pyctrl: a Python Suite for Systems and Control

Release 0.4

Mauricio C. de Oliveira

Sep 03, 2019

CONTENTS

CHAPTER

ONE

USER GUIDE

1.1 Introduction

This document describes a suite of Python packages that facilitate the implementation and deployment of dynamic filters and feedback controllers.

The package is entirely written in Python. We only support Python Version 3 and have no intent to support Version 2.

Python is widely deployed and supported, with large amounts of tutorials and documentation available on the web. A good resource to start learning Python is the Beginner's guide to Python.

All code should run on multiple desktop platforms, including Linux and MacOSX. It has also been tested and deployed in the Beaglebobe Black and the Raspberry Pi. In particular we support the Robotics Cape and the Beaglebone Blue. See Section *Installation* for details.

Mauricio de Oliveira

1.1.1 Acknowledgments

The following people contributed significantly to the developement of parts of this package:

1. Gabriel Fernandes
2. Zhu Zhuo

1.2 Installation

The Python source code is available from:

- pyctrl

1.2.1 Robotics Cape support

If you would like to run software on a Beaglebobe Black equipped with the Robotics Cape or the Beaglebone Blue you might need to download and install the Robotics Cape C library from:

- Robotics Cape library

Once you have the library installed and the cape running you should also install our Python bindings available as:

- rcpy package

1.2.2 Raspberry Pi support

TODO

1.3 Tutorial

In this tutorial you will learn how to use Controllers, work with *signals* and the various *blocks* available with the package `pyctrl`. You will also learn how to implement controllers that interact with hardware devices. You can run the code in this tutorial interactively using the python interpreter or by running them as scripts. All code is available in the Section *Examples*.

1.3.1 Hello World!

Start with the following simple *Hello World!* example:

```
# import Python's standard time module
import time

# import Controller and other blocks from modules
from pyctrl import Controller
from pyctrl.block import Printer
from pyctrl.block.clock import TimerClock

# initialize controller
hello = Controller()

# add the signal myclock
hello.add_signal('myclock')

# add a TimerClock as a source
hello.add_source('myclock',
                 TimerClock(period = 1),
                 ['myclock'],
                 enable = True)

# add a Printer as a sink
hello.add_sink('message',
               Printer(message = 'Hello World!'),
               ['myclock'],
               enable = True)

try:
    # run the controller
    with hello:
        # do nothing for 5 seconds
        time.sleep(5)

except KeyboardInterrupt:
    pass

finally:
    print('Done')
```

Depending on the platform you're running this program will print the message *Hello World!* on the screen 4 or 5 times. The complete program is in *hello_world.py*.

1.3.2 What's going on?

Let's analyze each part of the above code to make sense of what is going on. The first couple lines import the modules to be used from the standard Python's `time` and various *`pyctrl`* libraries:

```
import time
from pyctrl import Controller
from pyctrl.block import Printer
from pyctrl.block.clock import TimerClock
```

After importing `Controller` you can initialize the Python variable `hello` as being a `Controller`, more specifically an instance of the class *`pyctrl.Controller`*:

```
hello = Controller()
```

A *`pyctrl.Controller`*, by itself, does nothing useful, so let's add some *signals* and *blocks* that you can interact with. The line:

```
hello.add_signal('myclock')
```

adds the *signal* `myclock`.

A *signal* holds a numeric scalar or vector and is used to communicate between *blocks*. The next lines:

```
hello.add_source('myclock',
                 TimerClock(period = 1),
                 ['myclock'],
                 enable = True)
```

add a `TimerClock` as a *source*. A *source* is a type of block that produces at least one *output* and has *no inputs*.

The mandatory parameters to *`pyctrl.Controller.add_source()`* are a *label*, in this case `myclock`, a *`pyctrl.block`* object, in this case *`pyctrl.block.clock.TimerClock`*, and a *list of signal outputs*, in this case the list containg a single *signal* `['myclock']`. The keyword parameter *enable* is optional and means that the *source* `myclock` will be enabled when the controller *starts* and will be disabled when the controller *stops*.

An instance of the class *`pyctrl.block.clock.TimerClock`* implements a clock based on Python's `threading.Timer` class. It's performance and accuracy can vary depending on the particular implementation for your platform. The parameter `period = 1` passed to `TimerClock` means that the *source* `myclock` will write to the *signal* `myclock` a time stamp every *1* second.

The following line:

```
hello.add_sink('message',
               Printer(message = 'Hello World!'),
               ['myclock'],
               enable = True)
```

adds a *`pyctrl.block.Printer`* as a *sink*. A *sink* is a type of block that takes at least one *input* but produces *no output*.

The parameters to *`pyctrl.Controller.add_sink()`* are a *label*, in this case `'message'`, a *`pyctrl.block`* object, in this case *`pyctrl.block.Printer`*, and a *list of inputs*, in this case `['myclock']`. The keyword parameter *enable* means that the *sink* `message` will be enable when the controller *starts* and will be disabled when the controller *stops*.

An instance of the class *`pyctrl.block.Printer`* implements a *sink* that prints messages and the signals appearing at its input. In this case, the attribute `message = 'Hello World!'` is the message to be printed.

Having created a *source* and a *sink* you are ready to run the controller:

```
with hello:
    # do nothing for 5 seconds
    time.sleep(5)
```

Python's `with` statement automatically *starts* and *stops* the controller. Inside the `with`, the statement `time.sleep(5)` pauses the program for 5 seconds to let the controller run its loop and print *Hello World!* about 5 times. The actual number of times depends on the accuracy of the timer in your platform. Pause for 5.1 seconds instead if you would like to make sure it is printed exactly 5 times.

Secretly behind the statement `with hello` is a call to the pair of methods *pyctrl.Controller.start()* and *pyctrl.Controller.stop()*. In fact, alternatively, one could have written the not so clean:

```
hello.start()
# do nothing for 5 seconds
time.sleep(5)
hello.stop()
```

You should always enclose the controller action inside a Python `try` block as in:

```
try:
    # run the controller
    with hello:
        # do something
        pass

except KeyboardInterrupt:
    pass

finally:
    # do something at the end
    pass
```

This construction allows the controller to be stopped in a predictable way. Under the hood, the controller is run using multiple threads, which have a life of their own and can be tricky to stop. The `except` statement is called in case an *exception* occur. In this particular case if a `KeyboardInterrupt` occurs, for example by a user pressing the `<CTRL-C>` key, the code under the `except` is executing *without the usual accompanying error message*. The `finally` statement should come always after all `except` statements and to makes sure that certain instructions are always executed, even if an exception occur. The Python statement `pass` is used to signify that no instruction is to be executed.

When adding blocks to a controller the keyword argument *enable* is by default set to *False*, which means that blocks would remain enabled even after a controller is stopped. In the case of a `pyctrl.block.TimerClock` object, the clock would continue to run even as the program terminates, most likely locking your terminal, which is not the desired behavior you're after in your first example. Alternatively you could have disabled the clock "manually" by issuing the command:

```
hello.set_source('myclock', enabled = False)
```

for example in the `finally` statement.

The method *pyctrl.Controller.set_source()* allows you to set up attributes of your *source*, in the case the `enabled` attribute that effectively stops the clock. Likewise, *pyctrl.Controller.set_sink()* and *pyctrl.Controller.set_filter()* allow you to set up attributes in *sinks* and *filters*.

1.3.3 The controller loop

In order to understand what is going on on behind the scenes you can probe the contents of the variable `hello`. For example, after running the code in *Hello World!* a call to:

```
print(hello)
```

or simply `hello` if you are using the interactive Python shell, produces the output:

```
<class 'pyctrl.Controller'> with:
  0 timer(s), 4 signal(s),
  2 source(s), 0 filter(s), and 1 sink(s)
```

For more information you can use the method *pyctrl.Controller.info()*. For example:

```
print(hello.info('all'))
```

produces the output:

```
<class 'pyctrl.Controller'> with:
  0 timer(s), 4 signal(s),
  2 source(s), 0 filter(s), and 1 sink(s)
> timers
> signals
  1. clock
  2. duty
  3. is_running
  4. myclock
> sources
  1. clock[Clock, disabled] >> clock
  2. myclock[TimerClock, disabled] >> myclock
> filters
> sinks
  1. myclock >> message[Printer, disabled]
```

which details the *signals*, *sources*, *filters*, *sinks*, and *timers* present in the controller `hello`. Of course the *signals*, *sources* and *sinks* correspond to the ones you have added earlier. Three additional *signals*, `clock`, `duty` and `is_running` and the additional *device* `clock` show up. Those are always present and will be described later.

Note also that the relationship between *sources* and *sinks* with *signals* is indicated by a double arrow >>. In this case, the *source* `myclock` outputs to the *signal* `myclock` and the *sink* `message` has as input the same *signal* `myclock`.

Starting the controller `hello` with the statement `with` or *pyctrl.Controller.start()* fires up the following sequence of events:

1. The state of every *source*, *filter*, *sink*, or *timer* that was installed with the flag *enable* set to *True* is raised to `enabled`.
2. Every *source* is *read* and its outputs are copied to the *signals* connected to the *output* of the *source*. This process is repeated sequentially for every *source* which is in the state `enabled` until all *sources* have run once.
3. For each *filter*, the input signals are *written* to the *filter* that is then *read* and its outputs are copied to the *signals* connected to the *output* of the *filter*. This process is repeated sequentially for every *filter* which is in the state `enabled` until all *filters* have run once.
4. The input signals of every *sink* are *written* to the *sink*. This process is repeated sequentially for every *sink* which is in the state `enabled` until all *sinks* have run once.
5. If the *signal* `is_running` is still *True* go back to step 2, otherwise stop.

6. The state of every *source*, *filter*, *sink*, or *timer* that was installed with the flag *enable* set to *True* is lowered to `disabled`.

The *signal* `is_running` can be set to *False* by calling *pyctrl.Controller.stop()* or exiting the `with` statement. In the *Hello World!* example this is done after doing nothing for 5 seconds inside the `with` statement.

The *flow* of *signals* is established by adding *sources*, *filters*, and *sinks*, which are processed according to the above loop. The content of the input signals is made available to the *filters* and *sinks* as they are processed. For instance, replace the sink `message` by:

```
hello.add_sink('message',
               Printer(message = 'Hello World @ {:3.1f} s'),
               ['myclock'])
```

and run the controller to see a message that now prints the *Hello World* message followed by the value of the *signal* `myclock`. The format *{:3.1f}* is used as in Python's `format()` method. More than one *signal* can be printed by specifying multiple placeholders in the attribute `message` and more input signals.

1.3.4 Devices

As you might suspect after having gone through the *Hello World!* example, it is useful to have a controller with a clock. In fact, as you will learn later in *Timers*, every *pyctrl.Controller* comes equipped with some kind of clock. The method *pyctrl.Controller.add_device()* automates the process of adding blocks to a controller and is typically used when adding blocks that should behave as hardware devices, like a clock. For example, the following code:

```
from pyctrl import Controller

hello = Controller()
hello.add_device('clock',
                 'pyctrl.block.clock', 'TimerClock',
                 outputs = ['clock'],
                 enable = True,
                 kwargs = {'period': 1})
```

automatically creates a *pyctrl.block.clock.TimerClock* which is added to `controller` as the *source* labeled `clock` with *output signal* `clock`. As with regular *sources*, *filters*, and *sinks*, setting the attribute `enable` equal to *True* makes sure that the device is *enabled* at every call to *pyctrl.Controller.start()* and *disabled* at every call to *pyctrl.Controller.stop()*. The dictionary *kwargs* contains parameters that are passed when instantiating the device, in this case a *period* of 1 second.

The main difference between *pyctrl.Controller.add_device()* and, for instance, *pyctrl.Controller.add_source()* is that *pyctrl.Controller.add_device()* takes as arguments strings with the package and class name of the *source*, *filter*, or *sink*, whereas *pyctrl.Controller.add_source()* takes in an instance of an object.

The notion of *device* is much more than a simple convenience though. By having the controller dynamically initialize a block by providing the module and class names as strings to *pyctrl.Controller.add_device()*, i.e. the arguments `'pyctrl.block.clock'` and `'TimerClock'` above, it will be possible to remotely initialize blocks that rely on the presence of specific hardware using our *Client-Server Application Architecture*, as you will learn later. Note that you do not have to directly import any modules when using *pyctrl.Controller.add_device()*.

A controller with a timer based clock is so common that the above construction is provided as a module in *pyctrl.timer*. Using *pyctrl.timer* the *Hello World!* example can be simplified to:

```
# import Python's standard time module
import time
```

```
# import Controller and other blocks from modules
from pyctrl.timer import Controller
from pyctrl.block import Printer

# initialize controller
hello = Controller(period = 1)

# add a Printer as a sink
hello.add_sink('message',
               Printer(message = 'Hello World @ {:3.1f} s'),
               ['clock'],
               enable = True)

try:
    # run the controller
    with hello:
        # do nothing for 5 seconds
        time.sleep(5)

except KeyboardInterrupt:
    pass
```

The complete code is in *hello_timer_1.py*.

A call to `print(hello.info('all'))` produces:

```
<class 'pyctrl.timer.Controller'> with:
  0 timer(s), 3 signal(s),
  1 source(s), 0 filter(s), and 1 sink(s)
> timers
> signals
  1. clock
  2. duty
  3. is_running
> sources
  1. clock[TimerClock, disabled] >> clock
> filters
> sinks
  1. clock >> message[Printer, disabled]
```

which reveals the presence of the signal `clock` and the *device* *pyctrl.block.clock.TimerClock* as a *source*.

In some situations it might be helpful to be able to reset a controller to its original configuration. This can be done using the method *pyctrl.Controller.reset()*. For example, after initialization or after calling:

```
hello.reset()
```

`print(hello.info('all'))` returns:

```
<class 'pyctrl.timer.Controller'> with:
  0 timer(s), 3 signal(s),
  1 source(s), 0 filter(s), and 0 sink(s)
> timers
> signals
  1. clock
  2. duty
  3. is_running
> sources
```

```
  1. clock[TimerClock, disabled] >> clock
> filters
> sinks
```

which shows the presence of the *source* `clock` and the *signal* `clock` but no other *source*, *filter*, *sink*, or *timer*.

1.3.5 Timers

As you have learned so far, all *sources*, *filters*, and *sinks* are continually processed in a loop. In the above example you have equipped the controller `hello` with a *pyctrl.block.clock.TimerClock*, either explicitly, as in *Hello World!*, or implicitly, by loading *pyctrl.timer.Controller*. Note that the controller itself has no notion of time and that events happen periodically simply because of the presence of a *pyctrl.block.clock.TimerClock*, which will stop processing until the set period has elapsed. In fact, the base class *pyctrl.timer.Controller* is also equipped with a clock *source* except that this clock does not attempt to interrupt processing, but simply writes the current time into the *signal* `clock` every time the controller loop is restarted. A controller with such clock runs as fast as possible.

For example, the code:

```
# import Python's standard time module
import time

# import Controller and other blocks from modules
from pyctrl import Controller
from pyctrl.block import Printer

# initialize controller
hello = Controller()

# add a Printer as a sink
hello.add_sink('message',
                Printer(message = 'Current time {:5.3f} s',
                        endln = '\r'),
                ['clock'])

try:

    # run the controller
    with hello:
        # do nothing for 5 seconds
        time.sleep(5)

except KeyboardInterrupt:
    pass
```

will print the current time with 3 decimals as fast as possible on the screen. The additional attribute `endl = '\r'` introduces a carriage return without a line-feed so that the printing happens in a single terminal line. Now suppose that you still want to print the *Hello World!* message every second. You can achieve this using *timers*. Simply add the following snippet before running the controller:

```
# add a Printer as a timer
hello.add_timer('message',
                Printer(message = 'Hello World @ {:3.1f} s '),
                ['clock'], None,
                period = 1, repeat = True)
```

to see the *Hello World* message printed every second as the main loop prints the *Current time* message as fast as possible. The parameters of the method *pyctrl.Controller.add_timer()* are the *label* and *block*, in the case `'message'` and the *pyctrl.block.Printer* object, followed by a *list of signal inputs*, in this case `['clock']`, and a *list of signal outputs*, in this case `None`, then the *timer* period in seconds, and a flag to tell whether the execution of the *block* should repeat periodically, as opposed to just once.

An example of a useful *timer* event to be run only once is the following:

```
from pyctrl.block import Constant

# Add a timer to stop the controller
hello.add_timer('stop',
                Constant(value = 0),
                None, ['is_running'],
                period = 5, repeat = False)
```

which will stop the controller after 5 seconds by setting the *signal* `is_running` to zero. In fact, after adding the above timer one could run the controller loop by simply waiting for the controller to terminate using *pyctrl.Controller.join()* as in:

```
with hello:
    hello.join()
```

Note that your program will not terminate until all *blocks* and *timers* terminate, so it is still important that you always call *pyctrl.Controller.stop()* or use the `with` statement to exit cleanly.

A complete example with all the ideas discussed above can be found in *hello_timer_2.py*.

1.3.6 Filters

So far you have used only *sources*, like *pyctrl.block.clock.TimerClock*, and *sinks*, like *pyctrl.block.Printer*. *Sources* produce outputs and take no input and *sinks* take inputs but produce no output. *Filters* take inputs *and* produce outputs. Your first *filter* will be used to construct a signal which you will later apply to a motor. Consider the following code:

```
# import Controller and other blocks from modules
from pyctrl.timer import Controller
from pyctrl.block import Interp, Constant, Printer

# initialize controller
Ts = 0.1
hello = Controller(period = Ts)

# add motor signals
hello.add_signal('pwm')

# build interpolated input signal
ts = [0, 1, 2,   3,   4,   5,   5, 6]
us = [0, 0, 100, 100, -50, -50, 0, 0]

# add filter to interpolate data
hello.add_filter('input',
                 Interp(xp = us, fp = ts),
                 ['clock'],
                 ['pwm'])

# add logger
```

```
hello.add_sink('printer',
               Printer(message = 'time = {:3.1f} s, motor = {:+6.1f} %',
                       endln = '\r'),
               ['clock','pwm'])

# Add a timer to stop the controller
hello.add_timer('stop',
                Constant(value = 0),
                None, ['is_running'],
                period = 6, repeat = False)

try:

    # run the controller
    with hello:
        hello.join()

except KeyboardInterrupt:
    pass
```

As you learned before, the *sink* `printer` will print the time *signal* `clock` and the value of the *signal* `pwm` on the screen, and the *timer* `stop` will shutdown the controller after 6 seconds. The new block here is the *filter* `input`, which uses the block *`pyctrl.block.Interp`*. This block will take as input the time given by the *signal* `clock` and produce as a result a value that interpolates the values given in the arrays `ts` and `us`. Internally it uses `numpy.interp()` function. See the numpy documentation for details. The reason for the name `pwm` will be explained later in Section *Simulated motor example*. The block *`pyctrl.block.Interp`* will always consider its *x* argument to be relative to the first time the *filter* is written to. That's why `ts` starts at 0. If you need to run it again just reset the block calling:

```
hello.set_filter('input', reset = True)
```

as will be explained in Section *Modifying Blocks*.

The key aspect in this example is how *filters* process *signals*. This can be visualized by calling `print(hello.info('all'))`:

```
<class 'pyctrl.timer.Controller'> with:
  1 timer(s), 4 signal(s),
  1 source(s), 1 filter(s), and 1 sink(s)
> timers
  1. stop[Constant, period = 6, enabled] >> is_running
> signals
  1. clock
  2. duty
  3. is_running
  4. pwm
> sources
  1. clock[TimerClock, disabled] >> clock
> filters
  1. clock >> input[Interp, enabled] >> pwm
> sinks
  1. clock, pwm >> printer[Printer, enabled]
```

where you can see the relationship between the inputs and outputs *signals* indicated by a pair of arrows `>>` coming in *and* out of the the *filter* `input`. The complete code can be found in *hello_filter_1.py*.

Because of the way the controller loop proceeds (see *The controller loop*), you can use the same *signal* as both input and output of a filter. For example, the *filter*:

```
from pyctrl.block.system import Gain
hello.add_filter('gain',
                 Gain(gain = .5),
                 ['pwm'],
                 ['pwm'])
```

scales the *signal* `pwm` by a factor of 0.5 and has the `pwm` as both an input as well as an output *signal*.

1.3.7 Modifying Blocks

Instances of *pyctrl.block.Block* can have its attributes retrieved and modified using the methods *pyctrl.block.Block.get()* and *pyctrl.block.Block.set()*. There is also the special methods *pyctrl.block.Block.reset()*, *pyctrl.block.Block.is_enabled()* and *pyctrl.block.Block.set_enabled()*.

