost(model, Q, R, x0=xd, u0=ud)

# online MPC controller object is constructed with a horizon 6
ctrl = opt.create_mpc_iosystem(model, np.arange(0, 6) * 0.2, cost, constraints)

[7]:
# Define an I/O system implementing model predictive control
loop = ct.feedback(sys, ctrl, 1)
print(loop)

System: sys[7]
Inputs (2): u[0], u[1],
Outputs (5): y[0], y[1], y[2], y[3], y[4],
States (17): sys[1].x[0], sys[1].x[1], sys[1].x[2], sys[1].x[3], sys[1].x[4], sys[6]...
...
sys[6].x[7], sys[6].x[8], sys[6].x[9], sys[6].x[10], sys[6].x[11],

[9]:
import time

# loop = ClosedLoop(ctrl, model);
# x0 = [0, 0, 0, 0, 0]
Nsim = 60

start = time.time()
tout, xout = ct.input_output_response(loop, np.arange(0, Nsim) * 0.2, 0, 0)
end = time.time()
print("Computation time = %g seconds" % (end-start))

Computation time = 8.28132 seconds

[10]:
# Plot the results
# plt.subplot(2, 1, 1)
for i, y in enumerate(C @ xout):
plt.plot(tout, y)
plt.plot(tout, yd[i] * np.ones(tout.shape), 'k--')
plt.title('outputs')

# plt.subplot(2, 1, 2)
# plt.plot(t, u);
# plot(np.range(Nsim), us*ones(1, Nsim), 'k--')
# plt.title('inputs')

plt.tight_layout()

# Print the final error
xd - xout[:,-1]

[10]: array([-0.15441833, 0.00362039, 0.07760278, 0.00675162, 0.00698118])

10.2.5 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*.

[1]: import numpy as np
import matplotlib.pyplot as plt
import control as ct
import control.optimal as opt
ct.use_fbs_defaults()
Vehicle steering dynamics (Example 3.11)

The vehicle dynamics are given by a simple bicycle model. We take the state of the system as \((x, y, \theta)\) where \((x, y)\) is the position of the reference point of the vehicle in the plane and \(\theta\) is the angle of the vehicle with respect to horizontal. The vehicle input is given by \((v, \delta)\) where \(v\) is the forward velocity of the vehicle and \(\delta\) 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 (\(\text{maxsteer}\)).

- System state: \(x, y, \theta\)
- System input: \(v, \delta\)
- System output: \(x, y\)
- System parameters: \(\text{wheelbase}, \text{refoffset}, \text{maxsteer}\)

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 \(\delta\). To compute the angle \(\alpha\) 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 \(r\), and it follows from Figure 3.17 in FBS that \(b = r \tan \delta\) and \(a = r \tan \alpha\), 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 use a limit of 0.1 radians (about 6 degrees) at 10 m/s (≈ 35 kph) and 0.05 radians (about 3 degrees) at 30 m/s (≈ 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 (≈ 1.5 times the force of gravity).

```python
[2]: 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.

```python
[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]')
    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 $\theta$ 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
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:
")
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:

\[
A = \begin{bmatrix}
0 & 1 \\
0 & 0
\end{bmatrix}
\]
\[
B = \begin{bmatrix}
0.5 \\
1
\end{bmatrix}
\]
\[
C = \begin{bmatrix}
1 & 0
\end{bmatrix}
\]
\[
D = \begin{bmatrix}
0
\end{bmatrix}
\]

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 \( p(s) = s^2 + 2\zeta \omega_c s + \omega_c^2 \).

To find reasonable values of \( \omega_c \) 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 \( \omega_c \) is shown on the left, which shows that the response speed increases with increasing \( \omega_c \). 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 \( \zeta_c \) 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 \( \zeta_c = 0.7 \).

[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(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)

    # 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 \( \omega_c = 0.07 \)

```python
outfig = plt.subplot(2, 2, 2)
infig = 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

```python
normalized_label(inpfig, outfig)
plt.legend((r'$\zeta_c = 0.5$', r'$\zeta_c = 0.7$', r'$\zeta_c = 1$'))
plt.tight_layout()
```
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.

We find the eigenvalue from the characteristic polynomial

\[
\text{wo} = 1 \quad \text{# bandwidth for the observer} \\
\text{zo} = 0.7 \quad \text{# damping ratio for the observer} \\
eigs = \text{np.roots}([1, 2*\text{zo}*\text{wo}, \text{wo}^2])
\]

Compute the estimator gain using eigenvalue placement

\[
\text{L} = \text{np.transpose}(
    \text{ct.place}(
        \text{np.transpose}([A, B, C]), \text{eigs})
    )
\]

Create a linear model of the lateral dynamics driving the estimator

\[
\text{est} = \text{ct.StateSpace}([A - \text{L} @ C, \text{np.block}([B, \text{L}]), \text{np.eye}(2), \text{np.zeros}(2, 2))
\]

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.

Convert the curvy trajectory into normalized coordinates

\[
\text{x}_\text{ref} = x_\text{curvy}[0] / \text{wheelbase} \\
\text{y}_\text{ref} = x_\text{curvy}[1] / \text{wheelbase} \\
\text{theta}_\text{ref} = x_\text{curvy}[2] \\
\text{tau} = v_0 * T_\text{curvy} / b 
\]

Simulate the estimator, with a small initial error in y position

\[
\text{y}_\text{est}, \text{x}_\text{est} = \text{ct.forced_response}([\text{est}, \text{tau}, [\text{delta}_\text{curvy}, \text{y}_\text{ref}], [0.5, 0], \text{return_} \Rightarrow \text{x}=\text{True}) 
\]

Configure matplotlib plots to be a bit bigger and optimize layout

\[
\text{plt.figure}([9, 4.5])
\]

Plot the actual and estimated states

\[
\text{ax} = \text{plt.subplot}(2, 2, 1) \\
\text{plt.plot}(\text{t}, \text{y}_\text{ref}) \\
\text{plt.plot}(\text{t}, \text{x}_\text{est}[0]) \\
\text{ax.set}(\text{xlim}=[0, 10]) \\
\text{plt.legend}(['\text{actual}', '\text{estimated}']) \\
\text{plt.ylabel}('\text{Lateral position} \ y/b') \\
\text{ax} = \text{plt.subplot}(2, 2, 2) \\
\text{plt.plot}(\text{t}, \text{x}_\text{est}[0] - \text{y}_\text{ref}) \\
\text{ax.set}(\text{xlim}=[0, 10]) \\
\text{plt.ylabel}('\text{Lateral error}')
\]

(continues on next page)
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)

[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)))

    (continues on next page)
# Simulate the system

```python
t, y, x = ct.forced_response(clsys, tau, y_ref, [0.4, 0, 0.0, 0], return_x=True)
```

# Calculate the input used to generate the control response

```python
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

```python
plt.figure(figsize=[9, 4.5])
```

# Plot the actual and estimated states

```python
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 $v_0 t / b$')
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 = 0.4899999999999182
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.

[9]: import control.flatsys as fs

```python
# 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)
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)

[10]: def plot_vehicle_lanechange(traj):
    # 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]')
    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.xlabel('t')
    plt.plot([0, 0], [0, x[0, -1]], 'k--', linewidth=1)
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 $\delta$ at the endpoints.

[11]:

```python
# 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(8)

# Find a trajectory between the initial condition and the final condition
traj1 = fs.point_to_point(vehicle_flat, Tf, x0, u0, xf, uf, basis=poly)
plot_vehicle_lanechange(traj1)
```
Change of basis function

\[\text{bezier} = \text{fs.BezierFamily}(8)\]
\[\text{traj2} = \text{fs.point_to_point(vehicle\_flat, Tf, x0, u0, xf, uf, basis=bezier)}\]
\[\text{plot\_vehicle\_lanechange(traj2)}\]

Added cost function

\[\text{timepts = np.linspace(0, Tf, 12)}\]
\[\text{poly = fs.PolyFamily(8)}\]
\[\text{traj\_cost = opt.quadratic\_cost(}\]
\[\text{vehicle\_flat, np.diag([0, 0.1, 0]), np.diag([0.1, 10]), x0=xf, u0=uf)}\]
\[\text{constraints = [}\]
\[\text{opt.input\_range\_constraint(vehicle\_flat, [8, -0.1], [12, 0.1]) ]}\]
\[\text{traj3 = fs.point\_to\_point(}\]
\[\text{vehicle\_flat, timepts, x0, u0, xf, uf, cost=traj\_cost, basis=poly)}\]
\[\text{plot\_vehicle\_lanechange(traj3)}\]
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.

```
[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 reverse 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 = $\frac{s + 1.333}{s^2 + 7.828\times10^{-16} s - 1.848\times10^{-16}}$

Reverse TF = $\frac{-s + 1.333}{s}$
\[ s^2 - 7.828 \times 10^{-16} s - 1.848 \times 10^{-16} \]

[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 $F_m(s) = a^2 2/(s + a)^2$, where the response speed or aggressiveness of the steering is governed by the parameter $a$.

```python
# 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
y_ffwd = ct.step_response(Fm, np.linspace(0, 25, 100))
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.ylabel('Normalized position y/b')
```

(continues on next page)
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 $\zeta_o = 0.707$.

[14]:

```python
# 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(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)
```

(continues on next page)
# Compute the loop transfer function and plot Nyquist, Bode

\[ L_1 = P \cdot C_1 \]

plt.figure(); ct.nyquist_plot(L1, np.logspace(0.5, 3, 500))
plt.figure(); ct.bode_plot(L1, np.logspace(-1, 3, 500));

\[
K = [100. \quad -35.86] \\
L = [28.28 \quad 400. ] \\
C(s) = \\
\frac{-1.152e+04 \quad s + 4e+04}{s^2 + 42.42 \quad s + 6658}
\]

[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(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
c.t.gangof4(P, C1, np.logspace(-1, 3, 100))
c.t.gangof4(P, C2, np.logspace(-1, 3, 100))

10.2.6 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:

\[
\begin{align*}
    m\ddot{x} &= F_1 \cos \theta - F_2 \sin \theta - c \dot{x}, \\
    m\ddot{y} &= F_1 \sin \theta + F_2 \cos \theta - mg - c \dot{y}, \\
    J\ddot{\theta} &= r F_1.
\end{align*}
\]

The position and orientation of the center of mass of the aircraft is denoted by \( (x, y, \theta) \), \( m \) 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:

\[
\frac{dz}{dt} = \begin{bmatrix}
    z_4 \\
    z_5 \\
    z_6 \\
    -c/m z_4 \\
    -g - c/m z_5 \\
    0
\end{bmatrix} + \begin{bmatrix}
    0 \\
    0 \\
    0 \\
    1/m \cos \theta F_1 + 1/m \sin \theta F_2 \\
    1/m \sin \theta F_1 + 1/m \cos \theta F_2 \\
    r F_1
\end{bmatrix}
\]

### 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 $z_e = (0, 0, 0, 0, 0)$ are given by

$$\frac{dz}{dt} = \begin{bmatrix} z_4 \\ z_5 \\ z_6 \\ -g \sin z_3 - \frac{c}{m} z_4 \\ g(\cos z_3 - 1) - \frac{c}{m} z_5 \end{bmatrix} \quad A = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & -g & -c/m & 0 & 0 \\ 0 & 0 & 0 & 0 & -c/m & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

[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],
     [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

$$J = \int_0^\infty (\xi^T Q_x \xi + v^T Q_u v)dt,$$

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])
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)'`
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:

```python
# Look at different input weightings
Qu1a = diag([1, 1])
K1a, X, E = lqr(A, B, Qx1, Qu1a)
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)

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()
```

![Step Response for Inputs](image.png)
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:

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:

```python
from matplotlib.pyplot import *  # Grab MATLAB plotting functions
from control.matlab import *  # MATLAB-like functions

# 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)

# Transfer functions for dynamics
Pi = tf([r], [J, 0, 0])  # inner loop (roll)
Po = tf([1], [m, c, 0])  # outer loop (position)

# Lead compensator
k = 200
a = 2
b = 50
Ci = k * tf([1, a], [1, b])  # lead compensator
Li = Pi * Ci
```

For the inner loop, use a lead compensator.
The closed loop dynamics of the inner loop, $H_i$, are given by

\[
H_i = \parallel(\text{feedback}(C_i, P_i), \ -m^*g^*\text{feedback}(C_i*P_i, 1))
\]

Finally, we design the lateral compensator using another lead compensator

\[
\# \text{Now design the lateral control system}
\]
\[
a = 0.02 \\
b = 5 \\
K = 2 \\
Co = -K*\text{tf}([1, 0.3], [1, 10]) \quad \# \text{another lead compensator} \\
Lo = -m^*g^*P_o*Co
\]

The performance of the system can be characterized using the sensitivity function and the complementary sensitivity function:

\[
L = Co*Hi*P_o \\
S = \text{feedback}(1, L) \\
T = \text{feedback}(L, 1)
\]

\[
\text{t, y = step}(T, \text{T=linspace}(0,10,100)) \\
\text{plot}(y, \text{t}) \\
\text{title}("\text{Step Response}") \\
\text{grid}() \\
\text{xlabel}("\text{time (s)}") \\
\text{ylabel}("y(t)") \\
\text{show}()
\]

The frequency response and Nyquist plot for the loop transfer function are computed using the commands

\[
\text{bode}(L) \\
\text{show}()
\]
```python
[19]: nyquist(L, (0.0001, 1000))
    show()
```

```python
[20]: gangof4(Hi*Po, Co)
```
10.2.7 Output feedback control using LQR and extended Kalman filtering

RMM, 14 Feb 2022

This notebook illustrates the implementation of an extended Kalman filter and the use of the estimated state for LQR feedback of a vectored thrust aircraft model.

[11]:
```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as patches
import control as ct
```

System definition

We consider a (planar) vertical takeoff and landing aircraft model:

![Diagram of vectored thrust aircraft model]

The dynamics of the system with disturbances on the $x$ and $y$ variables are given by

\[
\begin{align*}
mx'' &= F_1 \cos \theta - F_2 \sin \theta - c\dot{x} + d_x, \\
m\dot{y} &= F_1 \sin \theta + F_2 \cos \theta - c\dot{y} - mg + d_y, \\
J\ddot{\theta} &= rF_1,
\end{align*}
\]

```
The measured values of the system are the position and orientation, with added noise $n_x, n_y, \text{ and } n_\theta$:

$$\vec{y} = \begin{bmatrix} x \\ y \\ \theta \end{bmatrix} + \begin{bmatrix} n_x \\ n_y \\ n_\theta \end{bmatrix}.$$ 

The dynamics are defined in the pvtol module:

```python
# pvtol = nominal system (no disturbances or noise)
# noisy_pvtol = pvtol w/ process disturbances and sensor noise
from pvtol import pvtol, noisy_pvtol, plot_results

# Find the equilibrium point corresponding to the origin
xe, ue = ct.find_eqpt(
    pvtol, np.zeros(pvtol.nstates),
    np.zeros(pvtol.ninputs), [0, 0, 0, 0, 0, 0],
    iu=range(2, pvtol.ninputs), iy=[0, 1])

x0, u0 = ct.find_eqpt(
    pvtol, np.zeros(pvtol.nstates),
    np.zeros(pvtol.ninputs), np.array([2, 1, 0, 0, 0, 0]),
    iu=range(2, pvtol.ninputs), iy=[0, 1])

# Extract the linearization for use in LQR design
pvtol_lin = pvtol.linearize(xe, ue)

print(pvtol, "\n")
print(noisy_pvtol)
```