However, once you install an instance of *pyctrl.block.Block* in a controller as a *source*, *filter*, *sink*, or *timer*, you should no longer directly call those methods. Instead you should retrieve and set block attributes using the family of methods *pyctrl.Controller.get_source()*, *pyctrl.Controller.get_filter()*, *pyctrl.Controller.get_sink()*, *pyctrl.Controller.get_timer()*, *pyctrl.Controller.set_source()*, *pyctrl.Controller.set_filter()*, *pyctrl.Controller.set_sink()*, and *pyctrl.Controller.set_timer()*. The reason for this is that, depending on the context, the block instance owned by the controller might be different than the one you originally installed. This is the case, for example, when you run a controller remotely using the *Client-Server Application Architecture*.

For example, for the same controller you implemented in Section *Filters*:

```
hello.get_source('clock')
```

would produce something like:

```
{'average_period': 0,
 'count': 33,
 'enabled': True,
 'period': 0.1,
 'time': 491901.67835816,
 'time_origin': 491898.26005493104}
```

which is a dictionary with all public properties of the source `clock`, an instance of *pyctrl.block.clock.TimerClock*. If only one attribute is sought, as in:

```
hello.get_source('clock', 'count')
```

then *pyctrl.Controller.get_source()* returns simply `33`.

Likewise:

```
hello.set_sink('printer', endln = '\n')
```

changes the attribute `endln` in the *sink* `printer`. More than one atribute can be changed at a time by passing multiple keyword arguments, as in:

```
hello.set_sink('printer', endln = '\n', message = 'New message')
```

There are two special attributes that can be invoked for any block: `reset` and `enabled`. Calling:

```
hello.set_source('clock', reset = True)
```

will internally call *pyctrl.block.Block.reset()* and:

```
hello.set_source('clock', enabled = False)
```

will call *pyctrl.block.Block.set_enabled()*.

Sources, *filters*, *sinks*, and *timers* can also be removed using *pyctrl.Controller.remove_source()*, *pyctrl.Controller.remove_filter()*, *pyctrl.Controller.remove_sink()*, and *pyctrl.Controller.remove_timer()*. For example:

```
hello.remove_sink('printer')
```

removes the *sink* `printer` from the controller loop.

Finally, one can also read and write to blocks using *pyctrl.Controller.read_source()*, *pyctrl.Controller.read_filter()*, *pyctrl.Controller.write_filter()*, and *pyctrl.Controller.write_sink()*. These will be used in the next section to read values from a block that logs running data.

1.3.8 Working with data

So far you have been running blocks and displaying the results on your screen using *pyctrl.block.Printer*. If you would want to store the generated data for further processing you should instead use the block *pyctrl.block.Logger*. Let us revisit the example from Section *Filters*, this time adding also a *pyctrl.block.Logger*. The only difference is the introduction of the additional *sink*:

```
from pyctrl.block import Logger

# add logger
hello.add_sink('logger',
               Logger(),
               ['clock','pwm'])
```

A complete example can be found in *hello_filter_2.py*. Once the controller has run, you can then retrieve all generated data by reading from the *sink* `logger` using the method *pyctrl.block.Logger.get()* to retrieve the property *log* as in:

```
# retrieve data from logger
data = hello.get_sink('logger', 'log')
```

retrieves the data stored in `logger` and copy it to the dictionary `data`. Data is stored by row, with one key per signals used as inputs to the *pyctrl.block.Logger*. One can conveniently access the data by using the signal label:

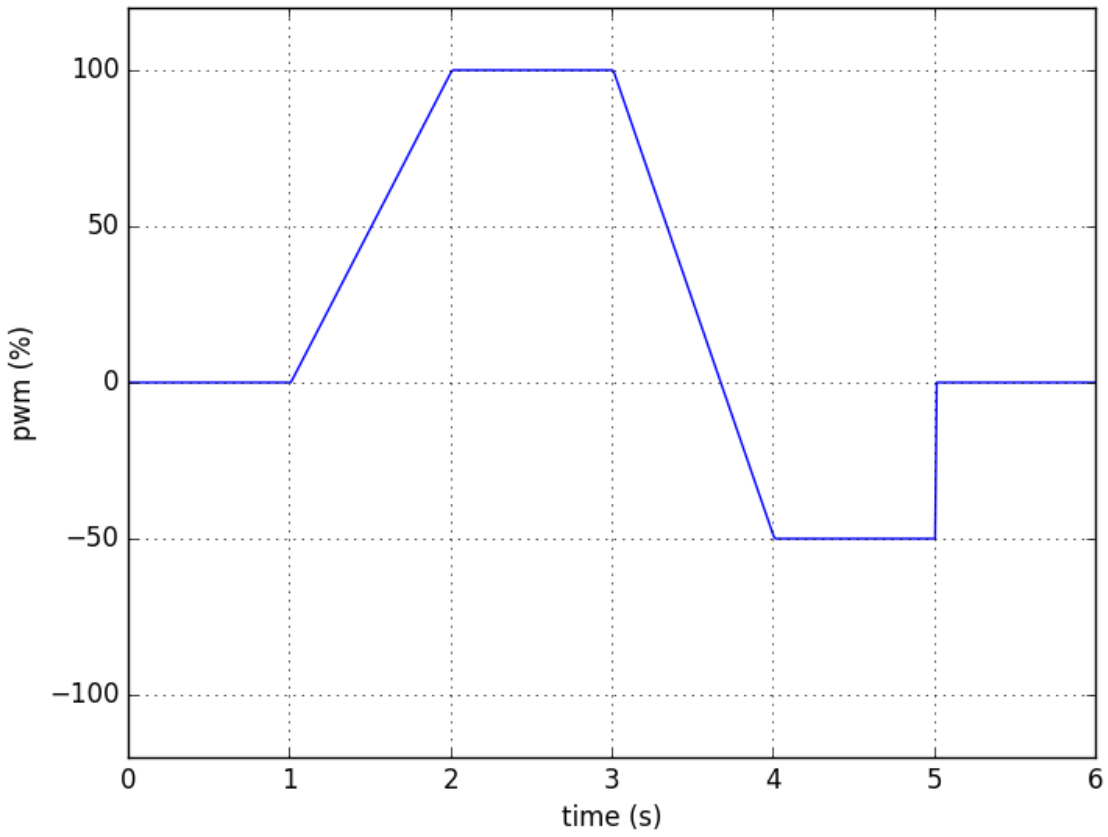
```
clock = data['clock']
pwm = data['pwm']
```

Since this is Python, you can now do whatever you please with the data. For example you can use matplotlib to plot the data:

```
# import matplotlib
import matplotlib.pyplot as plt

# start plot
plt.figure()
```

```
# plot input
plt.plot(clock, pwm, 'b')
plt.ylabel('pwm (%)')
plt.xlabel('time (s)')
plt.ylim((-120,120))
plt.xlim(0,6)
plt.grid()

# show plots
plt.show()
```

After running the controller `hello`, the above snippet should produce a plot like the one below:



from which you can visualize the input signal `pwm` constructed by the *`pyctrl.block.Interp`* block. Note that for better granularity the sampling period used in *hello_filter_2.py* is 0.01 s, whereas the one used in *hello_filter_1.py* was only 0.1 s.

1.3.9 Simulated motor example

You will now work on a more sophisticated example, in which you will combine various filters to produce a simulated model of a DC-motor. The complete code is in *simulated_motor_1.py*.

1.3.9.1 A transfer-function model

The beginnig of the code is similar to *hello_filter_2.py*:

```
# import Controller and other blocks from modules
from pyctrl.timer import Controller
from pyctrl.block import Interp, Logger, Constant
from pyctrl.system.tf import DTTF, LPF

# initialize controller
Ts = 0.01
simotor = Controller(period = Ts)

# build interpolated input signal
ts = [0, 1, 2,   3,   4,   5,   5, 6]
us = [0, 0, 100, 100, -50, -50, 0, 0]

# add motor signal
simotor.add_signal('pwm')

# add filter to interpolate data
simotor.add_filter('input',
                   Interp(xp = us, fp = ts),
                   ['clock'],
                   ['pwm'])
```

Note that you will be simulating this motor with a sampling period of 0.01 seconds, that is, a sampling frequency of 100 Hz. The model you will use for the DC-motor is based on the diffential equation model:

$$\tau\ddot{\theta} + \dot{\theta} = gu$$

where u is the motor input voltage, θ is the motor angular displacement, and g and τ are constants related to the motor physical parameters. The constant g is the *gain* of the motor, which relates the steady-state velocity achieved by the motor in response to a constant input voltage, and the constant τ is the time constant of the motor, which is a measure of how fast the motor respond to changes in its inputs. **If you have no idea of what's going on here, keep calm and read on! You do not need to understand all the details to be able to use this model.**

Without getting into details, in order to simulate this differential equation you will first convert the above model in the following discrete-time difference equation:

$$\theta_k - (1+c)\theta_{k-1} + c\theta_{k-2} = \frac{gT_s(1-c)}{2}\left(u_{k-1} + u_{k-2}\right), \quad c = e^{-\frac{T_s}{\tau}}$$

where T_s is the sampling period. It is this equation that you will simulate by creating the following *filter*:

```
# import math and numpy
import math, numpy

from pyctrl.block.system import System
from pyctrl.system.tf import DTTF

# Motor model parameters
tau = 1/55    # time constant (s)
g = 0.092     # gain (cycles/sec duty)
c = math.exp(-Ts/tau)
d = (g*Ts)*(1-c)/2

# add motor signals
simotor.add_signal('encoder')
```

```
# add motor filter
simotor.add_filter('motor',
                   System(model = DTTF(
                       numpy.array((0, d, d)),
                       numpy.array((1, -(1 + c), c)))),
                   ['pwm'],
                   ['encoder'])
```

The input signal to the *filter* `motor` is the *signal* `pwm`, which is the signal that receives the interpolated input data you created earlier. The ouput of the *filter* `motor` is the *signal* `encoder`, which corresponds to the motor angular position θ.

The *block* used in the *filter* `motor` is of the class *`pyctrl.block.system.System`*, which allows one to incorporate a variety of system models into filters. See *Module pyctrl.system* for other types of system models available. The particular model you are using is a `pyctrl.system.DTTF`, in which DTTF stands for *Discrete-Time Transfer-Function*. This model corresponds to the difference equation discussed above. Note dependency on Python's math library and numpy.

To wrap it up you will add a *sink* *`pyctrl.block.Logger`* to collect the data generated during the simulation and a *timer* to stop the controller:

```
# add logger
simotor.add_sink('logger',
                 Logger(),
                 ['clock','pwm','encoder'])

# Add a timer to stop the controller
simotor.add_timer('stop',
                  Constant(value = 0),
                  None, ['is_running'],
                  period = 6, repeat = False)
```

As usual, the simulation is run with:

```
# run the controller
with simotor:
    simotor.join()
```

1.3.9.2 Collecting and plotting the results

After running the simulation you can read the data collected by the logger:

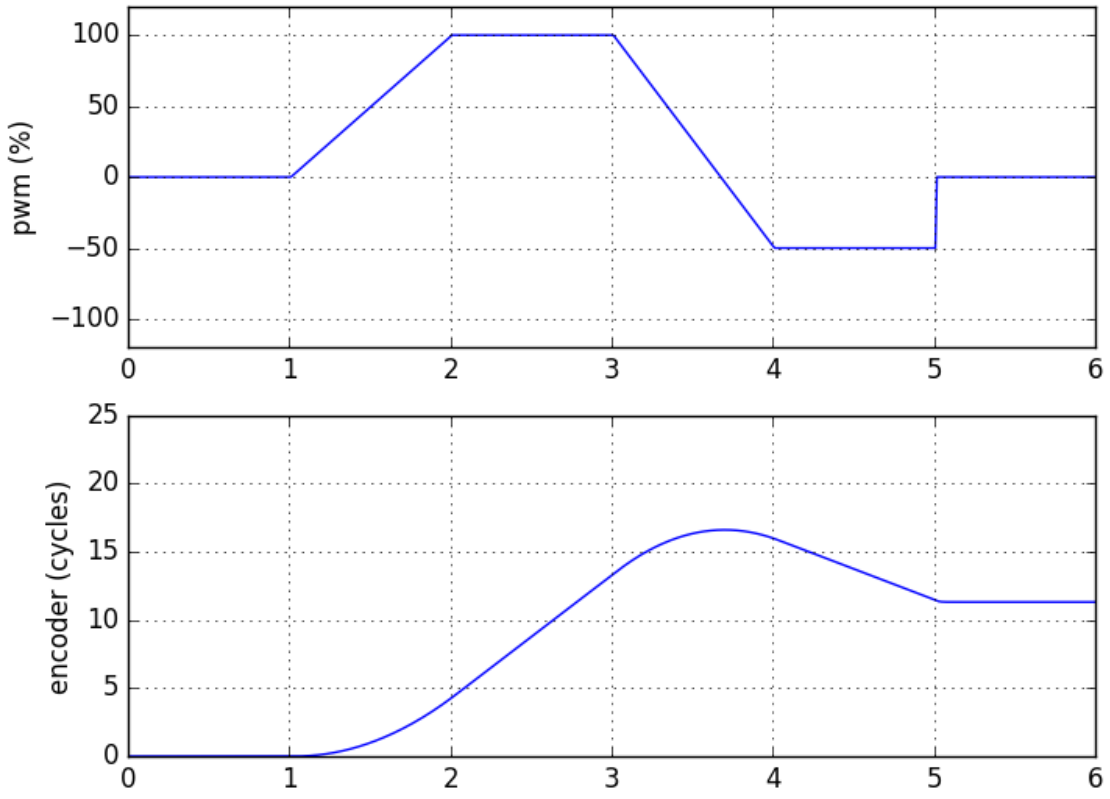
```
# read logger
data = simotor.get_sink('logger', 'log')
clock = data['clock']
pwm = data['pwm']
encoder = data['encoder']
```

and plot the results using matplotlib:

```
# import matplotlib
import matplotlib.pyplot as plt

# start plot
plt.figure()
```

```
# plot input
plt.subplot(2,1,1)
plt.plot(clock, pwm, 'b')
plt.ylabel('pwm (%)')
plt.ylim((-120,120))
plt.grid()

# plot position
plt.subplot(2,1,2)
plt.plot(clock, encoder,'b')
plt.ylabel('position (cycles)')
plt.ylim((0,25))
plt.grid()

# show plots
plt.show()
```

to obtain a plot similar to the one below:



where you can visualize both the motor input signal `pwm` and the motor output signal `encoder`, which predicts that the motor will stop at about 13 cycles (revolutions) from its original position if the input signal `pwm` were applied at its input.

1.3.9.3 Calculating velocity and low-pass filtering

The above setup is one that corresponds to a typical microcontroller interface to a DC-motor, in which the motor voltage is controlled through a PWM (Pulse-Width-Modulation) signal ranging from 0-100% of the pulse duty-cycle (with negative values indicating a reversal in voltage polarity), and the motor position is read using an encoder. In this situation, one might need to calculate the motor *velocity* from the measured position. You will do that now by adding a couple more filters to the simulated motor model. The complete code can be found in *simulated_motor_2.py*.

After introducing *filters* to produce the *signals* `pwm` and `encoder`, you will add another filter to calculate the speed by *differentiating* the `encoder` *signal*:

```
from pyctrl.block.system import Differentiator

# add motor speed signal
simotor.add_signal('speed')

# add motor speed filter
simotor.add_filter('speed',
                   Differentiator(),
                   ['clock','encoder'],
                   ['speed'])
```

The *filter* `speed` uses a block *pyctrl.block.system.Differentiator* that takes as input both the `clock` signal and the *signal* `encoder`, which is the one being differentiated, and produces the output *signal* `speed`.

Differentiating a *signal* is always a risky proposition, and should be avoided whenever possible. Even in this simulated environment, small variations in the clock period and in the underlying floating-point calculations will give rise to noise in the *signal* `speed`. In some cases one can get around by filtering the *signal*. For example, by introducing a *low-pass filter* as in:

```
from pyctrl.system.tf import LPF

# add low-pass signal
simotor.add_signal('fspeed')

# add low-pass filter
simotor.add_filter('LPF',
                   System(model = LPF(fc = 5, period = Ts)),
                   ['speed'],
                   ['fspeed'])
```

The *filter* `LPF` uses a block *pyctrl.block.system.System* that takes as input the `speeed` signal and produces the output *signal* `fspeed`, which is the filtered version of the input `speeed`. The model used in *pyctrl.block.system.System* is the low-pass filter *pyctrl.system.tf.LPF* with cutoff frequency `fc` equal to 5 Hz.

Finally collect all the data in the logger:

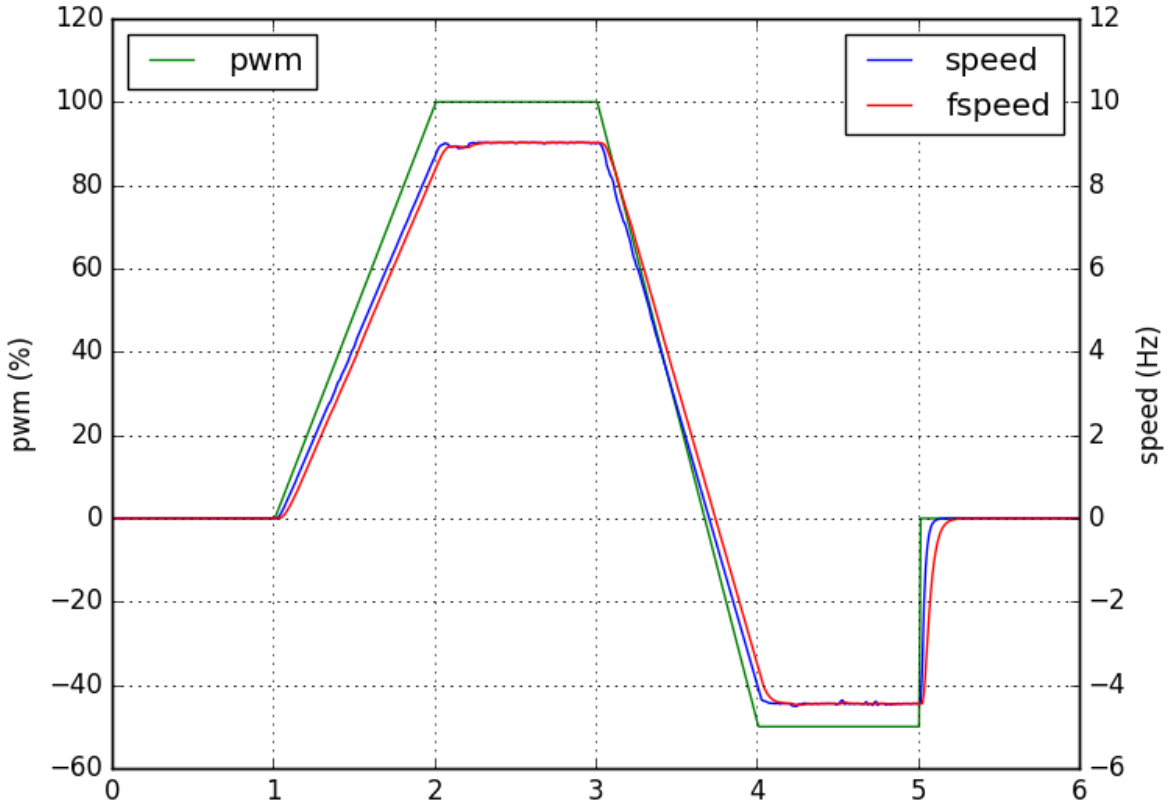
```
# add logger
simotor.add_sink('logger',
                 Logger(),
                 ['clock','pwm','encoder','speed','fspeed'])
```

After all that you should have a controller with the following blocks:

```
<class 'pyctrl.timer.Controller'> with:
  1 timer(s), 7 signal(s),
  1 source(s), 4 filter(s), and 1 sink(s)
> timers
```

```
  1. stop[Constant, period = 6, enabled] >> is_running
> signals
  1. clock
  2. duty
  3. encoder
  4. fspeed
  5. is_running
  6. pwm
  7. speed
> sources
  1. clock[TimerClock, disabled] >> clock
> filters
  1. clock >> input[Interp, enabled] >> pwm
  2. pwm >> motor[System, enabled] >> encoder
  3. clock, encoder >> speed[Differentiator, enabled] >> speed
  4. speed >> LPF[System, enabled] >> fspeed
> sinks
  1. clock, pwm, encoder, speed, fspeed >> logger[Logger, enabled]
```

Running *simulated_motor_2.py* produces a plot similar to the one shown below:



where you can simultaneously visualize the *signal* `pwm`, the *signal* `speed` as calculated by the differentiator, and the filtered speed *signal* `fspeed`.