We also define the properties of the disturbances, noise, and initial conditions:

```python
# Disturbance and noise intensities
Qv = np.diag([1e-2, 1e-2])
Qw = np.array([[2e-4, 0, 1e-5], [0, 2e-4, 1e-5], [1e-5, 1e-5, 1e-4]])
Qwinv = np.linalg.inv(Qw)

# Initial state covariance
P0 = np.eye(pvtol.nstates)
```
Control system design

We start by defining an extended Kalman filter to estimate the state of the system from the measured outputs.

```python
# Define the disturbance input and measured output matrices
F = np.array([[0, 0], [0, 0], [0, 0], [1, 0], [0, 1], [0, 0]])
C = np.eye(3, 6)

# Estimator update law
def estimator_update(t, x, u, params):
    # Extract the states of the estimator
    xhat = x[0:pvtol.nstates]
    P = x[pvtol.nstates:].reshape(pvtol.nstates, pvtol.nstates)

    # Extract the inputs to the estimator
    y = u[0:3]  # just grab the first three outputs
    u = u[3:5]  # get the inputs that were applied as well

    # Compute the linearization at the current state
    A = pvtol.A(xhat, u)  # A matrix depends on current state
    # A = pvtol.A(xe, ue)  # Fixed A matrix (for testing/comparison)

    # Compute the optimal again
    L = P @ C.T @ Qwinv

    # Update the state estimate
    xhatdot = pvtol.updfcn(t, xhat, u, params) - L @ (C @ xhat - y)

    # Update the covariance
    Pdot = A @ P + P @ A.T - P @ C.T @ Qwinv @ C @ P + F @ Qv @ F.T

    # Return the derivative
    return np.hstack([xhatdot, Pdot.reshape(-1)])

estimator = ct.NonlinearIOSystem(
    estimator_update, lambda t, x, u, params: x[0:pvtol.nstates],
    states=pvtol.nstates + pvtol.nstates**2,
    inputs=noisy_pvtol.state_labels[0:3] \ 
    + noisy_pvtol.input_labels[0:pvtol.ninputs],
    outputs=[f'x{h}[{i}]' for i in range(pvtol.nstates)],
)
print(estimator)
```

Object: sys[3]
Inputs (5): x0, x1, x2, F1, F2,
Outputs (6): xh0, xh1, xh2, xh3, xh4, xh5,
States (42): x[0], x[1], x[2], x[3], x[4], x[5], x[6], x[7], x[8], x[9], x[10], x[11],
            x[12], x[13], x[14], x[15], x[16], x[17], x[18], x[19], x[20], x[21], x[22], x[23],
            x[24], x[25], x[26], x[27], x[28], x[29], x[30], x[31], x[32], x[33], x[34], x[35],
            x[36], x[37], x[38], x[39], x[40], x[41],

We now define an LQR controller, using a physically motivated weighting:

```python
# 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
# due to loss in efficiency.
#
Qx = np.diag([100, 10, (180/np.pi) / 5, 0, 0, 0])
Qu = np.diag([10, 1])
K, _, _ = ct.lqr(A, B, Qx, Qu)

# Control system construction: combine LQR w/ EKF
#
# Use the linearization around the origin to design the optimal gains
# to see how they compare to the final value of P for the EKF
#
# Construct the state feedback controller with estimated state as input
statefbk, _ = ct.create_statefbk_iosystem(pvtol, K, estimator=estimator)
print(statefbk, "\n")

# Reconstruct the control system with the noisy version of the process
# Create a closed loop system around the controller
clsys = ct.interconnect(
    [noisy_pvtol, statefbk, estimator],
    inplist = statefbk.input_labels[0:pvtol.ninputs + pvtol.nstates] + \n    noisy_pvtol.input_labels[pvtol.ninputs:],
    inputs = statefbk.input_labels[0:pvtol.ninputs + pvtol.nstates] + \n    noisy_pvtol.input_labels[pvtol.ninputs:],
    outlist = pvtol.output_labels + statefbk.output_labels + estimator.output_labels,
    outputs = pvtol.output_labels + statefbk.output_labels + estimator.output_labels,
)
print(clsys)

Object: control

Inputs (14): xd[0], xd[1], xd[2], xd[3], xd[4], ud[0], ud[1], xh0, xh1, xh2, xh3,␣
˓→xh4, xh5,
Outputs (2): F1, F2,
States (0):

A = []
B = []
C = []

D = [[-3.16227766e+00 -1.31948924e-07 8.67680175e+00 -2.35855555e+00
      -6.98881806e-08 1.91220852e+00 1.00000000e+00 0.00000000e+00
      3.16227766e+00 1.31948924e-07 -8.67680175e+00 2.35855555e+00
      6.98881806e-08 -1.91220852e+00]]
Simulations

We now simulate the response of the system, starting with an instantiation of the noise:

```python
# Create the time vector for the simulation
Tf = 10
T = np.linspace(0, Tf, 1000)

# Create representative process disturbance and sensor noise vectors
np.random.seed(117) # avoid figures changing from run to run
V = ct.white_noise(T, Qv) # smaller disturbances and noise then design
W = ct.white_noise(T, Qw)
plt.plot(T, V[0], label="V[0]")
plt.plot(T, W[0], label="W[0]")
plt.legend();
```

![Simulations Graph](image_url)
LQR with EKF

[17]: # Put together the input for the system
U = np.vstack([np.outer(xe, np.ones_like(T)), # xd
                np.outer(ue, np.ones_like(T)), # ud
                V, W # disturbances and noise])
X0 = np.hstack([x0, np.zeros(pvtol.nstates), P0.reshape(-1)])

# Initial condition response
resp = ct.input_output_response(clsys, T, U, X0)

# Plot the response
plot_results(T, resp.states, resp.outputs[pvtol.nstates:])

[18]: # Response of the first two states, including internal estimates
    plt.figure()
    h1, = plt.plot(resp.time, resp.outputs[0], 'b-', linewidth=0.75)
    h2, = plt.plot(resp.time, resp.outputs[1], 'r-', linewidth=0.75)

    # Add on the internal estimator states
    xh0 = clsys.find_output('xh0')
    xh1 = clsys.find_output('xh1')
    h3, = plt.plot(resp.time, resp.outputs[xh0], 'k--')
    h4, = plt.plot(resp.time, resp.outputs[xh1], 'k--')

    plt.plot([0, 10], [0, 0], 'k--', linewidth=0.5)

(continues on next page)
Full state feedback

As a comparison, we can investigate the response of the system if the exact state was available:

[19]: # Compute the full state feedback solution
lqr_ctrl, _ = ct.create_statefbk_iosystem(pvtol, K)

lqr_clsys = ct.interconnect(
    [noisy_pvtol, lqr_ctrl],
    inplist = lqr_ctrl.input_labels[0:pvtol.ninputs + pvtol.nstates] + \
    noisy_pvtol.input_labels[pvtol.ninputs:],
    inputs = lqr_ctrl.input_labels[0:pvtol.ninputs + pvtol.nstates] + \
    noisy_pvtol.input_labels[pvtol.ninputs:],
    outlist = pvtol.output_labels + lqr_ctrl.output_labels,
    outputs = pvtol.output_labels + lqr_ctrl.output_labels
)

# Put together the input for the system
U = np.vstack([np.outer(xe, np.ones_like(T)), # xd
               np.outer(ue, np.ones_like(T)), # ud
               V, W * 0 # disturbances and noise
               ])

# Run a simulation with full state feedback
lqr_resp = ct.input_output_response(lqr_clsys, T, U, x0)
10.2.8 Stochastic Response

Richard M. Murray, 6 Feb 2022

This notebook illustrates the implementation of random processes and stochastic response. We focus on a system of the form

\[
\dot{X} = AX + FV \quad X \in \mathbb{R}^n
\]

where \( V \) is a white noise process and the system is a first order linear system.

```
import numpy as np
import scipy as sp
import matplotlib.pyplot as plt
import control as ct
from math import sqrt, exp

# Fix random number seed to avoid spurious figure regeneration
np.random.seed(1)
```

We begin by defining a simple first order system

\[
\frac{dX}{dt} = -aX + V, \quad Y = cX
\]
and a (scalar) white noise signal $V$ with intensity $Q$.

```python
[84]: # First order system
    a = 1
c = 1
    sys = ct.ss([-a], [[1]], [[c]], 0)

    # Create the time vector that we want to use
    Tf = 5
    T = np.linspace(0, Tf, 1000)
dt = T[1] - T[0]

    # Create the basis for a white noise signal
    Q = np.array([[0.1]])
    V = ct.white_noise(T, Q)

    plt.plot(T, V[0])
    plt.xlabel('Time [s]')
    plt.ylabel('V');
```

![White noise signal](image)

Note that the magnitude of the signal seems to be much larger than $Q$. This is because we have a Gaussian process $\Rightarrow$ the signal consists of a sequence of “impulse-like” functions that have magnitude that increases with the time step $dt$ as $1/\sqrt{dt}$ (this gives covariance $E(V(t_1)V^T(t_2)) = Q\delta(t_2 - t_1)$).

```python
[85]: # Calculate the sample properties and make sure they match
    print("mean(V) [0.0] = ", np.mean(V))
    print("cov(V) * dt [0.3g] = " % Q, np.round(np.cov(V), decimals=3) * dt)

    mean(V) [0.0] = 0.17348786109316244
    cov(V) * dt [0.1] = 0.09633133133133133
```

The response of the system to white noise can be computed using the `input_output_response` function:

```python
[86]: # Response of the first order system
    T, Y = ct.input_output_response(sys, T, V)
    plt.plot(T, Y)
    plt.xlabel('Time [s]')
    plt.ylabel('Y');
```
This is a first order system, and so we can compute the analytical correlation function and compare this to the sampled data:

```python
# Compare static properties to what we expect analytically
def r(tau):
    return c**2 * Q / (2 * a) * exp(-a * abs(tau))

print("* mean(Y) [\%0.3g] = %0.3g" % (0, np.mean(Y)))
print("* cov(Y) [\%0.3g] = %0.3g" % (r(0), np.cov(Y)))

* mean(Y) [0] = 0.165
* cov(Y) [0.05] = 0.0151
```

Finally, we look at the correlation function for the input and the output:

```python
# Correlation function for the input
tau, r_V = ct.correlation(T, V)

plt.plot(tau, r_V, 'r-')
plt.xlabel(r'$\tau$')
plt.ylabel(r'$r_V(\tau)$');
```
# Correlation function for the output

```python
tau, r_Y = ct.correlation(T, Y)
plt.plot(tau, r_Y)
plt.xlabel(r'$\tau$')
plt.ylabel(r'$r_Y(\tau)$')

# Compare to the analytical answer
plt.plot(tau, [r(t)[0, 0] for t in tau], 'k--');
```

The analytical curve may or may not line up that well with the correlation function based on the sample. Try running the code again with a different random seed to see how things change based on the specific random sequence chosen at the start.

Note: the right way to compute the correlation function would be to run a lot of different samples of white noise filtered through the system dynamics and compute $R(t_1, t_2)$ across those samples. The correlation function computes the covariance between $Y(t + \tau)$ and $Y(t)$ by varying $t$ over the time range.
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. Please see the Developer’s Wiki for detailed instructions.

Links

- Issue tracker: https://github.com/python-control/python-control/issues
- Mailing list: http://sourceforge.net/p/python-control/mailman/
C
control, 11
control.flatsys, 165
control.matlab, 123
control.optimal, 229
__mul__() (control.flatsys.LinearFlatSystem method), 172
__mul__() (control.flatsys.FlatSystem method), 178
__ne__ () (control.optimal.OptimalControlResult method), 240
__neg__ () (control.FrequencyResponseData method), 113
__neg__ () (control.LinearICSystem method), 178
__neg__ () (control.InterconnectedSystem method), 194
__neg__ () (control.LinearICSystem method), 200
__neg__ () (control.LinearICSystem method), 211
__neg__ () (control.LinearICSystem method), 220
__neg__ () (control.StateSpace method), 90
__neg__ () (control.LinearICSystem method), 100
__neg__ () (control.flatsys.FlatSystem method), 172
__neg__ () (control.flatsys.LinearFlatSystem method), 178
__new__ () (control.optimal.OptimalControlResult method), 240
__radd__() (control.FrequencyResponseData method), 113
__radd__() (control.LinearICSystem method), 108
__radd__() (control.InterconnectedSystem method), 194
__radd__() (control.LinearICSystem method), 200
__radd__() (control.LinearICSystem method), 211
__radd__() (control.LinearICSystem method), 220
__radd__() (control.StateSpace method), 90
__radd__() (control.TransferFunction method), 100
__radd__() (control.flatsys.FlatSystem method), 172
__radd__() (control.flatsys.LinearFlatSystem method), 178
__rdiv__() (control.LinearICSystem method), 200
__rdiv__() (control.LinearICSystem method), 211
__rdiv__() (control.StateSpace method), 90
__rdiv__() (control.LinearICSystem method), 178
__rmul__ () (control.FrequencyResponseData method), 113
__rmul__ () (control.InterconnectedSystem method), 194
__rmul__ () (control.LinearICSystem method), 200
__rmul__ () (control.LinearICSystem method), 212
__rmul__ () (control.NonlinearICSystem method), 220
__rmul__ () (control.StateSpace method), 90
__rmul__ () (control.TransferFunction method), 100
__rmul__ () (control.flatsys.FlatSystem method), 172
__rmul__ () (control.flatsys.LinearFlatSystem method), 178
__rsub__ () (control.FrequencyResponseData method), 113
__rsub__ () (control.InterconnectedSystem method), 194
__rsub__ () (control.LinearICSystem method), 201
__rsub__ () (control.LinearICSystem method), 212
__rsub__ () (control.NonlinearICSystem method), 220
__rsub__ () (control.StateSpace method), 91
__rsub__ () (control.TransferFunction method), 100
__rsub__ () (control.flatsys.FlatSystem method), 172
__rsub__ () (control.flatsys.LinearFlatSystem method), 178
__truediv__ () (control.FrequencyResponseData method), 113
__truediv__ () (control.InterconnectedSystem method), 194
__truediv__ () (control.LinearICSystem method), 201
__truediv__ () (control.LinearICSystem method), 212
__truediv__ () (control.NonlinearICSystem method), 220
__truediv__ () (control.StateSpace method), 91
__truediv__ () (control.TransferFunction method), 100
__truediv__ () (control.flatsys.FlatSystem method), 172
__truediv__ () (control.TransferFunction method), 100

A
A (control.flatsys.LinearFlatSystem attribute), 177
A (control.LinearICSystem attribute), 199
A (control.LinearICSystem attribute), 210
A (control.StateSpace attribute), 89
acker() (in module control), 49
append() (control.flatsys.LinearFlatSystem method), 178
append() (control.LinearICSystem method), 201
append() (control.LinearICSystem method), 212
append() (control.StateSpace method), 91
append() (in module control), 16
append() (in module control.matlab), 136
augw() (in module control), 76

B
B (control.flatsys.LinearFlatSystem attribute), 177
B (control.LinearICSystem attribute), 199
B (control.LinearICSystem attribute), 210
B (control.StateSpace attribute), 89
balred() (in module control), 58
balred() (in module control.matlab), 154
Index

BasisFamily (class in control.flatsys), 169
bdschur() (in module control), 76
BezierFamily (class in control.flatsys), 169
bode() (in module control.matlab), 143
bode_plot() (in module control), 20