Note how the order of the *filters* is important. Output that is needed as input for other filters must be computed first if they are to be applied *in the same iteration* of the controller loop. Otherwise, their update values will only be applied

on the next iteration. That would be the case, for example, if you had inverted the order of the *filters* `motor` and `speed` as in:

```
> filters
  1. clock >> input[Interp, enabled] >> pwm
  2. clock, encoder >> speed[Differentiator, enabled] >> speed
  3. pwm >> motor[System, enabled] >> encoder
  4. speed >> LPF[System, enabled] >> fspeed
```

which would make the *filter* `speed` always see the input *signal* `encoder` as calculated in the previous loop iteration. Note how this would also affect the input to the *filter* `LPF`!

1.3.10 Interfacing with hardware

In this section you will learn how to interface with real hardware. Of course you can only run the examples in this section if you have the appropriate hardware equipment.

1.3.10.1 Before you begin

For demonstration purposes it will be assumed that you have an Educational MIP (Mobile Inverted Pendulum) kit with a Beaglebone Black equipped with a Robotics Cape or a Beaglebone Blue. You may have to download additional libraries and the rcpy package. See Section *Installation* for details.

Make sure that all required software is installed and working before proceeding. Consult the documentation provided in the links above and the Section *Installation* for more details.

1.3.10.2 Installing devices

Before you can interact with hardware you have to install the appropriate devices. The following code will initialize a controller that can interface with the Robotics Cape:

```
# import Controller and other blocks from modules
from pyctrl.rc import Controller

# initialize controller
Ts = 0.01
bbb = Controller(period = Ts)
```

Note that the code is virtually the same as used before except that you are importing *Controller* from `pyctrl.rc` rather than from *`pyctrl`* or *`pyctrl.timer`*. This controller automatically adds a clock based on the MPU9250 IMU. You can check its presence by typing:

```
print(bbb.info('sources'))
```

which produces the output:

```
> sources
  1. clock[MPU9250, enabled] >> clock
```

It is now time to install the devices you will be using. For this demonstration you will use one of the MIP's motor and the corresponding encoder. First load the encoder:

```
# add encoder as source
bbb.add_device('encoder1',
               'pyctrl.rc.encoder', 'Encoder',
               outputs = ['encoder'],
               kwargs = {'encoder': 3,
                         'ratio': 60 * 35.557})
```

which will appear as a *source* labeled `encoder1` connected to the output *signal* `encoder`.

You install devices using the same method *pyctrl.Controller.add_device()* you already worked with before. Besides the mandatory parameters *label*, *device_module*, and *device_class*, you should pass the corresponding list of *inputs* and *outputs* signals as well as any initialization parameters in the dictionary *kwargs*.

The parameters in *kwargs* are specific to the device and are passed to the *device_module* and *device_class* constructor. Each device has its own specific set of parameters. In the above example, the attribute `encoder` is set to 3, which selects the 3rd (out of a total of 4 available) hardware encoder counter in the Beaglebone Black, and `ratio` is set to 60 * 35.557 to reflect the presence of a gear box connected between the encoder and the wheel shaft, which is the movement that you would like the encoder to measure. Using the above ratio, the unit of the *signal* `encoder` will be *cycles*, that is, one complete turn of the wheel will add or substract one to the *signal* `encoder`.

You load the motor as:

```
# add motor as sink
bbb.add_device('motor1',
               'pyctrl.rc.motor', 'Motor',
               inputs = ['pwm'],
               kwargs = {'motor': 3},
               enable = True)
```

which will appear as the *sink* `motor1` connected to the input *signal* `pwm`. Note that the above code makes use of the optional parameter `enable`, which controls whether the device should be enabled at *pyctrl.Controller.start()* and disabled at *pyctrl.Controller.stop()*. In the case of motors or other devices that can present danger if left in some unknown state, this is done for safety: terminating or aborting your code will automatically turn off the physical motor. Note that the *source* `encoder1` will remain enabled all the time, since there is no danger in keeping counting your encoder pulses even when the controller is off.

As with the encoder, the motor constructor takes the additional parameter `motor` provided in the dictionary *kwargs*. In this case you have selected the 3rd (out of a total of 4 available) hardware motor drivers (H-bridges) in the Robotics Cape or Beaglebone Blue. Those are driven by the Beaglebone Black or Blue PWM hardware generators, which in this case is controlled by the input signal `pwm` taking values between -100 and 100. Negative values reverse the polarity of the voltage applied to the motor causing a reversal in the motor direction. Note that the value of the actual voltage applied to the motor will depend on the voltage source connected to the Robotics Cape. In the case of the Educational MIP kit this voltage will be approximately 7.4 V when the battery is fully charged.

The current configuration of the controller after installing the devices is shown in the output of `print(bbb.info('all'))`:

```
<class 'pyctrl.rc.Controller'> with:
  0 timer(s), 5 signal(s),
  2 source(s), 0 filter(s), and 1 sink(s)
> timers
> signals
  1. clock
  2. duty
  3. encoder
  4. is_running
  5. pwm
> sources
```

```
  1. clock[MPU9250, enabled] >> clock
  2. encoder1[Encoder, enabled] >> encoder
> filters
> sinks
  1. pwm >> motor1[Motor, disabled]
```

1.3.10.3 Using hardware devices

Once hardware devices are installed as *sinks*, *filters*, or *sources*, you can use them exactly as before. *Sensors* will usually be installed as *sources* and *actuators* typically as *sinks*.

Because you use the same names for the signals handled by the encoder and motor devices as the ones used in the Section *Simulated motor example*, you can simply copy parts of that code to repeat the motor experiment, this time using real hardware. For example, the code:

```
from pyctrl.block import Interp, Logger, Constant
from pyctrl.block.system import System, Differentiator
from pyctrl.system.tf import LPF

# build interpolated input signal
ts = [0, 1, 2,   3,   4,   5,   5, 6]
us = [0, 0, 100, 100, -50, -50, 0, 0]

# add filter to interpolate data
bbb.add_filter('input',
               Interp(xp = us, fp = ts),
               ['clock'],
               ['pwm'])

# add motor speed signal
bbb.add_signal('speed')

# add motor speed filter
bbb.add_filter('speed',
               Differentiator(),
               ['clock','encoder'],
               ['speed'])

# add low-pass signal
bbb.add_signal('fspeed')

# add low-pass filter
bbb.add_filter('LPF',
               System(model = LPF(fc = 5, period = Ts)),
               ['speed'],
               ['fspeed'])

# add logger
bbb.add_sink('logger',
             Logger(),
             ['clock','pwm','encoder','speed','fspeed'])

# Add a timer to stop the controller
bbb.add_timer('stop',
              Constant(value = 0),
              None, ['is_running'],
              period = 6, repeat = False)
```

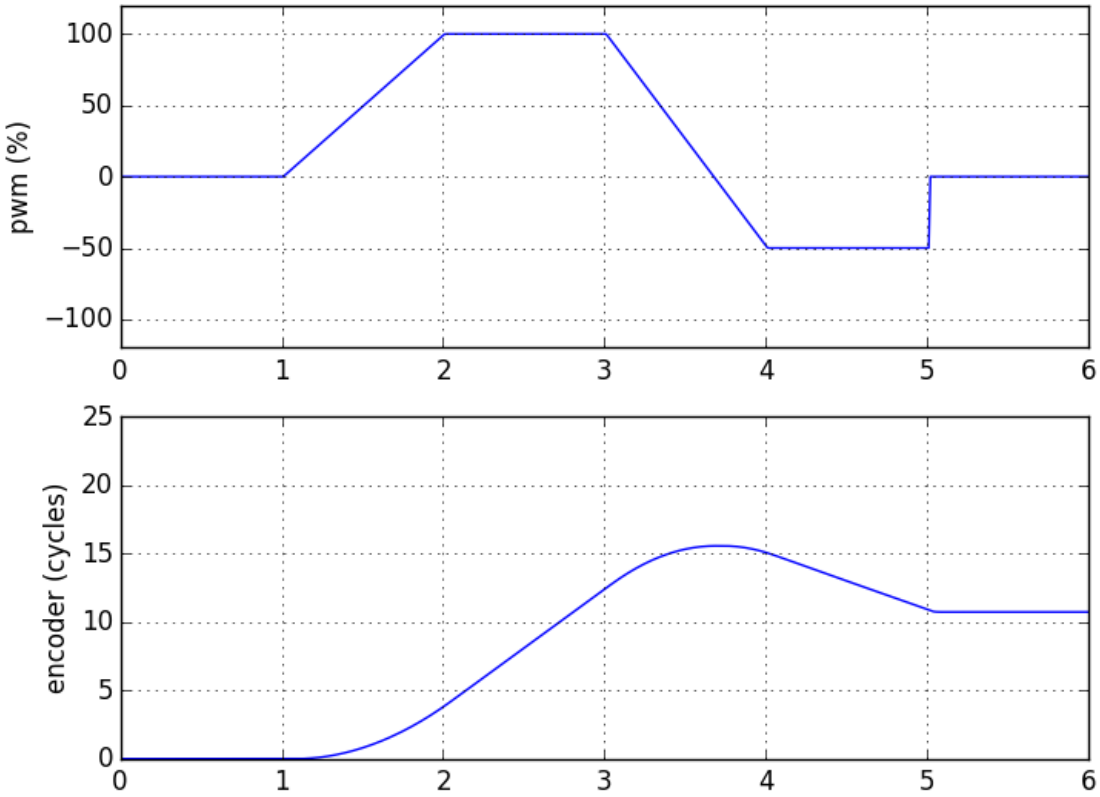
will produce a controller with the following connections:

```
<class 'pyctrl.rc.Controller'> with:
  1 timer(s), 7 signal(s),
  2 source(s), 3 filter(s), and 2 sink(s)
> timers
  1. stop[Constant, period = 6, enabled] >> is_running
> signals
  1. clock
  2. duty
  3. encoder
  4. fspeed
  5. is_running
  6. pwm
  7. speed
> sources
  1. clock[MPU9250, enabled] >> clock
  2. encoder1[Encoder, enabled] >> encoder
> filters
  1. clock >> input[Interp, enabled] >> pwm
  2. clock, encoder >> speed[Differentiator, enabled] >> speed
  3. speed >> LPF[System, enabled] >> fspeed
> sinks
  1. pwm >> motor1[Motor, disabled]
  2. clock, pwm, encoder, speed, fspeed >> logger[Logger, enabled]
```

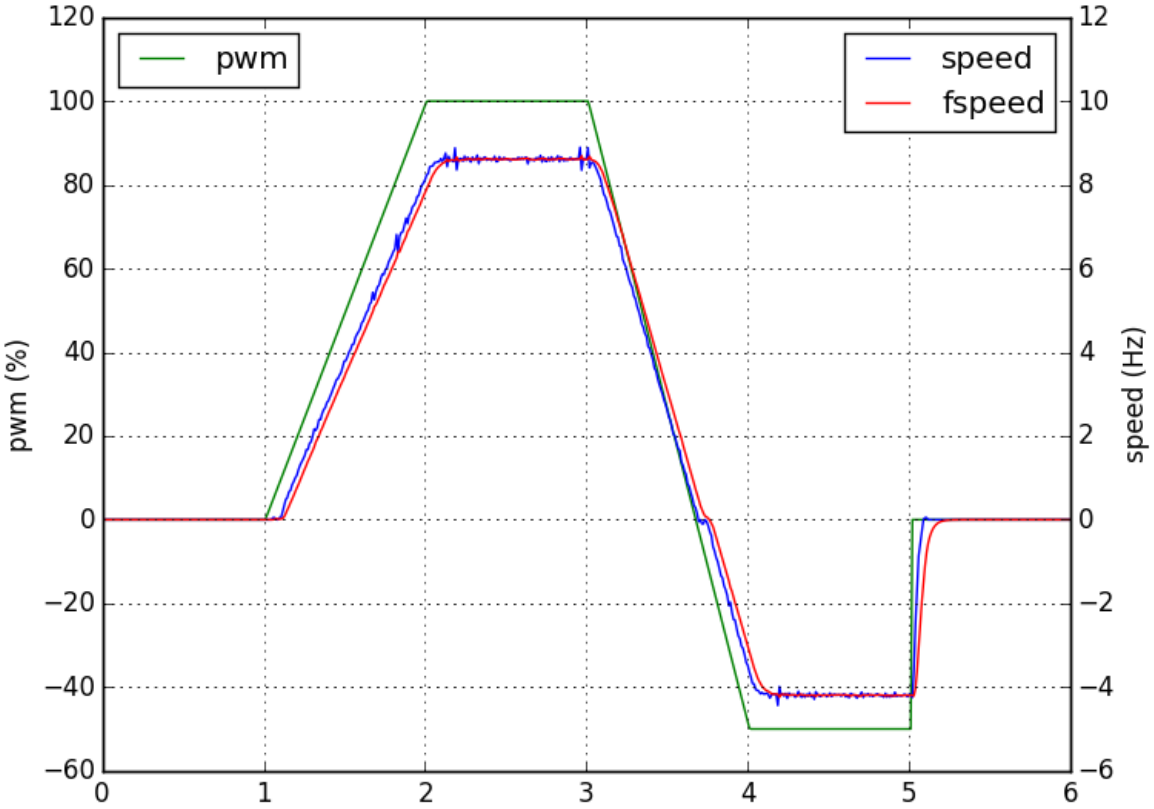
You run this controller program invoking:

```
# run the controller
with bbb:
    bbb.join()
```

Upon running the complete code provided in *rc_motor.py* the following plots are produced using matplotlib:



To the naked eye, the position plot above is virtually identical to the one obtained using the simulated model from Section *Simulated motor example*. Some subtle differences are visible in the velocity plot below:



where you can see that the motor has some difficulties overcoming stiction, that is the static friction force that dominates when the velocities become small: it takes a bit longer to start around 1 s and it gets *stuck* again around 3.7 s when the velocity becomes zero. Note also the more pronounced noise which is amplified by the differentiator and then attenuated by the low-pass filter.

You might want to take the additional step:

```
# reset the clock
bbb.set_source('clock', reset = True)
```

of resetting the clock before starting the controller if you want your clock to start at 0.

1.3.11 Closed-loop control

The initial motivation to write this package was to be able to easily implement and deploy feedback controllers. The subject of feedback control is extensive and will not be covered in any detail here. A completely unbiased and awesome reference is *[deO16]*. The treatment is suitable to undergraduates students with an engineering or science background.

Do not let yourself be intimidated by the language here, you do not need to understand all the details to implement or, better yet, to benefit from using a feedback controller!

1.3.11.1 Proportional-Integral motor speed control

You will now turn to the implementation of a closed-loop Proportional-Integral controller, or PI controller for short, on the same hardware used in the Section *Interfacing with hardware*. Start by installing the same devices as before, one motor and one encoder:

```
# import Controller and other blocks from modules
from pyctrl.rc import Controller

# initialize controller
Ts = 0.01
bbb = Controller(period = Ts)

# add encoder as source
bbb.add_device('encoder1',
               'pyctrl.rc.encoder', 'Encoder',
               outputs = ['encoder'],
               kwargs = {'encoder': 3,
                         'ratio': 60 * 35.557})

# add motor as sink
bbb.add_device('motor1',
               'pyctrl.rc.motor', 'Motor',
               inputs = ['pwm'],
               kwargs = {'motor': 3},
               enable = True)
```

Because you will be controlling the motor speed, add also a differentiator:

```
from pyctrl.block.system import Differentiator

# add motor speed signal
bbb.add_signal('speed')

# add motor speed filter
bbb.add_filter('speed',
               Differentiator(),
               ['clock','encoder'],
               ['speed'])
```

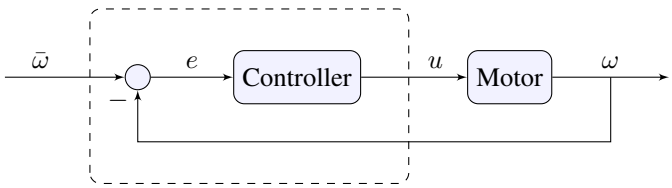
According to the dynamic model introduced earlier in Section *Simulated motor example*, the transfer-function from the PWM input, u, to the motor velocity, $\omega = \dot{\theta}$, is:

$$G(s) = \frac{\Omega(s)}{U(s)} = \frac{g}{\tau s + 1}$$

You will implement a PI (Proportional-Integral) controller with transfer-function:

$$K(s) = \frac{U(s)}{E(s)} = K_\mathrm{p} + \frac{K_\mathrm{i}}{s} = K_\mathrm{p}\frac{s + K_\mathrm{i}/K_\mathrm{p}}{s}$$

The way by which you will connect this controller to the motor is given in the feedback block-diagram:



Feedback here means that a measurement of the motor speed, ω, will be compared with a reference speed, $\bar{\omega}$, to create an *error signal*, e, that will then be *fed back* to the *Motor* by the *Controller*. When ω matches $\bar{\omega}$ exactly then the error signal, e, is zero. It is the controller's job to produce a suitable PWM input, u, so that this is possible. The PI controller does that by *integrating* the error signal. Indeed, the transfer-function of the PI controller corresponds to:

$$u(t) = K_\mathrm{p} e(t) + K_\mathrm{i} \int_0^t e(\tau)\, d\tau$$

In a way, the integrator *estimates* the necessary level of motor PWM input, u, so that the error can be made small, in other words, so that the motor can track a desired reference speed, $\bar{\omega}$. Indeed, if the controller succeeds in its task to keep the error signal small, that is $e = 0$, then the contribution from the proportional term, $K_\mathrm{p} e(t)$, will also be zero.

There's lot to be said about how to *design* suitable gains K_p and K_i *[deO16]*. Here you will choose

$$\frac{K_\mathrm{i}}{K_\mathrm{p}} = \tau^{-1}, \quad K_\mathrm{p} = g^{-1}$$

so that the closed-loop transfer-function from $\bar{\omega}$ to ω becomes

$$\frac{\Omega(s)}{\bar{\Omega}(s)} = \frac{G(s)K(s)}{1 + G(s)K(s)} = \frac{1}{\tau K_\mathrm{p}^{-1} g^{-1} s + 1} = \frac{1}{\tau s + 1}$$

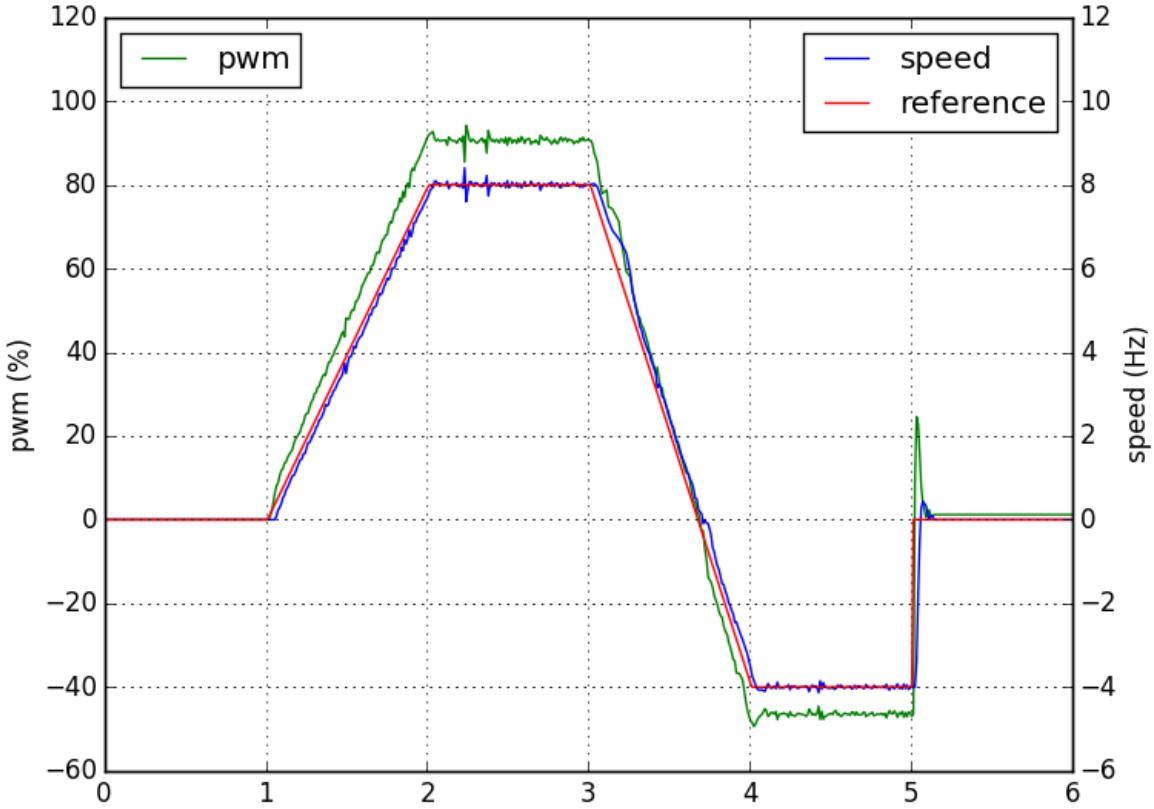
This will make the motor respond with the same time-constant as if it were in open-loop but this time with the ability to *track* a constant reference velocity signal $\bar{\omega}$.