create_mpc_iosystem() (control.optimal.OptimalControlProblem method), 237
create_mpc_iosystem() (in module control.optimal), 242
create_statefbk_iosystem() (in module control), 49
ctrb() (in module control), 47
ctrb() (in module control.matlab), 151

create_estimator_iosystem() (in module control), 71
copy() (control.flatsys.LinearFlatSystem method), 172
copy() (control.flatsys.LinearFlatSystem method), 178
copy() (control.FrequencyResponseData method), 114
copy() (control.InputOutputSystem method), 108
copy() (control.InterconnectedSystem method), 195
copy() (control.LinearICSystem method), 201
copy() (control.LinearIOSystem method), 212
copy() (control.NonlinearIOSystem method), 221
copy() (control.optimal.OptimalControlResult method), 240
copy() (control.StateSpace method), 91
copy() (control.TransferFunction method), 100
correlation() (in module control), 70
cost (control.optimal.OptimalControlResult attribute), 238
db2mag() (module control.matlab), 78
dcgain() (control.LinearICSystem method), 178
dcgain() (control.LinearICSystem method), 201
dcgain() (control.LinearIOSystem method), 212
dcgain() (control.StateSpace method), 91
dcgain() (control.TransferFunction method), 101
dcgain() (in module control), 35
dcgain() (in module control.matlab), 137
den (control.TransferFunction attribute), 101
descrcribing_function() (control.DescribingFunctionNonlinearity method), 226
descrcribing_function() (control.friction_backlash_nonlinearity method), 227
descrcribing_function() (control.relay_hysteresis_nonlinearity method), 228
descrcribing_function() (control.saturation_nonlinearity method), 228
descrcribing_function() (in module control), 36
descrcribing_function_plot() (in module control), 22
describingFunctionNonlinearity (class in control), 226
dlqe() (in module control), 72
dlqr() (in module control), 51

damp() (control.flatsys.LinearFlatSystem method), 178
damp() (control.FrequencyResponseData method), 114
damp() (control.LinearICSystem method), 201
damp() (control.LinearIOSystem method), 212
damp() (control.StateSpace method), 91
damp() (control.TransferFunction method), 100
damp() (in module control), 77
damp() (in module control.matlab), 138
dare() (in module control), 44
dare() (in module control.matlab), 160
db2mag() (module control.matlab), 78
db2mag() (module control.matlab), 129
dcgain() (control.flatsys.LinearFlatSystem method), 178
dcgain() (control.FrequencyResponseData method), 114
dcgain() (control.LinearICSystem method), 201
dcgain() (control.LinearIOSystem method), 212
dcgain() (control.StateSpace method), 91
dcgain() (control.TransferFunction method), 101
dcgain() (in module control), 35
dcgain() (in module control.matlab), 137
den (control.TransferFunction attribute), 101
descrcribing_function() (control.DescribingFunctionNonlinearity method), 226
descrcribing_function() (control.friction_backlash_nonlinearity method), 227
descrcribing_function() (control.relay_hysteresis_nonlinearity method), 228
descrcribing_function() (control.saturation_nonlinearity method), 228
descrcribing_function() (in module control), 36
descrcribing_function_plot() (in module control), 22
describingFunctionNonlinearity (class in control), 226
dlqe() (in module control), 72
dlqr() (in module control), 51
Index

E

eqa() (in module control), 60

eqa() (in module control.matlab), 156

eval() (control.flatsys.SystemTrajectory method), 186

eval() (control.FrequencyResponseData method), 114

eval_deriv() (control.flatsys.BezierFamily method), 170

eval_deriv() (control.flatsys.PolyFamily method), 185

evalfr() (in module control), 36

evalfr() (in module control.matlab), 146

F

feedback() (control.flatsys.FlatSystem method), 173

feedback() (control.flatsys.LinearFlatSystem method), 179

feedback() (control.FrequencyResponseData method), 114

feedback() (control.InputOutputSystem method), 108

feedback() (control.InterconnectedSystem method), 195

feedback() (control.LinearICSystem method), 202

feedback() (control.LinearICSystem method), 213

feedback() (control.LinearICSystem method), 221

feedback() (control.StateSpace method), 92

feedback() (control.TransferFunction method), 101

feedback() (in module control), 17

feedback() (in module control.matlab), 134

find_eqpt() (in module control), 62

find_input() (control.flatsys.FlatSystem method), 173

find_input() (control.flatsys.LinearFlatSystem method), 180

find_input() (control.FrequencyResponseData method), 114

find_input() (control.InputOutputSystem method), 109

find_input() (control.InterconnectedSystem method), 196

find_input() (control.LinearICSystem method), 202

find_input() (control.LinearICSystem method), 213

find_output() (control.fsys.FlatSystem method), 173

find_output() (control.fsys.LinearFlatSystem method), 180

find_output() (control.FrequencyResponseData method), 114

find_output() (control.InputOutputSystem method), 109

find_output() (control.InterconnectedSystem method), 196

find_output() (control.LinearICSystem method), 203

find_output() (control.LinearICSystem method), 213

find_output() (control.NonlinearICSystem method), 222

find_output() (control.StateSpace method), 92