Taking advantage of the blocks *pyctrl.block.system.System* and *pyctrl.block.system.Feedback*, and of the PID control algoritm provided in *pyctrl.system.tf.PID* you can calculate and implement this PI controller in only a few lines of code:

```
from pyctrl.block.system import Feedback, System
from pyctrl.system.tf import PID

# calculate PI controller gains
tau = 1/55    # time constant (s)
g = 0.092     # gain (cycles/sec duty)

Kp = 1/g
Ki = Kp/tau

# build controller block
pid = System(model = PID(Kp = Kp, Ki = Ki, period = Ts))

# add motor speed signal
bbb.add_signal('speed_reference')

# add controller to the loop
bbb.add_filter('PIcontrol',
               Feedback(block = pid),
               ['speed','speed_reference'],
               ['pwm'])
```

The block *pyctrl.block.system.Feedback* implements the operations inside the dashed box in the *feedback diagram*. That is, it calculates the error signal, e, and evaluates the block given as the attribute `block`, in this case the *pyctrl.block.system.System* containing as the attribute `model` the controller *pyctrl.system.tf.PID*.

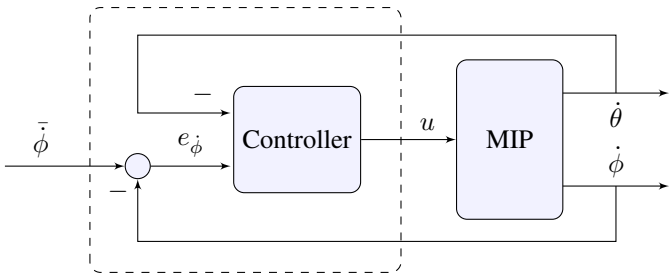
The complete code, including a reference speed that looks like the PWM input used before to drive the motor in Sections *Simulated motor example* and *Interfacing with hardware*, is in the example *rc_motor_control.py*. Results obtained with the MIP kit should look like the following plot:



Note how the motor speed *tracks* the reference signal in closed-loop, effectively calculating the required PWM input necessary for acomplishing that. Compare this behaviour with the previous *open-loop* graphs in which a curve similar to the reference speed was instead applied directly to the motor PWM input. Look also for some interesting side-effects of feedback control, such as the somewhat smoother behavior near the points where the motor reaches zero speed. Look for *[deO16]* for much more in depth discussions.

1.3.11.2 State-space MIP balance controller

Your second feedback controller will be more sophisticated. You will use two measurements to balance the MIP kit in its upright position. More details on the modeling and design of the controller you will implement here can be found in *[Zhuo16]*. The final controller corresponds to the following feedback diagram:



in which you can see that the feedback controller makes use of two measurements, the vertical angle velocity, $\dot{\theta}$, and the wheel angular velocity, $\dot{\phi}$. It also takes in a reference wheel angular velicity, $\bar{\dot{\phi}}$, that can be used to drive the MIP

backward and forward.

As described in detail in *[Zhuo16]*, the discrete-time controller, corresponding to the block inside the dashed box in the *feedback diagram*, is given by a discrete-time state-space model of the form:

$$x_{k+1} = Ax_k + By_k$$
$$u_k = Cx_k + Dy_k$$

where y_k represents the controller input, consisting of the measurement and error signals

$$y_k = \begin{pmatrix} \dot{\theta}_k \\ \dot{\phi}_k \\ \dot{\bar{\phi}}_k \end{pmatrix}$$

and u_k is the PWM input to be applied to both left and right motors.

Implementing this controller is very simple. First initialize the controller as:

```
# import blocks and controller
from pyctrl.rc.mip import Controller
from pyctrl.block.system import System, Subtract, Differentiator, Sum, Gain
from pyctrl.block.nl import ControlledCombination
from pyctrl.block import Logger, ShortCircuit
from pyctrl.system.ss import DTSS

# create mip
mip = Controller()
```

Note that you have imported the special `pyctrl.rc.mip.Controller` class that already initializes all devices needed for controlling the MIP. A look at `mip.info('all')`:

```
<class 'pyctrl.rc.mip.Controller'> with:
  0 timer(s), 9 signal(s),
  4 source(s), 0 filter(s), and 2 sink(s)
> timers
> signals
  1. clock
  2. duty
  3. encoder1
  4. encoder2
  5. is_running
  6. pwm1
  7. pwm2
  8. theta
  9. theta_dot
> sources
  1. clock[MPU9250, enabled] >> clock
  2. inclinometer[Inclinometer, enabled] >> theta, theta_dot
  3. encoder1[Encoder, enabled] >> encoder1
  4. encoder2[Encoder, enabled] >> encoder2
> filters
> sinks
  1. pwm1 >> motor1[Motor, disabled]
  2. pwm2 >> motor2[Motor, disabled]
```

reveals that `pyctrl.rc.mip.Controller` already installed the following devices:

1. **a clock**;

2. **one inclinometer**, which is based on a built in giroscope and will be used to measure $\dot{\theta}$; the inclinometer also produces a measurement of θ that is only accurate under small velocities and accelerations;
3. **two motors**, which give access to the two PWM signals driving the left and right motors of the MIP;
4. **two encoders**, which measure the relative angular displacement between the body of MIP and the axis of the left and right motors, from which you will measure $\dot{\phi}$.

The angular velocity $\dot{\phi}$ can be obtained after averaging the two wheel encoders and differentiating the resulting angle ϕ:

```
# phi is the average of the encoders
mip.add_signal('phi')
mip.add_filter('phi',
               Sum(gain=0.5),
               ['encoder1','encoder2'],
               ['phi'])

# phi dot
mip.add_signal('phi_dot')
mip.add_filter('phi_dot',
               Differentiator(),
               ['clock','phi'],
               ['phi_dot'])
```

Also add the reference signal $\bar{\dot{\phi}}$:

```
# phi dot reference
mip.add_signal('phi_dot_reference')
```

Having all signals necessary for feedback, construct and implemented the feedack controller as follows:

```
import numpy as np

# state-space matrices
A = np.array([[0.913134, 0.0363383],[-0.0692862, 0.994003]])
B = np.array([[0.00284353, -0.000539063], [0.00162443, -0.00128745]])
C = np.array([[-383.009, 303.07]])
D = np.array([[-1.22015, 0]])

B = 2*np.pi*(100/7.4)*np.hstack((-B, B[:,1:]))
D = 2*np.pi*(100/7.4)*np.hstack((-D, D[:,1:]))

ssctrl = DTSS(A,B,C,D)

mip.add_signal('pwm')
mip.add_filter('controller',
               System(model = ssctrl),
               ['theta_dot','phi_dot','phi_dot_reference'],
               ['pwm'])
```

As a final step connect the *signal* `pwm` to both motors using a *`pyctrl.block.ShortCircuit`*:

```
# connect to motors
mip.add_filter('cl1',
               ShortCircuit(),
               ['pwm'],
               ['pwm1'])
mip.add_filter('cl2',
```

```
                ShortCircuit(),
                ['pwm'],
                ['pwm2'])
```

The code for a complete controller with some added bells and whitles to let you drive the MIP while balancing upright is given in *rc_mip_balance.py*. A video of the resulting balancing controller is available here.

1.4 More advanced usage

The next sections describe tasks that are better suited to advanced users, such as working with the provided Client-Server architecture, extending Controllers, or writing your own Blocks. Make sure you have gone through the *Tutorial* and have a good understanding of the concepts discussed there before reading this chapter.

1.4.1 Qualified names and containers

Instances of the class *pyctrl.block.container.Container* can hold and execute *signals*, *sources*, *filters*, *sinks*, and *timers*, just like an instance of *pyctrl.Controller*. Indeed *pyctrl.Controller* inherits most of its functionality from *pyctrl.block.container.Container*. An instance of *pyctrl.block.container.Container* works as a *filter*, which can be installed using *pyctrl.Controller.add_filter* or *pyctrl.Controller.add_device* or as a timer using *pyctrl.Controller.add_timer*.

As the name suggests, a *pyctrl.block.container.Container* can contain other blocks, just like a *pyctrl.Controller* does. In order to access elements inside a container one uses a *qualified name* involving the special character forward slash (*/*). For example, consider the following code:

```
# import Controller and other blocks from modules
from pyctrl.timer import Controller
from pyctrl.block.system import Gain
from pyctrl.block.container import Container, Input, Output

# initialize controller
controller = Controller(period = 1)

controller.add_signals('s1','s2','s3')

controller.add_filter('gain',
                      Gain(gain = 2),
                      ['s1'], ['s2'])

# add container
controller.add_filter('container',
                      Container(),
                      ['s1'], ['s3'])

# add elements inside the container
controller.add_signals('container/s1', 'container/s2')

controller.add_source('container/input',
                      Input(),
                      ['s1'])

controller.add_filter('container/gain',
                      Gain(gain = 3),
                      ['s1'],['s2'])
```

```
controller.add_sink('container/output1',
                    Output(),
                    ['s2'])
```

The command:

```
controller.add_filter('container',
                      Container(),
                      ['s1'], ['s3'])
```

adds a *filter* called `container` with input `s1` and output `s3`. Once an instance of *pyctrl.block.container.Container* has been added to a controller, its elements can be accessed by using a qualified name which is preceded by the name of the container separated by a forward slash (/). For example, the code:

```
controller.add_signals('container/s1', 'container/s2')
```

adds two signals, `s1` and `s2`, to the container `container`. Note that names inside containers are local, `s1` and `container/s1` refer to different signals!

Likewise, the command:

```
controller.add_filter('container/gain',
                      Gain(gain = 3),
                      ['s1'],['s2'])
```

adds a *filter* called `gain` to the container `container`. Note that the *inputs* and *outputs* above refer to signals which are *local* to `container`, that is the signals `container/s1` and `container/s2`. In fact, the parameters *inputs* and *outputs* in *pyctrl.Controller.add_filter* as well as *pyctrl.Controller.add_source*, *pyctrl.Controller.add_sink*, and *pyctrl.Controller.add_device*, must all be local symbols. In order to connect the inputs and output of the container with signals of the controller we use two special blocks: *pyctrl.block.container.Input* and *pyctrl.block.container.Output*. For example:

```
controller.add_source('container/input',
                      Input(),
                      ['s1'])
```

connects the single input of the container, the signal `s1` to the local container signal `container/s1`, and:

```
controller.add_sink('container/output1',
                    Output(),
                    ['s2'])
```

connects the local container signal `container/s2` to the single output of the container, the signal `s3`.

The above controller corresponds to the following configuration:

```
<class 'pyctrl.timer.Controller'> with:
  0 timer(s), 6 signal(s),
  1 source(s), 2 filter(s), and 0 sink(s)
> timers
> signals
  1. clock
  2. duty
  3. is_running
  4. s1
  5. s2
  6. s3
```

```
> sources
  1. clock[TimerClock, disabled] >> clock
> filters
  1. s1 >> gain[Gain, enabled] >> s2
  2. s1 >> container[Container, disabled] >> s3
     <class 'pyctrl.block.container.Container'> with:
       0 timer(s), 2 signal(s),
       1 source(s), 1 filter(s), and 1 sink(s)
     > timers
     > signals
       1. s1
       2. s2
     > sources
       1. input[Input, enabled] >> s1
     > filters
       1. s1 >> gain[Gain, enabled] >> s2
     > sinks
       1. s2 >> output1[Output, enabled]
> sinks
```

Note how the contents of the container are shown indented. Executing:

```
import time
controller.set_signal('s1', 1)
with controller:
    time.sleep(1.1)

print('s1 = {}'.format(controller.get_signal('s2')))
print('s2 = {}'.format(controller.get_signal('s3')))
```

will produce:

```
s1 = 2
s2 = 3
```

For a practical example of containers used to synchronize activities on a timer see *rc_mip_balance.py*.

1.4.2 Multiplexing and demultiplexing

Blocks that are instances of `pyctrl.block.BufferBlock` support *multiplexing of inputs* and *demultiplexing of outputs*.

Multiplexing means that all the inputs of a `pyctrl.block.BufferBlock` are collected into a single numpy 1D-array before the block is evaluated.

Demultiplexing means that the outputs of a `pyctrl.block.BufferBlock` are split into multiple outputs after the block is evaluated.

For example, the blocks `pyctrl.block.system.System` and `pyctrl.block.system.TimeVaryingSystem` always multiplexes their input. This means that instances of `pyctrl.system.System` can seamlessly handle systems with multiple inputs.

The attributes `mux` and `demux` can be also used to modify the behavior of existing blocks. For this reason, you will rarely need a special block for multiplexing and demultiplexing. If you do, just use `pyctrl.block.BufferBlock`. For example, a mux-type block can be created by setting `demux = True` in a `pyctrl.block.BufferBlock` as in:

```
from pyctrl.block import BufferBlock
controller.add_filter('mux',
                      BufferBlock(mux = True),
                      ['input1','input2'],
                      ['muxout'])
```

Likewise, you could modify an existing block, such as *pyctrl.block.system.Gain* to demultiplex its outputs as in:

```
from pyctrl.block.system import Gain
controller.add_filter('gain',
                      Gain(demux = True),
                      ['muxout'],
                      ['output1','output2'])
```

Because blocks can arbitrarily manipulate signals, it is not possible to detect inconsistencies in the sizes of inputs and output until execution time. Even then some blocks might simply ignore discrepancies without generating any errors! For example, a block like:

```
controller.add_filter('gain',
                      Gain(),
                      ['input1'],
                      ['output1','output2'])
```

is not only valid but also does not generate any runtime error. However, only the output `output1` contains a multiple of `output2`. Since `output2` does not match any input it is simply ignored. Likewise, in:

```
controller.add_filter('gain',
                      Gain(),
                      ['input1','input2'],
                      ['output1'])
```

only the input `intput1` gets passed on to the output `output1`. Again, no runtime errors are ever generated.

Finally, the block:

```
controller.add_filter('gain',
                      Gain(gain = numpy.array([-1,2]), demux = True),
                      ['input1'],
                      ['output1','output2'])
```

leverages demultiplexing and the use of a numpy array as a gain to produce a signal `output1` which is `input1` multiplied by *-1* and a signal `output2` which is `input1` multiplied by *2*.

1.4.3 Client-Server Application Architecture

Since the beginnings of the development of this package one goal was to be able to deploy and run controllers on embedded systems. With that goal in mind we provide two special classes of controllers: `pyctrl.server.Controller` and *pyctrl.client.Controller*, and two scripts: `pyctrl_start_server` and `pyctrl_stop_server` to start and stop a controller server. Those scripts and classes can be combined to run applications remotely.

1.4.3.1 Starting the server

Start by using the script `pyctrl_start_server` to create a server for you. In this tutorial, you will create a server on the same machine you will be running the client. The process of initializing a server on a remote machine is virtually identical. Type:

```
pyctrl_start_server
```

which start the server and produces the following output:

```
pyctrl_start_server (version 1.0)
> Options:
   Hostname[port]: localhost[9999]
  Sampling period: ---
    Verbose level: 1

Type 'pyctrl_start_server -h' for more options

<class 'pyctrl.Controller'> with:
  0 timer(s), 3 signal(s),
  1 source(s), 0 filter(s), and 0 sink(s)
> timers
> signals
  1. clock
  2. duty
  3. is_running
> sources
  1. clock[Clock, enabled] >> clock
> filters
> sinks

> Starting server... done
> Hit Ctrl-C or use 'pyctrl_stop_server' to exit the server
```

showing that a server has been started at the `localhost` at the port `9999`. Those are the default values for *host* and *port*. It also shows that the server is running a controller which is an instance of the basic *`pyctrl.Controller`* class.

The attribute `host` is qualified name or valid IP address of the machine you're connecting to and `port` is the port you would like to connect. The connection is established using a TCP network socket. See *Options available with pyctrl_start_server* for how to set these options.

1.4.3.2 Connecting your client

Start a new console and a new python shell. Proceed as in Section *Hello World!* and create a controller:

```
from pyctrl.client import Controller
hello = Controller()
```

The only difference is that you imported `Controller` from the class *`pyctrl.client.Controller`*, as opposed to from *`pyctrl.Controller`*. Once you have initialized a controller as a *client* and you have a controller running as a server,the flow is very much like before. For example, we can query the controller using `print(hello.info('all'))`, which in this case should reproduce the exact same configuration of the controller running on the server:

```
pyctrl_start_server (version 1.0)
> Options:
```

```
   Hostname[port]: localhost[9999]
  Sampling period: ---
    Verbose level: 1

<class 'pyctrl.Controller'> with:
  0 timer(s), 3 signal(s),
  1 source(s), 0 filter(s), and 0 sink(s)
> timers
> signals
  1. clock
  2. duty
  3. is_running
> sources
  1. clock[Clock, enabled] >> clock
> filters
> sinks
```

Programming a client controller is, for the most part, just like programming a local controller. However, there are some important caveats you should be aware of. For instance, attempting to install an instance *pyctrl.block.clock.TimerClock* as a *source* by typing:

```
from pyctrl.block.clock import TimerClock
hello.add_source('myclock',
                 TimerClock(period = 1),
                 ['myclock'])
```

will fail. The reason for the failure is because an instance of the class *pyctrl.block.clock.TimerClock* cannot be safely *transferred* from your hardware to another, that is from the *client* to the *server*. Here is where the notion of a *device* comes in handy. Instead of instantiating *pyctrl.block.clock.TimerClock* on the client, you can use *pyctrl.Controller.add_device()* to have the remote controller instantiate it directly on the server hardware! As in Section *Devices* type:

```
hello.add_device('myclock',
                 'pyctrl.block.clock', 'TimerClock',
                 outputs = ['myclock'],
                 kwargs = {'period': 1},
                 enable = True)
```

to add a *device* *pyctrl.block.clock.TimerClock* by letting the remote server instantiate the object.

Alternatively, you can provide the optional parameters *module* and *kwargs*:

```
# initialize controller
hello = Controller(host = 'localhost', port = 9999,
                   module = 'pyctrl.timer',
                   kwargs = {'period': 1})
```

which will install the controller of class *pyctrl.timer.Controller*, which already contains a *pyctrl.block.clock.TimerClock* clock, directly on the server.

From this point on, just proceed as in *Hello World!* to add a *pyctrl.block.Printer*:

```
from pyctrl.block import Printer
hello.add_sink('message',
               Printer(message = 'Hello World!'),
               ['myclock'],
               enable = True)
```

and run the controller:

```
import time
with hello:
    # do nothing for 5 seconds
    time.sleep(5)
```

If you can't see anything happening for five seconds, look again. This time not on the console running the *client*, but on the console running the *server*. You should see the message *Hello World!* printed there a couple of times. What you have accomplished is running a task on the remote server controller by programming it on the client controller. Effectively, and appart from some subtleties concerning devices, the only difference was importing from *pyctrl.client.Controller* rather than from *pyctrl.Controller*.

1.4.3.3 What's under the hood?

Before moving forward, a bit of a technical note. You might be wondering why *pyctrl.block.clock.TimerClock* could not be added as a *source* but it is fine to add *pyctrl.block.Printer* as a *sink*. The difference has to do with the ability of the client controller to *transfer* a block to the remote controller. In order for that to process to happen, the block *pyctrl.block.Printer* has to be safely deconstructed, packed, transmitted over the network socket, unpacked and then reconstructed at the server controller.

As mentioned earlier, the transmission part is done using a TCP network socket. The packing and unpacking bit is done using a technique called serialization. We rely on Python's pickle module to handle the dirtiest parts of this job. In a nutshell, if an object cannot be serialized by *pickle*, that is it cannot be *pickled*, then it cannot be installed remotely as a *source*, *filter*, *sink*, or *timer*. In this case, it needs to be installed as a *device*.

If you are curious why *pyctrl.block.clock.TimerClock* could not be serialized, it is because *pyctrl.block.clock.TimerClock* runs on a separate process thread, and there is no way to simply transfer the thread information over the network. As for *pyctrl.block.Printer*, it is possible to use its attributes to reconstruct it on the server side.

Note that what decides if an object can be *pickled* or not is not its base class but the contents of its attributes *at the time one attempts to pickle it*. For example, a *pyctrl.block.Printer* in which you have setup the attribute `file` to redirect its output to a local file will fail to install as a *sink*. You could instead install it as a *device*, but in this case, the output would be redirected to a file that lives in the remote server rather than the local client.

A final note about serialization and *pickle* is that this process is inherently unsafe from a security standpoint. Code that is embedded in a serialized object can be used to take control of or damage the server host by running malicious code. If security is a concern, it must be addressed at the network level, before a client is allowed to connect to a server, for example by setting up a firewall that restricts connection to a know number of potential client addresses combined with some strong form of authentication.

1.4.3.4 Options available with `pyctrl_start_server`

Starting `pyctrl_start_server` with the `-h` flag displays the options available:

```
usage: pyctrl_start_server [-h] [-m MODULE] [-c CONTROLLER] [-H HOST] [-p PORT]
                           [-v VERBOSE] [-t PERIOD]

pyctrl_start_server (version 1.0)

optional arguments:
  -h, --help            show this help message and exit (default: False)
  -m MODULE, --module MODULE
                        controller module (default: pyctrl)
  -c CONTROLLER, --controller CONTROLLER
```

```
                        controller class (default: Controller)
  -H HOST, --host HOST  host name or IP address (default: localhost)
  -p PORT, --port PORT  port number (default: 9999)
  -v VERBOSE, --verbose VERBOSE
                        level of verbosity (default: 1)
  -t PERIOD, --period PERIOD
                        sampling period in seconds (default: 0.01)
```

Besides getting help one can initialize a server with any arbitrary controller using the `-m`, `--module` and `-c`, `--controller` options as in:

```
pyctrl_start_server -m pyctrl.timer
```

which initializes the server controller to be an instance of *pyctrl.timer.Controller* instead of the default *pyctrl.Controller*.

Another useful pair of options is `-H`, `--host` and `-p`, `--port`, which can be used to change the current host name or IP address and port. For example:

```
pyctrl_start_server -m pyctrl.rc.mip -H 192.168.0.132 -p 9090
```

would initialize the server using an instance of `pyctrl.rc.mip.Controller` at the local network IP address *192.168.0.132* at the port *9090*.

Finally `-t`, `--period` lets one set the controller sampling period and `-v`, `--verbose` control how much messages you would see coming out of the server controller. Setting verbose to a number higher than *2* produces an enormous amount of information that could be useful for debugging.

Out of all these options, `-v`, `--verbose`, `-H`, `--host` and `-p`, `--port`, are the ones that cannot be changed by a client connected to the controller server after it's been initialized.

1.4.3.5 Working with `pyctrl.client.Controller`

As shown above, working with an instance of *pyctrl.client.Controller* is for the most part identical to working with any other instance of *pyctrl.Controller*. In this section you will learn a couple of useful practices when using a client controller.

A first issue is setting the client to talk to the server at the right address and port. This can be done by initializing the client with attributes `host` and `port`. For example:

```
from pyctrl.client import Controller
client = Controller(host = '192.168.0.132', port = 9090)
```

would connect the client to a local network server at address *192.168.0.132* and port *9090*.

Once connected, an usual mistake is to make assumptions about the current state of a server controller. Since another client could have connected to the server earlier and changed settings in unpredictable ways, it might be useful to call `pyctrl.client.Controller.reset()` to *reset* the remote controller at the server before doing anything:

```
client.reset()
```

`pyctrl.client.Controller.reset()` can also be used to install a completely new controller on the server, as if using the `-m`, `--module` and `-c`, `--controller` options in `pyctrl_start_server`. For example:

```
client.reset(module = 'pyctrl.timer')
```

install a new instance of *pyctrl.timer.Controller* in the remote server. You can query the server about its controller class by using *pyctrl.Controller.info()* as in:

```
client.info('class')
```

which should then return the string "<class 'pyctrl.timer.Controller'>". You can also pass arguments to the controller constructor. For example:

```
client.reset(module = 'pyctrl.timer', kwargs = {'period': 0.1})
```

will install a new instance of *pyctrl.timer.Controller* running at 10 Hz on the remote server.

For convenience, all these operations can be performed by the *pyctrl.client.Controller* constructor. For example:

```
from pyctrl.client import Controller
client = Controller(host = '192.168.0.132', port = 9090,
                    module = 'pyctrl.timer',
                    kwargs = {'period': 0.1})
```

initializes the client and resets the remote controller by installing a new instance of *pyctrl.timer.Controller* running at 10 Hz on the remote server.

1.4.3.6 SSH and port forwarding

A common setup is that of a server running on an embedded system, such as a Beaglebone Black or a Raspberry Pi, controlled remotely by a computer. In most cases, connections to the server will be established using ssh.

The following is a typical session: the user on the client computer, *user@client*, establishes a connection to the remote server, in this case a Beaglebone Black as *root@192.168.0.68*, using ssh:

```
user@client:~$ ssh -L9999:localhost:9999 root@192.168.0.68
Debian GNU/Linux 8

BeagleBoard.org Debian Image 2016-11-06

Support/FAQ: http://elinux.org/Beagleboard:BeagleBoneBlack_Debian

default username:password is [debian:temppwd]

Last login: Sat Apr  1 17:06:08 2017 from 192.168.0.1
root@beaglebone:~#
```

The important detail here is the argument `-L9999:localhost:9999`, which tells ssh to establish a *tunnel*, that is to forward the port 9999 from the server to the client.

Because of that, the user can initiate the server using `localhost` as its host name:

```
root@beaglebone:~# pyctrl_start_server
pyctrl_start_server (version 1.0)

Type 'pyctrl_start_server -h' for more options

> Options:
   Hostname[port]: localhost[9999]
  Sampling period: ---
    Verbose level: 1
```

```
<class 'pyctrl.Controller'> with:
  0 timer(s), 3 signal(s),
  1 source(s), 0 filter(s), and 0 sink(s)
> timers
> signals
  1. clock
  2. duty
  3. is_running
> sources
  1. clock[Clock, enabled] >> clock
> filters
> sinks

> Starting server... done
> Hit Ctrl-C or use 'pyctrl_stop_server' to exit the server
```

After starting the server, on another terminal, the user runs his application as a client, also connected to `localhost`. For example, using the interactive shell:

```
user@client:~$ python
Python 3.4.5 |Anaconda 2.3.0 (x86_64)| (default, Jul  2 2016, 17:47:57)
[GCC 4.2.1 Compatible Apple LLVM 4.2 (clang-425.0.28)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> from pyctrl.client import Controller
>>> hello = Controller()
>>> print(hello.info('all'))
<class 'pyctrl.Controller'> with:
  0 timer(s), 3 signal(s),
  1 source(s), 0 filter(s), and 0 sink(s)
> timers
> signals
  1. clock
  2. duty
  3. is_running
> sources
  1. clock[Clock, enabled] >> clock
> filters
> sinks

>>> quit()
user@client:~$
```

In this way, all network traffic is managed by ssh. An additional advantage is that this traffic is also encrypted and potentially compressed, which adds efficiency and security to the entire session. With ssh you can also forward your X graphics terminal using the `-X` flag. For example:

```
ssh -X -L9999:localhost:9999 root@192.168.0.68
```

would forward both the 9999 port and your X terminal session.

1.4.4 Performance considerations

1.4.4.1 Error Handling

Since you're using Python, error testing and handling can be kept at a minimum. Thanks to duck typing and other Python features, anything that could go wrong will be dealt with at execution time, with errors being propagated using

the standard Python exception mechanism. As discussed in Section *What's going on?*, users can handle errors by enclosing segments of code using the `try` statement:

```
try:
    # do something
    ...

except:
    # do something if exception was raised
    ...

finally:
    # always do this
    ...
```

Most of the time, error checking is limited to operations that could invalidate your controller. In particular, very few error checking tests are performed in the methods *pyctrl.block.Block.read()* and *pyctrl.block.Block.write()*. Such tests would be repeated in the main controller loop and could potentially impact performance. If you need to perform tests in those methods consider using assertions. This means that those tests could be completely turned off if they are impacting performance by invoking Python with the `-O` flag.

1.4.5 Extending Controllers

One can take advantage of python's object oriented features to extend the functionality of the basic *pyctrl.Controller*. All that is necessary is to inherit from *pyctrl.Controller*.

Inheritance is an easy way to equip controllers with special hardware capabilities. That was the case, for example, with the class *pyctrl.timer.Controller* described in *Devices*. In fact, this new class is so simple that its entire code easily fits here:

```
class Controller(pyctrl.Controller):
    """
    :py:class:`pyctrl.timer.Controller` implements a controller
    with a :py:class:`pyctrl.block.clock.TimerClock`.

    The clock is enabled and disabled automatically when calling
    `start()` and `stop()`.

    :param period: the clock period (default 0.01)
    """

    def __init__(self, **kwargs):

        # period
        self.period = kwargs.pop('period', 0.01)

        # discard argument 'noclock'
        kwargs.pop('noclock', None)

        # Initialize controller
        super().__init__(noclock = True, **kwargs)

    def _reset(self):

        # call super
        super()._reset()
```

```
        # add signal clock
        self.add_signal('clock')

        # add device clock
        self.add_source('clock',
                        ('pyctrl.block.clock', 'TimerClock'),
                        ['clock'],
                        enable = True,
                        kwargs = {'period': self.period})

        # reset clock
        self.set_source('clock', reset=True)
```

Virtually all functionality is provided by the base class *pyctrl.Controller*. The only methods overloaded are `pyctrl.Controller.__init__()` and `pyctrl.Controller._reset()`.

The method `pyctrl.timer.Controller.__init__()` is the standard python constructor, which in this case parses the new attribute `period` before calling the base class `pyctrl.Controller.__init__()` using:

```
super().__init__(**kwargs)
```

Note that this is done using pop as in:

```
self.period = kwargs.pop('period', 0.01)
```

Using `pop()` makes sure the keyword `period` is removed from the dictionary `kwargs`. Any remaining keywords need to be valid attributes of the base class or an exception will be raised.

Most of the action is in the method `pyctrl.Controller._reset()`. In fact, a closer look at `pyctrl.block.container.Controller.__init__()`, which is the method called by `super().__init__(**kwargs)` in `pyctrl.Controller.__init__()`, reveals:

```
    def __init__(self, **kwargs):

        # set enabled as False by default
        if 'enabled' not in kwargs:
            kwargs['enabled'] = False

        # call super
        super().__init__(**kwargs)

        # call _reset
        self._reset()
```

where a call to the method `pyctrl.Controller._reset()` can be spotted after a couple of definitions.

If you overload `pyctrl.Controller._reset()` make sure to call:

```
super()._reset()
```

before doing any other task. This will make sure that whatever tasks that need to be performed by the base class have already taken place and won't undo any of your own initialization.

The method `pyctrl.Controller._reset()` is also called by the method *pyctrl.Controller.reset()*. In fact, one rarely needs to overload any method other than `pyctrl.Controller.__init__()` and `pyctrl.Controller._reset()`.

A typical reason for extending *pyctrl.Controller* is to provide the user with a set of devices that continue to

exist even after a call to *pyctrl.Controller.reset()*. For example, the following code is from `pyctrl.rc.mip.Controller()`:

```
class Controller(pyctrl.rc.Controller):

    def _reset(self):

        # call super
        super()._reset()

        self.add_signals('theta','theta_dot',
                         'encoder1','encoder2',
                         'pwm1','pwm2')

        # add source: imu
        self.add_source('inclinometer',
                        ('pyctrl.rc.mpu9250', 'Inclinometer'),
                        ['theta','theta_dot'])

        # add source: encoder1
        self.add_source('encoder1',
                        ('pyctrl.rc.encoder', 'Encoder'),
                        ['encoder1'],
                        kwargs = {'encoder': 3,
                                  'ratio': 60 * 35.557})

        # add source: encoder2
        self.add_source('encoder2',
                        ('pyctrl.rc.encoder', 'Encoder'),
                        ['encoder2'],
                        kwargs = {'encoder': 2,
                                  'ratio': - 60 * 35.557})

        # add sink: motor1
        self.add_sink('motor1',
                      ('pyctrl.rc.motor', 'Motor'),
                      ['pwm1'],
                      kwargs = {'motor': 3},
                      enable = True)

        # add sink: motor2
        self.add_sink('motor2',
                      ('pyctrl.rc.motor', 'Motor'),
                      ['pwm2'],
                      kwargs = {'motor': 2,
                                'ratio': -100},
                      enable = True)
```

which adds a number of devices to the base class `pyctrl.rc.Controller()` that can be used with the Robotics Cape and the Educational MIP as described in *Interfacing with hardware*.

1.4.6 Writting your own Blocks

The package *pyctrl* is designed so that you can easily extend its functionality by writing simple python code for your own blocks. You can write blocks to support your specific hardware or implement an algorithm that is currently not available in *Module pyctrl.block*.

Your blocks should inherit from *pyctrl.block.Block* or one of its derived class, such as *pyctrl.block.BufferBlock*, which are described next.

1.4.6.1 Extending pyctrl.block.Block

A *pyctrl.block.Block* needs to know how to do two things: respond to calls to *pyctrl.block.Block.read()* and/or *pyctrl.block.Block.write()*. If a block is to be used as a *source* then it needs to respond to *pyctrl.block.Block.read()*, if it is to be used as a *sink* it needs to respond to *pyctrl.block.Block.write()*, and it if is to be used as a *filter* it needs to respond to both.

For example consider the following code for a simple block:

```
import pyctrl.block

class MyOneBlock(pyctrl.block.Source, pyctrl.block.Block):

    def read(self):
        return (1,)
```

Multiple inheritance is used to make sure this block can only be used as a *source* by inheriting from *pyctrl.block.Source* *and* *pyctrl.block.Block*. The order is important! If you try to use `MyOneBlock` as a *sink* or a *filter* an exception will be raised since `MyOneBlock` does not overload *pyctrl.block.Block.write()*. Likewise, *sinks* must inherit from *pyctrl.block.Sink* and *filters* from *pyctrl.block.Filter*.

All this block does is output a *signal* which is the constant *1*. Note that the return value of *pyctrl.block.Block.read()* must be a tuple with numbers or numpy 1D-arrays. You could use your block in a controller like this:

```
# add a MyOneBlock as a source
controller.add_source('mysource',
                      MyOneBlock(),
                      ['signal'])
```

which would write *1* to the *signal* `signal` every time the controller loop is run.

Consider now the slightest more sophisticated block:

```
import pyctrl.block

class MySumBlock(pyctrl.block.Filter, pyctrl.block.Block):

    def __init__(self, **kwargs):

        # you must call super().__init__
        super().__init__(**kwargs)

        # create local buffer
        self.buffer = ()

    def write(self, *values):

        # copy values to buffer
        self.buffer = values

    def read(self):

        # return sum of all values as first entry
        return (sum(self.buffer), )
```

Because `MySumBlock` inherits from *pyctrl.block.Filter* if can be be used a *filter*. It must therefore overload both *pyctrl.block.Block.write()* and *pyctrl.block.Block.read()*. For instance:

```
# add a MySumBlock as a filter
controller.add_filter('myfilter',
                      MySumBlock(),
                      ['signal1','signal2','signal3'],
                      ['sum'])
```

would set the *signal* `sum` to be equal to the sum of the three input *signals* `signal1`, `signal2`, and `signal3`. When placed in a controller loop, the loop will first call `MySumBlock.write()` then `MySumBlock.read()` as if running a code similar to the following:

```
myfilter.write(signal1, signal2, signal3)
(sum, ) = myfilter.read()
```

At the end of a loop iteration the variable `sum` would contain the sum of the three variables `signal1`, `signal2`, and `signal3`. Of course the code run by *pyctrl.Controller* is never explicitly expanded as above.

A couple of important details here. First `MySumBlock.__init__()` calls `pyctrl.block.Block.__init__()` then proceeds to create its own attribute `buffer`. Note that *pyctrl.block.Block()* does not accept positional arguments, only keyword arguments. As you will learn soon, this facilitates handling errors in the constructor. Second the method `MySumBlock.write()` should always take a variable number of arguments, represented by the python construction `*values`. Inside `MySumBlock.write()` the variable `values` is a tuple. Third, because *pyctrl.block.Block.write()* and *pyctrl.block.Block.read()* are called separately, it is often the case that one needs an internal variable to store values to be carried from *pyctrl.block.Block.write()* to *pyctrl.block.Block.read()*. This is so common that *pyctrl.block* provides a specialized class *pyctrl.block.BufferBlock*, which you will learn about in the next section.

1.4.6.2 Extending `pyctrl.block.BufferBlock`

The class *pyctrl.block.BufferBlock* has several features that can facilitate the implementation of blocks. First, py:meth:*pyctrl.block.BufferBlock.read* and *pyctrl.block.BufferBlock.write()* work with a an internal attribute `buffer`, which can be used to carry values from *pyctrl.block.BufferBlock.write()* to *pyctrl.block.BufferBlock.read()*. Second, it support *multiplexing* and *demultiplexing* of inputs as discussed in Section *Multiplexing and demultiplexing*.

Consider as an example the block *pyctrl.block.system.Gain*, which produces an output which correspond to its inputs multiplied by a fixed *gain*. Because *pyctrl.block.system.Gain* does nothing to its inputs when it is written it does not overload *pyctrl.block.BufferBlock.write()*. All the action is on the method *pyctrl.block.system.Gain.write()*:

```
    def write(self, *values):
        """
        Writes product of :py:attr:`gain` times current input to the
        private :py:attr:`buffer`.

        :param vararg values: values
        """

        # call super
        super().write(*values)

        self.buffer = tuple(v * self.gain for v in self.buffer)
```

Note that it starts by calling `super().write(*values)`, which will take care of any multiplexing at the input, followed by the actual calculation, which in this case is performed using a list comprehension:

```
self.buffer = tuple(v * self.gain for v in self.buffer)
```

that produces the desired output tuple.

For another example consider the block *pyctrl.block.system.Sum* and its method *pyctrl.block.system.Sum.write()*:

```
    def write(self, *values):
        """
        Writes product of `gain` times the sum of the current input to the private␣
↪`buffer`.

        :param vararg values: list of values
        :return: tuple with scaled input
        """
        # call super
        super(Gain, self).write(*values)

        self.buffer = (self.gain * numpy.sum(self.buffer, axis=0), )
```

The only new detail here is the use of `super(Gain, self).write(*values)`. This is because *pyctrl.block.system.Sum* inherits from *pyctrl.block.system.Gain*, and you would like to call *pyctrl.block.BufferBlock.write()* instead of *pyctrl.block.system.Gain.write()*.

1.5 Examples

The following examples can be found in the directory `examples`.