find_output() (control.TransferFunction method), 101

find_state() (control.fsys.LinearFlatSystem method), 213

find_state() (control.fsys.LinearFlatSystem method), 222

find_state() (control.StateSpace method), 92

find_state() (control.TransferFunction method), 101

FlatSystem (class in control.fsys), 170

forced_response() (in module control), 27

forward() (control.fsys.FlatSystem method), 173

forward() (control.fsys.LinearFlatSystem method), 180

frd() (in module control), 14

frd() (in module control.matlab), 126

freqresp() (control.fsys.LinearFlatSystem method), 180

freqresp() (control.FrequencyResponseData method), 115

freqresp() (control.LinearICSystem method), 203

freqresp() (control.LinearICSystem method), 213

freqresp() (control.StateSpace method), 92

freqresp() (control.TransferFunction method), 101

freqresp() (in module control), 37

freqresp() (in module control.matlab), 145

frequency_response() (control.fsys.LinearFlatSystem method), 180

frequency_response() (control.FrequencyResponseData method), 115
Index
Python Control Library Documentation, Release dev

set_outputs() (control.LinearICSystem method), 207
set_outputs() (control.Flatsys.FlatSystem method), 175
set_outputs() (control.Flatsys.LinearFlatSystem method), 184
set_outputs() (control.FrequencyResponseData method), 116
set_outputs() (control.InputOutputSystem method), 110
set_outputs() (control.InterconnectedSystem method), 197
set_outputs() (control.LinearICSystem method), 207
set_outputs() (control.LinearICSystem method), 218
set_outputs() (control.LinearICSystem method), 223
set_outputs() (control.StateSpace method), 96
set_outputs() (control.TransferFunction method), 105
set_states() (control.Flatsys.FlatSystem method), 175
set_states() (control.Flatsys.LinearFlatSystem method), 184
set_states() (control.FrequencyResponseData method), 116
set_states() (control.InputOutputSystem method), 110
set_states() (control.InterconnectedSystem method), 198
set_states() (control.LinearICSystem method), 207
set_states() (control.LinearICSystem method), 218
set_states() (control.NonlinearICSystem method), 223
set_states() (control.StateSpace method), 96
set_states() (control.TransferFunction method), 105
setdefault() (control.optimal.OptimalControlResult method), 240
sisotool() (in module control), 42
sisotool() (in module control.matlab), 148
slycot_laub() (control.Flatsys.LinearFlatSystem method), 184
slycot_laub() (control.LinearICSystem method), 208
slycot_laub() (control.LinearICSystem method), 218
slycot_laub() (control.StateSpace method), 97
solve_ocp() (in module control.optimal), 241
squeeze (control.TimeResponseData attribute), 118
ss() (in module control), 11
ss() (in module control.matlab), 125
ss2io() (in module control), 67
ss2tf() (in module control), 81
ss2tf() (in module control.matlab), 130
ssdata() (in module control), 82
stability_margins() (in module control), 39
state_poly_constraint() (in module control.optimal), 244
state_range_constraint() (in module control.optimal), 245
states (control.Flatsys.LinearFlatSystem property), 185
states (control.LinearICSystem property), 208
states (control.LinearICSystem property), 218
states (control.optimal.OptimalControlResult attribute), 238
states (control.StateSpace property), 97
states (control.TimeResponseData property), 120
StateSpace (class in control), 87
step() (in module control.matlab), 140
step_response() (in module control), 32
success (control.optimal.OptimalControlResult attribute), 238
summing_junction() (in module control), 67
SystemTrajectory (class in control.flatsys), 185

T
t (control.TimeResponseData attribute), 117
tf() (in module control), 13
tf() (in module control.matlab), 123
tf2io() (in module control), 68
tf2ss() (in module control), 83
tf2ss() (in module control.matlab), 131
tfdata() (in module control), 84
tfdata() (in module control.matlab), 132
time (control.TimeResponseData property), 120
timebase() (in module control), 84
timebaseEqual() (in module control), 84
TimeResponseData (class in control), 117
TransferFunction (class in control), 97
transpose (control.TimeResponseData attribute), 118

U
u (control.TimeResponseData attribute), 117
unused_signals() (control.InterconnectedSystem method), 198
unused_signals() (control.LinearICSystem method), 208
unwrap() (in module control), 85
unwrap() (in module control.matlab), 162
update() (control.optimal.OptimalControlResult method), 240
use_fbs_defaults() (in module control), 9
use_legacy_defaults() (in module control), 10
use_matlab_defaults() (in module control), 9
use_numpy_matrix() (in module control), 10

V
values() (control.optimal.OptimalControlResult method), 240

W
white_noise() (in module control), 75

X
x (control.TimeResponseData attribute), 117
Y
y (control.TimeResponseData attribute), 117

Z
z (control.TransferFunction attribute), 105
zero() (in module control.matlab), 138
zeros() (control.flatsys.LinearFlatSystem method), 185
zeros() (control.LinearICSystem method), 208
zeros() (control.LinearIOSystem method), 218
zeros() (control.StateSpace method), 97
zeros() (control.TransferFunction method), 106
zeros() (in module control), 41