1.5.1 hello_world.py

```
def main():

    # import python's standard time module
    import time

    # import Controller and other blocks from modules
    from pyctrl import Controller
    from pyctrl.block import Printer
    from pyctrl.block.clock import TimerClock

    # initialize controller
    hello = Controller()

    # add the signal myclock
    hello.add_signal('myclock')

    # add a TimerClock as a source
    hello.add_source('myclock',
                     TimerClock(period = 1),
                     ['myclock'],
                     enable = True)

    # add a Printer as a sink
    hello.add_sink('message',
```

```
                    Printer(message = 'Hello World!'),
                    ['myclock'],
                    enable = True)

    try:
        # run the controller
        with hello:
            # do nothing for 5 seconds
            time.sleep(5)

    except KeyboardInterrupt:
        pass

    finally:
        print('Done')
```

1.5.2 hello_timer_1.py

```
def main():

    # import python's standard time module
    import time

    # import Controller and other blocks from modules
    from pyctrl.timer import Controller
    from pyctrl.block import Printer

    # initialize controller
    hello = Controller(period = 1)

    # add a Printer as a sink
    hello.add_sink('message',
                    Printer(message = 'Hello World @ {:3.1f} s'),
                    ['clock'],
                    enable = True)

    # print controller info
    print(hello.info('all'))

    try:
        # run the controller
        print('> Run the controller.')

        print('> Do nothing for 5 s with the controller on...')
        with hello:
            # do nothing for 5 seconds
            time.sleep(5)

        print('> Do nothing for 2 s with the controller off...')
        time.sleep(2)

        print('> Do nothing for 5 s with the controller on...')
        with hello:
            # do nothing for 5 seconds
            time.sleep(5)
```

```
        print('> Done with the controller.')

    except KeyboardInterrupt:
        pass
```

1.5.3 hello_timer_2.py

```
def main():

    # import python's standard time module
    import time

    # import Controller and other blocks from modules
    from pyctrl import Controller
    from pyctrl.block import Printer, Constant

    # initialize controller
    hello = Controller()

    # add a Printer as a timer
    hello.add_timer('message',
                    Printer(message = 'Hello World @ {:3.1f} s '),
                    ['clock'], None,
                    period = 1, repeat = True)

    # Add a timer to stop the controller
    hello.add_timer('stop',
                    Constant(value = 0),
                    None, ['is_running'],
                    period = 5, repeat = False)

    # add a Printer as a sink
    hello.add_sink('message',
                    Printer(message = 'Current time {:5.3f} s', endln = '\r'),
                    ['clock'])

    # print controller info
    print(hello.info('all'))

    try:

        # run the controller
        print('> Run the controller.')
        with hello:

            # wait for the controller to finish on its own
            hello.join()

        print('> Done with the controller.')

    except KeyboardInterrupt:
        pass
```

1.5.4 hello_filter_1.py

```
def main():

    # import Controller and other blocks from modules
    from pyctrl.timer import Controller
    from pyctrl.block import Interp, Constant, Printer

    # initialize controller
    Ts = 0.1
    hello = Controller(period = Ts)

    # add pwm signal
    hello.add_signal('pwm')

    # build interpolated input signal
    ts = [0, 1, 2,   3,   4,   5,   5, 6]
    us = [0, 0, 100, 100, -50, -50, 0, 0]

    # add filter to interpolate data
    hello.add_filter('input',
                     Interp(xp = us, fp = ts),
                     ['clock'],
                     ['pwm'])

    # add logger
    hello.add_sink('printer',
                   Printer(message = 'time = {:3.1f} s, motor = {:+6.1f} %',
                           endln = '\r'),
                   ['clock','pwm'])

    # Add a timer to stop the controller
    hello.add_timer('stop',
                    Constant(value = 0),
                    None, ['is_running'],
                    period = 6, repeat = False)

    # print controller info
    print(hello.info('all'))

    try:

        # run the controller
        print('> Run the controller.')
        with hello:

            # wait for the controller to finish on its own
            hello.join()

        print('> Done with the controller.')

    except KeyboardInterrupt:
        pass
```

1.5.5 hello_filter_2.py

```
def main():

    import sys

    # import Controller and other blocks from modules
    from pyctrl.timer import Controller
    from pyctrl.block import Interp, Printer, Constant, Logger

    # initialize controller
    Ts = 0.01
    hello = Controller(period = Ts)

    # add pwm signal
    hello.add_signal('pwm')

    # build interpolated input signal
    ts = [0, 1, 2,   3,   4,   5,   5, 6]
    us = [0, 0, 100, 100, -50, -50, 0, 0]

    # add filter to interpolate data
    hello.add_filter('input',
                     Interp(xp = us, fp = ts),
                     ['clock'],
                     ['pwm'])

    # add logger
    hello.add_sink('printer',
                   Printer(message = 'time = {:3.1f} s, motor = {:+6.1f} %',
                           endln = '\r'),
                   ['clock','pwm'])

    # add logger
    hello.add_sink('logger',
                   Logger(),
                   ['clock','pwm'])

    # Add a timer to stop the controller
    hello.add_timer('stop',
                    Constant(value = 0),
                    None, ['is_running'],
                    period = 6, repeat = False)

    # print controller info
    print(hello.info('all'))

    try:

        # run the controller
        print('> Run the controller.')
        with hello:

            # wait for the controller to finish on its own
            hello.join()

        print('> Done with the controller.')
```

```
    except KeyboardInterrupt:
        pass

    # retrieve data from logger
    data = hello.get_sink('logger', 'log')

    try:

        # import matplotlib
        import matplotlib.pyplot as plt

    except:

        print('! Could not load matplotlib, skipping plots')
        sys.exit(0)

    print('> Will plot')

    try:

        # start plot
        plt.figure()

    except:

        print('! Could not plot graphics')
        print('> Make sure you have a connection to a windows manager')
        sys.exit(0)

    # plot input
    plt.plot(data['clock'], data['motor'], 'b')
    plt.ylabel('pwm (%)')
    plt.xlabel('time (s)')
    plt.ylim((-120,120))
    plt.xlim(0,6)
    plt.grid()

    # show plots
    plt.show()
```

1.5.6 hello_client.py

```
def main():

    # import python's standard time module
    import time

    # import Controller and other blocks from modules
    from pyctrl.client import Controller
    from pyctrl.block import Printer

    # initialize controller
    hello = Controller(host = 'localhost', port = 9999,
                       module = 'pyctrl.timer',
                       kwargs = {'period': 1})
```

```
    # add a Printer as a sink
    hello.add_sink('message',
                   Printer(message = 'Hello World @ {:4.2f}s'),
                   ['clock'],
                   enable = True)

    # print controller information
    print(hello.info('all'))

    try:
        # run the controller
        with hello:
            # do nothing for 5 seconds
            time.sleep(5)

    except KeyboardInterrupt:
        pass
```

1.5.7 simulated_motor_1.py

```
def main():

    # import python's standard math module and numpy
    import math, numpy, sys

    # import Controller and other blocks from modules
    from pyctrl.timer import Controller
    from pyctrl.block import Interp, Logger, Constant
    from pyctrl.block.system import System
    from pyctrl.system.tf import DTTF

    # initialize controller
    Ts = 0.01
    simotor = Controller(period = Ts)

    # build interpolated input signal
    ts = [0, 1, 2,   3,   4,   5,   5, 6]
    us = [0, 0, 100, 100, -50, -50, 0, 0]

    # add pwm signal
    simotor.add_signal('pwm')

    # add filter to interpolate data
    simotor.add_filter('input',
                       Interp(xp = us, fp = ts),
                       ['clock'],
                       ['pwm'])

    # Motor model parameters
    tau = 1/55    # time constant (s)
    g = 0.092     # gain (cycles/sec duty)
    c = math.exp(-Ts/tau)
    d = (g*Ts)*(1-c)/2

    # add motor signals
    simotor.add_signal('encoder')
```

```
# add motor filter
simotor.add_filter('motor',
                   System(model = DTTF(
                       numpy.array((0, d, d)),
                       numpy.array((1, -(1 + c), c)))),
                   ['pwm'],
                   ['encoder'])

# add logger
simotor.add_sink('logger',
                 Logger(),
                 ['clock','pwm','encoder'])

# Add a timer to stop the controller
simotor.add_timer('stop',
                  Constant(value = 0),
                  None, ['is_running'],
                  period = 6, repeat = False)

# print controller info
print(simotor.info('all'))

try:

    # run the controller
    print('> Run the controller.')
    with simotor:

        # wait for the controller to finish on its own
        simotor.join()

    print('> Done with the controller.')

except KeyboardInterrupt:
    pass

finally:
    pass

# read logger
data = simotor.get_sink('logger', 'log')

try:

    # import matplotlib
    import matplotlib.pyplot as plt

except:

    print('! Could not load matplotlib, skipping plots')
    sys.exit(0)

print('> Will plot')

try:

    # start plot
```

```
        plt.figure()

    except:

        print('! Could not plot graphics')
        print('> Make sure you have a connection to a windows manager')
        sys.exit(0)

    # plot pwm
    plt.subplot(2,1,1)
    plt.plot(data['clock'], data['pwm'], 'b')
    plt.ylabel('pwm (%)')
    plt.ylim((-120,120))
    plt.xlim(0,6)
    plt.grid()

    # plot encoder
    plt.subplot(2,1,2)
    plt.plot(data['clock'], data['encoder'],'b')
    plt.ylabel('encoder (cycles)')
    plt.ylim((0,25))
    plt.xlim(0,6)
    plt.grid()

    # show plots
    plt.show()
```

1.5.8 simulated_motor_2.py

```
def main():

    # import python's standard math module and numpy
    import math, numpy, sys

    # import Controller and other blocks from modules
    from pyctrl.timer import Controller
    from pyctrl.block import Interp, Logger, Constant
    from pyctrl.block.system import System, Differentiator
    from pyctrl.system.tf import DTTF, LPF

    # initialize controller
    Ts = 0.01
    simotor = Controller(period = Ts)

    # build interpolated input signal
    ts = [0, 1, 2,   3,   4,   5,   5, 6]
    us = [0, 0, 100, 100, -50, -50, 0, 0]

    # add pwm signal
    simotor.add_signal('pwm')

    # add filter to interpolate data
    simotor.add_filter('input',
                       Interp(xp = us, fp = ts),
                       ['clock'],
                       ['pwm'])
```

```
# Motor model parameters
tau = 1/55   # time constant (s)
g = 0.092    # gain (cycles/sec duty)
c = math.exp(-Ts/tau)
d = (g*Ts)*(1-c)/2

# add motor signals
simotor.add_signal('encoder')

# add motor filter
simotor.add_filter('motor',
                   System(model = DTTF(
                       numpy.array((0, d, d)),
                       numpy.array((1, -(1 + c), c)))),
                   ['pwm'],
                   ['encoder'])

# add motor speed signal
simotor.add_signal('speed')

# add motor speed filter
simotor.add_filter('speed',
                   Differentiator(),
                   ['clock','encoder'],
                   ['speed'])

# add low-pass signal
simotor.add_signal('fspeed')

# add low-pass filter
simotor.add_filter('LPF',
                   System(model = LPF(fc = 5, period = Ts)),
                   ['speed'],
                   ['fspeed'])

# add logger
simotor.add_sink('logger',
                 Logger(),
                 ['clock','pwm','encoder','speed','fspeed'])

# Add a timer to stop the controller
simotor.add_timer('stop',
                  Constant(value = 0),
                  None, ['is_running'],
                  period = 6, repeat = False)

# print controller info
print(simotor.info('all'))

try:

    # run the controller
    print('> Run the controller.')
    with simotor:

        # wait for the controller to finish on its own
        simotor.join()
```

```
    print('> Done with the controller.')

except KeyboardInterrupt:
    pass

finally:
    pass

# read logger
data = simotor.get_sink('logger', 'log')

try:

    # import matplotlib
    import matplotlib.pyplot as plt

except:

    print('! Could not load matplotlib, skipping plots')
    sys.exit(0)

print('> Will plot')

try:

    # start plot
    plt.figure()

except:

    print('! Could not plot graphics')
    print('> Make sure you have a connection to a windows manager')
    sys.exit(0)

# plot pwm
ax1 = plt.gca()

ax1.plot(data['clock'], data['pwm'],'g', label='pwm')
ax1.set_ylabel('pwm (%)')
ax1.set_ylim((-60,120))
ax1.grid()
plt.legend(loc = 2)

# plot velocity
ax2 = plt.twinx()

ax2.plot(data['clock'], data['speed'],'b', label='speed')
ax2.plot(data['clock'], data['fspeed'], 'r', label='fspeed')
ax2.set_ylabel('speed (Hz)')
ax2.set_ylim((-6,12))
ax2.set_xlim(0,6)
ax2.grid()
plt.legend(loc = 1)

# show plots
plt.show()
```

1.5.9 rc_motor.py

```
def main():

    # import python's standard math module and numpy
    import math, numpy, sys

    # import Controller and other blocks from modules
    from pyctrl.timer import Controller
    from pyctrl.block import Interp, Logger, Constant
    from pyctrl.block.system import System, Differentiator
    from pyctrl.system.tf import DTTF, LPF

    # initialize controller
    Ts = 0.01
    simotor = Controller(period = Ts)

    # build interpolated input signal
    ts = [0, 1, 2,   3,   4,   5,   5, 6]
    us = [0, 0, 100, 100, -50, -50, 0, 0]

    # add pwm signal
    simotor.add_signal('pwm')

    # add filter to interpolate data
    simotor.add_filter('input',
                       Interp(xp = us, fp = ts),
                       ['clock'],
                       ['pwm'])

    # Motor model parameters
    tau = 1/55    # time constant (s)
    g = 0.092     # gain (cycles/sec duty)
    c = math.exp(-Ts/tau)
    d = (g*Ts)*(1-c)/2

    # add motor signals
    simotor.add_signal('encoder')

    # add motor filter
    simotor.add_filter('motor',
                       System(model = DTTF(
                           numpy.array((0, d, d)),
                           numpy.array((1, -(1 + c), c)))),
                       ['pwm'],
                       ['encoder'])

    # add motor speed signal
    simotor.add_signal('speed')

    # add motor speed filter
    simotor.add_filter('speed',
                       Differentiator(),
                       ['clock','encoder'],
                       ['speed'])

    # add low-pass signal
    simotor.add_signal('fspeed')
```

```
# add low-pass filter
simotor.add_filter('LPF',
                   System(model = LPF(fc = 5, period = Ts)),
                   ['speed'],
                   ['fspeed'])

# add logger
simotor.add_sink('logger',
                 Logger(),
                 ['clock','pwm','encoder','speed','fspeed'])

# Add a timer to stop the controller
simotor.add_timer('stop',
                  Constant(value = 0),
                  None, ['is_running'],
                  period = 6, repeat = False)

# print controller info
print(simotor.info('all'))

try:

    # run the controller
    print('> Run the controller.')
    with simotor:

        # wait for the controller to finish on its own
        simotor.join()

    print('> Done with the controller.')

except KeyboardInterrupt:
    pass

finally:
    pass

# read logger
data = simotor.get_sink('logger', 'log')

try:

    # import matplotlib
    import matplotlib.pyplot as plt

except:

    print('! Could not load matplotlib, skipping plots')
    sys.exit(0)

print('> Will plot')

try:

    # start plot
    plt.figure()
```

```
    except:

        print('! Could not plot graphics')
        print('> Make sure you have a connection to a windows manager')
        sys.exit(0)

    # plot pwm
    ax1 = plt.gca()

    ax1.plot(data['clock'], data['pwm'],'g', label='pwm')
    ax1.set_ylabel('pwm (%)')
    ax1.set_ylim((-60,120))
    ax1.grid()
    plt.legend(loc = 2)

    # plot velocity
    ax2 = plt.twinx()

    ax2.plot(data['clock'], data['speed'],'b', label='speed')
    ax2.plot(data['clock'], data['fspeed'], 'r', label='fspeed')
    ax2.set_ylabel('speed (Hz)')
    ax2.set_ylim((-6,12))
    ax2.set_xlim(0,6)
    ax2.grid()
    plt.legend(loc = 1)

    # show plots
    plt.show()
```

1.5.10 rc_motor_control.py

```
def main():

    # import python's standard math module and numpy
    import math, numpy, sys

    # import Controller and other blocks from modules
    from pyctrl.rc import Controller
    from pyctrl.block import Interp, Logger, Constant
    from pyctrl.block.system import System, Differentiator, Feedback
    from pyctrl.system.tf import PID

    # initialize controller
    Ts = 0.01
    bbb = Controller(period = Ts)

    # add encoder as source
    bbb.add_source('encoder1',
                   ('pyctrl.rc.encoder', 'Encoder'),
                   ['encoder'],
                   kwargs = {'encoder': 3,
                             'ratio': 60 * 35.557})

    # add motor as sink
    bbb.add_sink('motor1',
                 ('pyctrl.rc.motor', 'Motor'),
```

```
                ['pwm'],
                kwargs = {'motor': 3},
                enable = True)

    # add motor speed signal
    bbb.add_signal('speed')

    # add motor speed filter
    bbb.add_filter('speed',
                   Differentiator(),
                   ['clock','encoder'],
                   ['speed'])

    # calculate PI controller gains
    tau = 1/55   # time constant (s)
    g = 0.092    # gain (cycles/sec duty)

    Kp = 1/g
    Ki = Kp/tau

    print('Controller gains: Kp = {}, Ki = {}'.format(Kp, Ki))

    # build controller block
    pid = System(model = PID(Kp = Kp, Ki = Ki, period = Ts))

    # add motor speed signal
    bbb.add_signal('speed_reference')

    bbb.add_filter('PIcontrol',
                   Feedback(block = pid),
                   ['speed','speed_reference'],
                   ['pwm'])

    # build interpolated input signal
    ts = [0, 1, 2,  3,   4,  5, 5, 6]
    us = [0, 0, 8, 8, -4, -4, 0, 0]

    # add filter to interpolate data
    bbb.add_filter('input',
                   Interp(xp = us, fp = ts),
                   ['clock'],
                   ['speed_reference'])

    # add logger
    bbb.add_sink('logger',
                 Logger(),
                 ['clock','pwm','encoder','speed','speed_reference'])

    # Add a timer to stop the controller
    bbb.add_timer('stop',
                  Constant(value = 0),
                  None, ['is_running'],
                  period = 6, repeat = False)

    # print controller info
    print(bbb.info('all'))

    try:
```

```
        # run the controller
        print('> Run the controller.')

        # set speed_reference
        #bbb.set_signal('speed_reference', 5)

        # reset clock
        bbb.set_source('clock', reset = True)
        with bbb:

            # wait for the controller to finish on its own
            bbb.join()

        print('> Done with the controller.')

    except KeyboardInterrupt:
        pass

    finally:
        pass

    # read logger
    data = bbb.get_sink('logger', 'log')

    try:

        # import matplotlib
        import matplotlib.pyplot as plt

    except:

        print('! Could not load matplotlib, skipping plots')
        sys.exit(0)

    print('> Will plot')

    try:

        # start plot
        plt.figure()

    except:

        print('! Could not plot graphics')
        print('> Make sure you have a connection to a windows manager')
        sys.exit(0)

    # plot pwm
    plt.subplot(2,1,1)
    plt.plot(data['clock'], data['pwm'], 'b')
    plt.ylabel('pwm (%)')
    plt.ylim((-120,120))
    plt.xlim(0,6)
    plt.grid()

    # plot encoder
    plt.subplot(2,1,2)
```

```
    plt.plot(data['clock'], data['encoder'],'b')
    plt.ylabel('position (cycles)')
    plt.ylim((0,25))
    plt.xlim(0,6)
    plt.grid()

    # start plot
    plt.figure()

    # plot pwm
    ax1 = plt.gca()

    ax1.plot(data['clock'], data['pwm'],'g', label='pwm')
    ax1.set_ylabel('pwm (%)')
    ax1.set_ylim((-60,120))
    ax1.grid()
    plt.legend(loc = 2)

    # plot velocity
    ax2 = plt.twinx()

    ax2.plot(data['clock'], data['speed'],'b', label='speed')
    ax2.plot(data['clock'], data['speed_reference'], 'r', label='reference')
    ax2.set_ylabel('speed (Hz)')
    ax2.set_ylim((-6,12))
    ax2.set_xlim(0,6)
    ax2.grid()
    plt.legend(loc = 1)

    # show plots
    plt.show()
```

1.5.11 rc_mip_balance.py

```
def main():

    # import blocks and controller
    from pyctrl.rc.mip import Controller
    from pyctrl.block.container import Container, Input, Output
    from pyctrl.block.system import System, Subtract, Differentiator, Sum, Gain
    from pyctrl.block.nl import ControlledCombination, Product
    from pyctrl.block import Fade, Printer
    from pyctrl.system.ss import DTSS
    from pyctrl.block.logic import CompareAbsWithHysterisis, SetFilter, State
    from rcpy.gpio import GRN_LED, PAUSE_BTN
    from rcpy.led import red

    # export json?
    export_json = False

    # create mip
    mip = Controller()

    # phi is the average of the encoders
    mip.add_signal('phi')
    mip.add_filter('phi',
```

```
                Sum(gain=0.5),
                ['encoder1','encoder2'],
                ['phi'])

# phi dot
mip.add_signal('phi_dot')
mip.add_filter('phi_dot',
                Differentiator(),
                ['clock','phi'],
                ['phi_dot'])

# phi dot and steer reference
mip.add_signals('phi_dot_reference', 'phi_dot_reference_fade')
mip.add_signals('steer_reference', 'steer_reference_fade')

# add fade in filter
mip.add_filter('fade',
                Fade(target = [0, 0.5], period = 5),
                ['clock','phi_dot_reference','steer_reference'],
                ['phi_dot_reference_fade','steer_reference_fade'])

# state-space matrices
A = np.array([[0.913134, 0.0363383],[-0.0692862, 0.994003]])
B = np.array([[0.00284353, -0.000539063], [0.00162443, -0.00128745]])
C = np.array([[-383.009, 303.07]])
D = np.array([[-1.22015, 0]])

B = 2*np.pi*(100/7.4)*np.hstack((-B, B[:,1:]))
D = 2*np.pi*(100/7.4)*np.hstack((-D, D[:,1:]))

ssctrl = DTSS(A,B,C,D)

# state-space controller
mip.add_signals('pwm')
mip.add_filter('controller',
                System(model = ssctrl),
                ['theta_dot','phi_dot','phi_dot_reference_fade'],
                ['pwm'])

# enable pwm only if about small_angle
mip.add_signals('small_angle', 'small_angle_pwm')
mip.add_filter('small_angle_pwm',
                Product(),
                ['small_angle', 'pwm'],
                ['small_angle_pwm'])

# steering biasing
mip.add_filter('steer',
                ControlledCombination(),
                ['steer_reference_fade',
                 'small_angle_pwm','small_angle_pwm'],
                ['pwm1','pwm2'])

# set references
mip.set_signal('phi_dot_reference',0)
mip.set_signal('steer_reference',0.5)

# add supervisor actions on a timer
```

```
    # actions are inside a container so that they are executed all at once
    mip.add_timer('supervisor',
                  Container(),
                  ['theta'],
                  ['small_angle','is_running'],
                  period = 0.5, repeat = True)

    mip.add_signals('timer/supervisor/theta',
                    'timer/supervisor/small_angle')

    mip.add_source('timer/supervisor/theta',
                   Input(),
                   ['theta'])

    mip.add_sink('timer/supervisor/small_angle',
                 Output(),
                 ['small_angle'])

    mip.add_sink('timer/supervisor/is_running',
                 Output(),
                 ['is_running'])

    # add small angle sensor
    mip.add_filter('timer/supervisor/is_angle_small',
                   CompareAbsWithHysterisis(threshold = 0.11,
                                            hysterisis = 0.09,
                                            offset = -0.07,
                                            state = (State.LOW,)),
                   ['theta'],
                   ['small_angle'])

    # reset controller and fade
    mip.add_sink('timer/supervisor/reset_controller',
                 SetFilter(label = ['/controller','/fade'],
                           on_rise = {'reset': True}),
                 ['small_angle'])

    # add green led
    mip.add_sink('timer/supervisor/green_led',
                 ('pyctrl.rc.led', 'LED'),
                 ['small_angle'],
                 kwargs = {'pin': GRN_LED},
                 enable = True)

    # add pause button on a timer
    mip.add_source('timer/supervisor/pause_button',
                   ('pyctrl.rc.button', 'Button'),
                   ['is_running'],
                   kwargs = {'pin': PAUSE_BTN,
                             'invert': True},
                   enable = True)

    # print controller
    print(mip.info('all'))

    # export json?
    if export_json:
```

```
        from pyctrl.flask import JSONEncoder

        # export controller as json
        json = JSONEncoder(sort_keys = True, indent = 4).encode(mip)
        with open('rc_mip_balance.json', 'w') as f:
            f.write(json)

    fd = sys.stdin.fileno()
    old_settings = termios.tcgetattr(fd)
    try:

        print("""
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*                       M I P   B A L A N C E                         *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
""")

        print("""
Hold your MIP upright to start balancing

Use your keyboard to control the mip:

* UP and DOWN arrows move forward and back
* LEFT and RIGHT arrows steer
* / stops forward motion
* . stops steering
* SPACE resets forward motion and steering

""")


        # reset everything
        mip.set_source('clock',reset=True)
        mip.set_source('encoder1',reset=True)
        mip.set_source('encoder2',reset=True)
        mip.set_filter('controller',reset=True)
        mip.set_source('inclinometer',reset=True)

        # turn on red led
        red.on()

        # start the controller
        mip.start()

        print("Press Ctrl-C or press the <PAUSE> button to exit")

        # fire thread to update velocities
        thread = threading.Thread(target = get_arrows,
                                  args = (mip, fd))
        thread.daemon = False
        thread.start()

        # and wait until controller dies
        mip.join()

        # print message
        print("\nDone with balancing")
```

```
    except KeyboardInterrupt:

        print("\nBalancing aborted")

    finally:

        # turn off red led
        red.off()

        # make sure it exits
        mip.set_state(pyctrl.EXITING)

        print("Press any key to exit")

        thread.join()

        termios.tcsetattr(fd, termios.TCSADRAIN, old_settings)
```

1.6 References

CHAPTER

TWO

REFERENCE GUIDE

2.1 Module *pyctrl*

class pyctrl.**Controller**(***kwargs*)

Bases: *pyctrl.block.container.Container*

pyctrl.Controller provides functionality for running signal flow tasks.

A Controller can be in one of three states:

1. IDLE
2. RUNNING
3. EXITING

Upon initialization a Controller state is set to IDLE.

Parameters **kwargs** – should be left empty

Raises pyctrl.ControllerException if any parameters are passed to py:data‘**kwargs‘

reset()

Stop the controller, remove all devices, sources, sinks, filters, and all signals except *is_running* and *duty*.

Objects that inherit from *Controller* can customize *reset()* by overloading the private method *_reset()*.

get_state()

Return the current state of the Controller.

Returns the state of the Controller

set_state(*state*)

Set the current state of the Controller.

Parameters **state** – the state of the controller

start()

Start Controller loop.

stop()

Stop Controller loop.

join()

Wait for Controller thread to terminate.

add_device(*label*, *device_module*, *device_class*, ***kwargs*)

Add device to Container.

Parameters

- **label** (*str*) – the device label
- **device_module** (*str*) – the device module
- **device_class** (*str*) – the device class
- **enable** (*bool*) – if the device needs to be enable and disabled when calling *set_enable* (default False)
- **inputs** (*list*) – a list of input signals (default *[]*)
- **outputs** (*list*) – a list of output signals (default *[]*)
- **verbose** (*bool*) – if verbose issue warning (default *False*)
- **type** (BlockType) – the device type; only required if BlockType.timer (default None)
- **kwargs** (*kwargs*) – other keyword arguments to be passed to the device class initialization

add_filter(*label*, *filter_*, *inputs*, *outputs*, ***kwargs*)
Add filter to Container.

Parameters

- **label** (*str*) – the filter label
- **filter** (pyctrl.block) – the filter block
- **inputs** (*list*) – a list of input signals
- **outputs** (*list*) – a list of output signals
- **order** (*int*) – if positive, set execution order, otherwise add as last (default *-1*)

add_signal(*label*)
Add signal to Container.

Parameters **label** (*str*) – the signal label

add_signals(**labels*)
Add multiple signal to Container.

Parameters **labels** (*vargs*) – the signal labels

add_sink(*label*, *sink*, *inputs*, ***kwargs*)
Add sink to Container.

Parameters

- **label** (*str*) – the sink label
- **sink** (pyctrl.block) – the sink block
- **inputs** (*list*) – a list of input signals
- **order** (*int*) – if positive, set execution order, otherwise add as last (default *-1*)

add_source(*label*, *source*, *outputs*, ***kwargs*)
Add source to Container.

Parameters

- **label** (*str*) – the source label
- **source** (pyctrl.block) – the source block
- **outputs** (*list*) – a list of output signals

- **order** (*int*) – if positive, set execution order, otherwise add as last (default *-1*)

add_timer(*label*, *blk*, *inputs*, *outputs*, *period*, *repeat=True*, ***kwargs*)
Add timer to Container.

Parameters

- **label** (*str*) – the timer label
- **blk** (pyctrl.block) – the timer block
- **inputs** (*list*) – a list of input signals
- **outputs** (*list*) – a list of output signals
- **period** (*int*) – run timer in period seconds
- **repeat** (*bool*) – repeat if True (default True)

find_filter(*value*)
Return label if object value is a filter. Otherwise return None.

Returns the filter label

Return type str

find_sink(*value*)
Return label if object value is a sink. Otherwise return None.

Returns the sink label

Return type str

find_source(*value*)
Return label if object value is a source. Otherwise return None.

Returns the source label

Return type str

find_timer(*value*)
Return label if object value is a timer. Otherwise return None.

Returns the timer label

Return type str

get_filter(*label*, **keys*)
Get attributes from filter. Call method *pyctrl.block.Block.get()*.

Parameters

- **label** (*str*) – the filter label
- **keys** (*vargs*) – the keys of the attributes to get

Returns dictionary of attributes

Return type dict

get_parent()
Get controlller reference.

Returns parent

get_signal(*label*)
Get the value of signal.

Parameters **label** (*str*) – the signal label

Returns the signal value

get_signals(**labels*)
Get the values of signals.

Parameters **labels** (*vargs*) – the signal labels

Returns the signal values

Return type list

get_sink(*label*, **keys*)
Get attributes from sink. Call method *pyctrl.block.Block.get()*.

Parameters

- **label** (*str*) – the sink label
- **keys** (*vargs*) – the keys of the attributes to get

Returns dictionary of attributes

Return type dict

get_source(*label*, **keys*)
Get attributes from source. Call method *pyctrl.block.Block.get()*.

Parameters

- **label** (*str*) – the source label
- **keys** (*vargs*) – the keys of the attributes to get

Returns dictionary of attributes or single value

Return type dict or value

get_timer(*label*, **keys*)
Get attributes from timer. Call method *pyctrl.block.Block.get()*.

Parameters

- **label** (*str*) – the timer label
- **keys** (*vargs*) – the keys of the attributes to get

Returns dictionary of attributes

Return type dict

get_type()
Get block `type`. `type` is a member of class *pyctrl.block.BlockType*

Returns `type`

html(**keys*)
Format *pyctrl.block.Block* in HTML.

By default uses the result of *pyctrl.block.Block.get()*.

Parameters **keys** – string or tuple of strings with property names

Raise `KeyError` if `key` is not defined

info(**vargs*, ***kwargs*)
Returns a string with information on the Container.

Parameters **options** – can be one of *signals*, *devices*, *sources*, *filters*, *sinks*, *timers*, *all*, *summary*, or *class*

Returns string with information on the Container

is_enabled()
Return `enabled` state.

Returns `enabled`

list_filters()
List of the filters currently on Container.

Returns a list of filter labels

Return type list

list_signals()
List of the signals currently on Container.

Returns a list of signal labels

Return type list

list_sinks()
List of the sinks currently on Container.

Returns a list of sink labels

Return type list

list_sources()
List of the sources currently on Container.

Returns a list of source labels

Return type list

list_timers()
List of the timers currently on Container.

Returns a list of timer labels

Return type list

read()
Read from *pyctrl.block.container.Container*.

Returns values

Retype tuple

Raise `pyctrl.block.container.ContainerException` if block does not support read

read_filter(*label*)
Read from filter. Call method method *pyctrl.block.Block.read()*.

Parameters **label** (*str*) – the filter label

read_source(*label*)
Read from source. Call method *pyctrl.block.Block.read()*.

Parameters **label** (*str*) – the source label

remove_filter(*label*)
Remove filter from Container.

Parameters **label** (*str*) – the filter label

remove_signal(*label*)
Remove signal from Container.

Parameters **label** (*str*) – the signal label to be removed

remove_sink(*label*)
Remove sink from Container.

Parameters **label** (*str*) – the sink label

remove_source(*label*)
Remove source from Container.

Parameters **label** (*str*) – the source label

remove_timer(*label*)
Remove timer from Container.

Parameters **label** (*str*) – the timer label

set(*exclude=()*, ***kwargs*)
Set properties of *pyctrl.block.Block*.

Parameters

- **exclude** (*tuple*) – attributes to exclude (default ())
- **reset** (*bool*) – if True calls *reset()*
- **enabled** (*bool*) – set enabled attribute
- **parent** (pyctrl.block.container.Container) – set parent attribute
- **kwargs** (*kwargs*) – other keyword arguments

Raise *pyctrl.block.BlockException* if any of the kwargs is left unprocessed

set_enabled(*enabled=True*)
Enable Container.

set_filter(*label*, ***kwargs*)
Set filter attributes. Call method *pyctrl.block.Block.set()*.

All attributes must be passed as key-value pairs and vary depending on the type of block.

Parameters

- **label** (*str*) – the filter label
- **inputs** (*list*) – set filter input signals
- **outputs** (*list*) – set filter output signals
- **kwargs** (*kwargs*) – other key-value pairs of attributes

set_parent(*parent*)
Set controlller reference.

:param pyctrl.block.containter.Container parent: parent container

set_signal(*label*, *value*)
Set the value of signal. Call method *pyctrl.block.Block.set()*.

Parameters

- **label** (*str*) – the signal label
- **value** – the value to be set

set_sink(*label*, ***kwargs*)
Set sink attributes. Call method *pyctrl.block.Block.set()*.

All attributes must be passed as key-value pairs and vary depending on the type of block.

Parameters

- **label** (*str*) – the sink label
- **inputs** (*list*) – set sink input signals
- **kwargs** (*kwargs*) – other key-value pairs of attributes

set_source(*label*, ***kwargs*)
Set source attributes. Call method *pyctrl.block.Block.set()*.

All attributes must be passed as key-value pairs and vary depending on the type of block.

Parameters

- **label** (*str*) – the source label
- **outputs** (*list*) – set source output signals
- **kwargs** (*kwargs*) – other key-value pairs of attributes

set_timer(*label*, ***kwargs*)
Set timer attributes. Call method *pyctrl.block.Block.set()*.

All attributes must be passed as key-value pairs and vary depending on the type of block.

Parameters

- **label** (*str*) – the timer label
- **inputs** (*list*) – set timer input signals
- **outputs** (*list*) – set timer output signals
- **kwargs** (*kwargs*) – other key-value pairs of attributes

write(**values*)
Write to *pyctrl.block.container.Container*.

Parameters **values** (*vararg*) – values

Raise `pyctrl.block.container.ContainerException` if block does not support write

write_filter(*label*, **values*)
Write to filter. Call method method *pyctrl.block.Block.write()*.

Parameters

- **label** (*str*) – the filter label
- **values** (*vargs*) – the values to write to filter

write_sink(*label*, **values*)
Write to sink. Call method *pyctrl.block.Block.write()*.

Parameters

- **label** (*str*) – the sink label
- **values** (*vargs*) – the values to write to sink

2.2 Module *pyctrl.timer*

class pyctrl.timer.**Controller**(***kwargs*)
Bases: *pyctrl.Controller*

pyctrl.timer.Controller implements a controller with a *pyctrl.block.clock.TimerClock*.

The clock is enabled and disabled automatically when calling *start()* and *stop()*.

Parameters **period** – the clock period (default 0.01)

2.3 Module *pyctrl.client*

class pyctrl.client.**Controller**(***kwargs*)
Bases: *pyctrl.Controller*

pyctrl.client.Controller provides a controller that can remotely interact with a server.

Parameters

- **host** – host name or id address (default: 'localhost')
- **port** – port numer (default: 9999)

2.4 Module *pyctrl.server*

pyctrl.server.**verbose**(*value=1*)
Set verbose level

Parameters **value** – verbose level (default = 1)

pyctrl.server.**version**()
Return server version

Returns server version

Retype str

pyctrl.server.**help**(*key*)
Return help on available commands

Parameters **key** – command key

pyctrl.server.**reset**(***kwargs*)
Reset controller

Parameters

- **module** (*str*) – name of the controller module (default = 'pyctrl')
- **pyctrl_class** (*str*) – name of the controller class (default = 'Controller')
- **kwargs** (*kwargs*) – other key-value pairs of attributes

pyctrl.server.**set_controller**(*_controller=<class 'pyctrl.Controller'> with: 0 timer(s), 2 signal(s), 0 source(s), 0 filter(s), and 0 sink(s)*)
Set controller commands

Parameters **_controller** – an instance of *pyctrl.Controller*

class pyctrl.server.**Handler**(*request*, *client_address*, *server*)
Bases: socketserver.StreamRequestHandler

Handles socket controller requests

2.5 Module *pyctrl.block*

This module provides the basic building blocks for implementing controllers.

class pyctrl.block.**BlockType**
Bases: enum.Enum

An enumeration.

exception pyctrl.block.**BlockException**
Bases: Exception

Exception class for blocks

exception pyctrl.block.**BlockWarning**
Bases: Warning

Warning class for blocks

class pyctrl.block.**Source**(**kwargs*)
Bases: object

Base class for all source blocks.

get_type()
Get block type. type is a member of class *pyctrl.block.BlockType*

Returns type

write(**values*)
Write to *pyctrl.block.Block*.

Parameters **values** (*vararg*) – values

Raise *pyctrl.block.BlockException* if block does not support write

class pyctrl.block.**Sink**(**kwargs*)
Bases: object

Base class for all sink blocks.

get_type()
Get block type. type is a member of class *pyctrl.block.BlockType*

Returns type

read()
Read from *pyctrl.block.Block*.

Returns values

Retype tuple

Raise *pyctrl.block.BlockException* if block does not support read

class pyctrl.block.**Filter**(**kwargs*)
Bases: object

Base class for all filter blocks.

get_type()
Get block `type`. `type` is a member of class *pyctrl.block.BlockType*

Returns `type`

class pyctrl.block.**Block**(**kwargs*)
Bases: `object`

pyctrl.block.Block provides the basic functionality for all types of blocks.

pyctrl.block.Block does not take any parameters other than `enable`

Parameters

- **enable** (*bool*) – set block as enabled (default True)
- **kwargs** (*kwargs*) – additional keyword arguments

Raise *pyctrl.block.BlockException* if any of the `kwargs` is left unprocessed

is_enabled()
Return `enabled` state.

Returns `enabled`

set_enabled(*enabled=True*)
Set `enabled` state.

Parameters **enabled** (*bool*) – True or False (default True)

set_parent(*parent*)
Set `controlller` reference.

:param `pyctrl.block.containter.Container` parent: parent container

get_parent()
Get `controlller` reference.

Returns `parent`

reset()
Reset *pyctrl.block.Block*.

Does nothing here but allows another *pyctrl.block.Block* to reset itself.

set(*exclude=()*, ***kwargs*)
Set properties of *pyctrl.block.Block*.

Parameters

- **exclude** (*tuple*) – attributes to exclude (default ())
- **reset** (*bool*) – if True calls *reset()*
- **enabled** (*bool*) – set enabled attribute
- **parent** (`pyctrl.block.container.Container`) – set parent attribute
- **kwargs** (*kwargs*) – other keyword arguments

Raise *pyctrl.block.BlockException* if any of the kwargs is left unprocessed

get(**keys*, *exclude=()*)
Get properties of blocks. For example:

```
block.get('enabled')
```

will retrieve the value of the property `enabled`. Returns a tuple with key values if argument `keys` is a list.

Parameters

- **keys** – string or tuple of strings with property names
- **exclude** (*tuple*) – tuple with keys never to be returned (Default ())

Raise `KeyError` if `key` is not defined

html(**keys*)
Format *pyctrl.block.Block* in HTML.

By default uses the result of *pyctrl.block.Block.get()*.

Parameters **keys** – string or tuple of strings with property names

Raise `KeyError` if `key` is not defined

read()
Read from *pyctrl.block.Block*.

Returns values

Retype tuple

Raise *pyctrl.block.BlockException* if block does not support read

write(**values*)
Write to *pyctrl.block.Block*.

Parameters **values** (*vararg*) – values

Raise *pyctrl.block.BlockException* if block does not support write

class pyctrl.block.**BufferBlock**(***kwargs*)
Bases: *pyctrl.block.Block*

pyctrl.block.BufferBlock provides the basic functionality for blocks that implement *pyctrl.block.Block.read* and *pyctrl.block.Block.write* through a local buffer `buffer`.

A *pyctrl.block.BufferBlock* has the property `buffer`.

Writing to a *pyctrl.block.BufferBlock* writes to the `buffer`.

Reading from a *pyctrl.block.BufferBlock* reads from the `buffer`.

Multiplexing and demultiplexing options are available.

If `mux` is False (`demux` is False) then *pyctrl.block.BufferBlock.read()* (*pyctrl.block.BufferBlock.write()*) are simply copied to (from) the `buffer`.

If `mux` is True then *pyctrl.block.BufferBlock.read()* writes a numpy array with the contents of :py:data‘*values‘ to `buffer`.

If `demux` is True then *pyctrl.block.BufferBlock.write()* splits `buffer` into a tuple with scalar entries.

Parameters

- **mux** (*bool*) – mux flag (default False)
- **demux** (*bool*) – demux flag (default False)

get(**keys*, *exclude=()*)
Get properties of a *pyctrl.block.BufferBlock*.

This method excludes `buffer` from the list of properties.

Parameters

- **keys** – string or tuple of strings with property names
- **exclude** (*tuple*) – tuple with keys never to be returned (Default ())

set(*exclude=()*, ***kwargs*)
Set properties of a *pyctrl.block.BufferBlock*.

This method excludes `buffer` from the list of properties.

Parameters

- **exclude** (*tuple*) – attributes to exclude (default ())
- **kwargs** (*kwargs*) – other keyword arguments

write(**values*)
Writes to the private `buffer` property.

If `mux` is False then **values* are simply copied to the `buffer`.

If `mux` is True then **values* writes a numpy array with the contents of **values* to the first entry of `buffer`.

Parameters **values** (*vararg*) – list of values

read()
Returns the private `buffer` property.

If `demux` is False then read returns a copy of the local `buffer`.

If `demux` is True then `buffer` is split into a tuple with scalar entries.

Returns `buffer`

class pyctrl.block.**ShortCircuit**(***kwargs*)
Bases: *pyctrl.block.Filter*, *pyctrl.block.BufferBlock*

pyctrl.block.ShortCircuit copies input to the output, that is

$y = u$

class pyctrl.block.**Printer**(***kwargs*)
Bases: *pyctrl.block.Sink*, *pyctrl.block.Block*

pyctrl.block.Printer prints the values of its input signals.

Parameters

- **endln** (*str*) – end-of-line character (default *'\n'*)
- **frmt** (*str*) – format string (default *{: 12.4f}*)
- **sep** (*str*) – field separator (default ' ')
- **message** (*str*) – message to print (default *None*)
- **file** (*file*) – file to print on (default *sys.stdout*)

set(*exclude=()*, ***kwargs*)
Set properties of *pyctrl.block.Printer*.

Parameters

- **exclude** (*tuple*) – attributes to exclude (default ())
- **endl** (*str*) – end-of-line character

- **frmt** (*str*) – format string
- **sep** (*str*) – field separator
- **message** (*str*) – message to print
- **file** (*file*) – file to print on
- **kwargs** (*kwargs*) – other keyword arguments

write(**values*)
Write formated entries of *values* to *file*.

class pyctrl.block.**Constant**(***kwargs*)
Bases: *pyctrl.block.Source*, *pyctrl.block.BufferBlock*

pyctrl.block.Constant outputs a constant.

Parameters **value** – constant

set(*exclude=()*, ***kwargs*)
Set properties of *pyctrl.block.Constant*.

Parameters

- **exclude** (*tuple*) – attributes to exclude (default ())
- **value** – value or list with values
- **kwargs** (*kwargs*) – other keyword arguments

class pyctrl.block.**Signal**(***kwargs*)
Bases: *pyctrl.block.Source*, *pyctrl.block.BufferBlock*

pyctrl.block.Signal outputs values corresponding to its attribute `signal` sequentially each time *pyctrl.block.BufferBlock.read()* is called.

If `repeat` is True, signal repeats periodically.

Parameters

- **signal** – `numpy.ndarray` or list with values
- **repeat** (*bool*) – if True then signal repeats periodically

reset()
Reset *pyctrl.block.Signal* index back to `0`.

set(*exclude=()*, ***kwargs*)
Set properties of *pyctrl.block.Signal*.

Parameters

- **exclude** (*tuple*) – attributes to exclude (default ())
- **signal** – `numpy.ndarray` or list with values
- **repeat** (*bool*) – if True then signal repeats periodically
- **kwargs** (*kwargs*) – other keyword arguments

read()
Read from *pyctrl.block.Signal*.

If `repeat` is True, `index` becomes `0` after end of `signal`.

Returns current value of `signal` and increments current `index`.

class pyctrl.block.**Interp**(***kwargs*)

Bases: *pyctrl.block.Filter*, *pyctrl.block.BufferBlock*

pyctrl.block.Interp outputs values of a vector `fp` interpolated according to the vector `xp` each time *pyctrl.block.BufferBlock.read()* is called.

If `repeat` is True, signal repeats periodically.

Parameters

- **xp** – `numpy.ndarray` or list with the x-coordinates of the data points, must be increasing
- **fp** – `numpy.ndarray` or list with the y-coordinates
- **left** (*float*) – value to return for x < xp[0] (default is fp[0]).
- **right** (*float*) – value to return for x > xp[-1] (default is fp[-1]).
- **period** (*float*) – a period for the x-coordinates; parameters left and right are ignored if period is specified (default None)

reset()

Reset *pyctrl.block.Interp* so that the input of the next call to *pyctrl.block.Signal.read()* become the origin.

set(*exclude=()*, ***kwargs*)

Set properties of *pyctrl.block.Interp*.

Parameters

- **exclude** (*tuple*) – attributes to exclude (default ())
- **xp** – `numpy.ndarray` or list with the x-coordinates of the data points, must be increasing
- **fp** – `numpy.ndarray` or list with the y-coordinates
- **left** (*float*) – value to return for x < xp[0]
- **right** (*float*) – value to return for x > xp[-1]
- **period** (*float*) – a period for the x-coordinates; parameters left and right are ignored if period is specified
- **kwargs** (*kwargs*) – other keyword arguments

write(**values*)

Writes input to the private `buffer`. Input is the interpolating variable.

read()

Read from *pyctrl.block.Interp*.

Returns current interpolated value using `numpy.iterp()`.

class pyctrl.block.**Fade**(***kwargs*)

Bases: *pyctrl.block.Filter*, *pyctrl.block.BufferBlock*

`pyctrl.block.FadeIn` outputs values of a vector `fp` interpolated according to the vector `xp` each time *pyctrl.block.BufferBlock.read()* is called.

If `repeat` is True, signal repeats periodically.

Parameters **period** (*float*) – a period for the x-coordinates; parameters left and right are ignored if period is specified (default None)

reset()
Reset *pyctrl.block.Interp* so that the input of the next call to *pyctrl.block.Signal.read()* become the origin.

set(*exclude=()*, ***kwargs*)
Set properties of *pyctrl.block.Interp*.

Parameters

- **exclude** (*tuple*) – attributes to exclude (default ())
- **period** (*float*) – a period for the x-coordinates; parameters left and right are ignored if period is specified
- **kwargs** (*kwargs*) – other keyword arguments

write(**values*)
Writes input to the private `buffer`. First input is the interpolating variable.

read()
Read from *pyctrl.block.Signal*.

Returns current interpolated value using `numpy.iterp()`.

class pyctrl.block.**Map**(***kwargs*)
Bases: *pyctrl.block.Filter*, *pyctrl.block.BufferBlock*

A *pyctrl.block.Map* block applies 'function' to each input and returns tuple with results.

Parameters **function** – the function to be applied (default identity)

write(**values*)
pyctrl.block.Map write.

Returns tuple with the result of `function` applied to each input.

class pyctrl.block.**Apply**(***kwargs*)
Bases: *pyctrl.block.Filter*, *pyctrl.block.BufferBlock*

The Block *pyctrl.block.Apply* applies `function` to all inputs and returns a tuple with the result.

Parameters **function** – the function to be applied (default identity)

write(**values*)
pyctrl.block.Apply write.

Returns tuple with the result of `function` applied to all inputs.

class pyctrl.block.**Logger**(*number_of_rows=12000*, *number_of_columns=0*, ***kwargs*)
Bases: *pyctrl.block.Sink*, *pyctrl.block.Block*

pyctrl.block.Logger stores signals into an array.

Parameters

- **number_of_rows** (*int*) – number of stored rows (default 12000)
- **number_of_columns** (*int*) – number of columns (default 0)
- **labels** (*list*) – list with labels
- **auto_reset** (*bool*) – auto reset flag

get(**keys*, *exclude=()*)
Get properties of *pyctrl.block.Logger*.

The special key *log* can be used to retrieve the current stored entries. It calls the method *pyctrl.block.Logger.get_log()*.

Parameters

- **keys** – string or tuple of strings with property names
- **exclude** (*tuple*) – tuple with keys never to be returned (Default ())

Raise KeyError if key is not defined

set(*exclude=()*, ***kwargs*)
Set properties of *pyctrl.block.Logger*.

Parameters

- **labels** (*list*) – list with labels
- **auto_reset** (*bool*) – auto reset flag
- **exclude** (*tuple*) – attributes to exclude
- **kwargs** (*kwargs*) – other keyword arguments

Raise *BlockException* if any of the kwargs is left unprocessed

get_log()
Retrieve current entries of *pyctrl.block.Logger*.

If property *labels* is set return entries as a dictionary with keys from *labels*.

If property *labels* is *None* return entries in an array where each column correspond to an input. Inputs which are vectors are flattened.

Returns dictionary or numpy array with current entries.

class pyctrl.block.**Wrap**(***kwargs*)
Bases: *pyctrl.block.Filter*, *pyctrl.block.BufferBlock*

pyctrl.block.Wrap attempt to make a possible discontinuous angular signal into a continuous one.

Use the attribute *scaling* to change units. Default is radians.

Parameters

- **turns** (*int*) – the current number of turns
- **theta** (*float*) – current unwraped value of the signal
- **threshold** (*int*) – the threshold to use when deciding whether a discontinuity has been reached (Default 0.25 * scaling)
- **scaling** (*float*) – scaling factor (default 2*pi)

2.6 Module *pyctrl.block.container*

This module provides the basic building blocks for implementing Containers.

class pyctrl.block.container.**Input**(***kwargs*)
Bases: *pyctrl.block.Source*, *pyctrl.block.BufferBlock*

pyctrl.block.container.Input provides a block that connects a container input signals to local container signals .

class pyctrl.block.container.**Output**(***kwargs*)
Bases: *pyctrl.block.Sink*, *pyctrl.block.BufferBlock*

pyctrl.block.container.Output provides a block that connects local container signals to a container output signals.

class pyctrl.block.container.**Container**(***kwargs*)
Bases: *pyctrl.block.Filter*, *pyctrl.block.Block*

pyctrl.block.container.Container provides a block that can contain other blocks.

reset()
Reset all sources, sinks, filters, and timers.

html(**keys*)
Format *pyctrl.block.Block* in HTML.

By default uses the result of *pyctrl.block.Block.get()*.

Parameters **keys** – string or tuple of strings with property names

Raise `KeyError` if `key` is not defined

info(**vargs*, ***kwargs*)
Returns a string with information on the Container.

Parameters **options** – can be one of *signals*, *devices*, *sources*, *filters*, *sinks*, *timers*, *all*, *summary*, or *class*

Returns string with information on the Container

add_signal(*label*)
Add signal to Container.

Parameters **label** (*str*) – the signal label

add_signals(**labels*)
Add multiple signal to Container.

Parameters **labels** (*vargs*) – the signal labels

remove_signal(*label*)
Remove signal from Container.

Parameters **label** (*str*) – the signal label to be removed

set_signal(*label*, *value*)
Set the value of signal. Call method *pyctrl.block.Block.set()*.

Parameters

- **label** (*str*) – the signal label
- **value** – the value to be set

get_signal(*label*)
Get the value of signal.

Parameters **label** (*str*) – the signal label

Returns the signal value

get_signals(**labels*)
Get the values of signals.

Parameters **labels** (*vargs*) – the signal labels

Returns the signal values

Return type list

list_signals()
List of the signals currently on Container.

Returns a list of signal labels

Return type list

add_source(*label*, *source*, *outputs*, ***kwargs*)
Add source to Container.

Parameters

- **label** (*str*) – the source label
- **source** (pyctrl.block) – the source block
- **outputs** (*list*) – a list of output signals
- **order** (*int*) – if positive, set execution order, otherwise add as last (default *-1*)

remove_source(*label*)
Remove source from Container.

Parameters **label** (*str*) – the source label

set_source(*label*, ***kwargs*)
Set source attributes. Call method *pyctrl.block.Block.set()*.

All attributes must be passed as key-value pairs and vary depending on the type of block.

Parameters

- **label** (*str*) – the source label
- **outputs** (*list*) – set source output signals
- **kwargs** (*kwargs*) – other key-value pairs of attributes

get_source(*label*, **keys*)
Get attributes from source. Call method *pyctrl.block.Block.get()*.

Parameters

- **label** (*str*) – the source label
- **keys** (*vargs*) – the keys of the attributes to get

Returns dictionary of attributes or single value

Return type dict or value

read_source(*label*)
Read from source. Call method *pyctrl.block.Block.read()*.

Parameters **label** (*str*) – the source label

list_sources()
List of the sources currently on Container.

Returns a list of source labels

Return type list

add_sink(*label*, *sink*, *inputs*, ***kwargs*)
Add sink to Container.

Parameters

- **label** (*str*) – the sink label
- **sink** (pyctrl.block) – the sink block
- **inputs** (*list*) – a list of input signals
- **order** (*int*) – if positive, set execution order, otherwise add as last (default *-1*)

remove_sink(*label*)
Remove sink from Container.

Parameters **label** (*str*) – the sink label

set_sink(*label*, ***kwargs*)
Set sink attributes. Call method *pyctrl.block.Block.set()*.

All attributes must be passed as key-value pairs and vary depending on the type of block.

Parameters

- **label** (*str*) – the sink label
- **inputs** (*list*) – set sink input signals
- **kwargs** (*kwargs*) – other key-value pairs of attributes

get_sink(*label*, **keys*)
Get attributes from sink. Call method *pyctrl.block.Block.get()*.

Parameters

- **label** (*str*) – the sink label
- **keys** (*vargs*) – the keys of the attributes to get

Returns dictionary of attributes

Return type dict

find_sink(*value*)
Return label if object value is a sink. Otherwise return None.

Returns the sink label

Return type str

find_filter(*value*)
Return label if object value is a filter. Otherwise return None.

Returns the filter label

Return type str

find_source(*value*)
Return label if object value is a source. Otherwise return None.

Returns the source label

Return type str

find_timer(*value*)
Return label if object value is a timer. Otherwise return None.

Returns the timer label

Return type str

write_sink(*label*, **values*)
Write to sink. Call method *pyctrl.block.Block.write()*.

Parameters

- **label** (*str*) – the sink label
- **values** (*vargs*) – the values to write to sink

list_sinks()
List of the sinks currently on Container.

Returns a list of sink labels

Return type list

add_filter(*label*, *filter_*, *inputs*, *outputs*, ***kwargs*)
Add filter to Container.

Parameters

- **label** (*str*) – the filter label
- **filter** (pyctrl.block) – the filter block
- **inputs** (*list*) – a list of input signals
- **outputs** (*list*) – a list of output signals
- **order** (*int*) – if positive, set execution order, otherwise add as last (default *-1*)

remove_filter(*label*)
Remove filter from Container.

Parameters **label** (*str*) – the filter label

set_filter(*label*, ***kwargs*)
Set filter attributes. Call method *pyctrl.block.Block.set()*.

All attributes must be passed as key-value pairs and vary depending on the type of block.

Parameters

- **label** (*str*) – the filter label
- **inputs** (*list*) – set filter input signals
- **outputs** (*list*) – set filter output signals
- **kwargs** (*kwargs*) – other key-value pairs of attributes

get_filter(*label*, **keys*)
Get attributes from filter. Call method *pyctrl.block.Block.get()*.

Parameters

- **label** (*str*) – the filter label
- **keys** (*vargs*) – the keys of the attributes to get

Returns dictionary of attributes

Return type dict

read_filter(*label*)
Read from filter. Call method method *pyctrl.block.Block.read()*.

Parameters **label** (*str*) – the filter label

write_filter(*label*, **values*)
Write to filter. Call method method *pyctrl.block.Block.write()*.

Parameters

- **label** (*str*) – the filter label
- **values** (*vargs*) – the values to write to filter

list_filters()
List of the filters currently on Container.

Returns a list of filter labels

Return type list

add_device(*label*, *device_module*, *device_class*, ***kwargs*)
Add device to Container.

Parameters

- **label** (*str*) – the device label
- **device_module** (*str*) – the device module
- **device_class** (*str*) – the device class
- **enable** (*bool*) – if the device needs to be enable and disabled when calling *set_enable* (default False)
- **inputs** (*list*) – a list of input signals (default *[]*)
- **outputs** (*list*) – a list of output signals (default *[]*)
- **verbose** (*bool*) – if verbose issue warning (default *False*)
- **type** (BlockType) – the device type; only required if BlockType.timer (default None)
- **kwargs** (*kwargs*) – other keyword arguments to be passed to the device class initialization

add_timer(*label*, *blk*, *inputs*, *outputs*, *period*, *repeat=True*, ***kwargs*)
Add timer to Container.

Parameters

- **label** (*str*) – the timer label
- **blk** (pyctrl.block) – the timer block
- **inputs** (*list*) – a list of input signals
- **outputs** (*list*) – a list of output signals
- **period** (*int*) – run timer in period seconds
- **repeat** (*bool*) – repeat if True (default True)

remove_timer(*label*)
Remove timer from Container.

Parameters **label** (*str*) – the timer label

set_timer(*label*, ***kwargs*)
Set timer attributes. Call method *pyctrl.block.Block.set()*.

All attributes must be passed as key-value pairs and vary depending on the type of block.

Parameters

- **label** (*str*) – the timer label
- **inputs** (*list*) – set timer input signals
- **outputs** (*list*) – set timer output signals
- **kwargs** (*kwargs*) – other key-value pairs of attributes

get_timer(*label*, **keys*)
Get attributes from timer. Call method *pyctrl.block.Block.get()*.

Parameters

- **label** (*str*) – the timer label
- **keys** (*vargs*) – the keys of the attributes to get

Returns dictionary of attributes

Return type dict

list_timers()
List of the timers currently on Container.

Returns a list of timer labels

Return type list

write(**values*)
Write to *pyctrl.block.container.Container*.

Parameters **values** (*vararg*) – values

Raise pyctrl.block.container.ContainerException if block does not support write

read()
Read from *pyctrl.block.container.Container*.

Returns values

Retype tuple

Raise pyctrl.block.container.ContainerException if block does not support read

set_enabled(*enabled=True*)
Enable Container.

2.7 Module *pyctrl.block.clock*

class pyctrl.block.clock.**Clock**(***kwargs*)
Bases: *pyctrl.block.Source*, *pyctrl.block.Block*

pyctrl.block.clock.Clock provides a basic clock that writes the current time to its output.

reset()
Reset *pyctrl.block.clock.Clock* by setting the origin of time to the present time and the clock count to zero.

get(**keys*, *exclude=()*)
Get properties of *pyctrl.block.clock.Clock*.

Available attributes are:

1. `average_period`
2. `time_origin`
3. `count`

The elapsed time since initialization or last reset can be obtained using the method *pyctrl.block.clock.Clock.read()*.

Parameters

- **keys** – string or tuple of strings with property names
- **exclude** (*tuple*) – keys never to be returned (default ())

read()
Read from *pyctrl.block.clock.Clock*.

Returns tuple with elapsed time since initialization or last reset

calculate_average_period()
Calculate the average period since *pyctrl.block.clock.Clock* was initialized or reset.

Returns average period

Retype float

calibrate(*eps=0.01*, *N=100*, *K=20*)
Calibration routine that attempts to callibrate clock by fine tuning the clock's period.

pyctrl.block.clock.Clock must support *get('period')* and *set(period = float)* must be able to accept arbitrary floats as periods.

class pyctrl.block.clock.**TimerClock**(***kwargs*)
Bases: *pyctrl.block.clock.Clock*

pyctrl.block.clock.TimerClock provides a clock that reads the current time periodically.

Parameters **period** (*float*) – period in seconds

set(*exclude=()*, ***kwargs*)
Set properties of *pyctrl.block.clock.TimerClock*.

Parameters

- **exclude** (*tuple*) – attributes to exclude
- **period** (*float*) – clock period
- **kwargs** (*kwargs*) – other keyword arguments

Raise *pyctrl.block.BlockException* if any of the `kwargs` is left unprocessed

get(**keys*, *exclude=()*)
Get properties of *pyctrl.block.clock.TimerClock*.

Available attributes are those from *pyctrl.block.clock.Clock.get()* and:

1. `period`

The elapsed time since initialization or last reset can be obtained using the method *pyctrl.block.clock.TimerClock.read()*.

Parameters

- **keys** – string or tuple of strings with property names
- **exclude** (*tuple*) – keys never to be returned (Default ())

set_enabled(*enabled=True*)
Set *pyctrl.block.clock.TimerClock* `enabled` state.

Parameters **enabled** (*bool*) – True or False (default True)

read()
Read from *pyctrl.block.clock.TimerClock*.

Returns tuple with elapsed time since initialization or last reset

2.8 Module *pyctrl.block.system*

This module provides blocks for dynamic systems.

class pyctrl.block.system.**System**(***kwargs*)
Bases: *pyctrl.block.Filter*, *pyctrl.block.BufferBlock*

pyctrl.block.system.System is a wrapper for a time-invariant dynamic system model.

Inputs of this block are always multiplexed.

Parameters **model** – an instance of *pyctrl.system.System*

set(*exclude=()*, ***kwargs*)
Set properties of *pyctrl.block.system.System* block.

Parameters **model** – an instance of *pyctrl.system.System*

reset()
Reset *pyctrl.block.system.System* block.

Calls *pyctrl.system.System.set_output()* for `model` with 0.

write(**values*)
Update `model` and write to the private `buffer`.

Parameters **values** (*vararg*) – values

class pyctrl.block.system.**TimeVaryingSystem**(***kwargs*)
Bases: *pyctrl.block.Filter*, *pyctrl.block.BufferBlock*

`pyctrl.block.system.TimeVarying` is a wrapper for a time-varying dynamic system model.

The first signal must be a clock.

Parameters **model** – an instance of *pyctrl.system.TVSystem*

set(*exclude=()*, ***kwargs*)
Set properties of `pyctrl.block.system.TimeVarying` block.

Parameters **model** – an instance of *pyctrl.system.TVSystem*

reset()
Reset `pyctrl.block.system.TimeVarying` block.

Calls *pyctrl.system.System.set_output()* for `model` with `(0,0)`.

write(**values*)
Update `model` and write to the private `buffer`.

The first signal must be a clock.

Parameters **values** (*vararg*) – values

class pyctrl.block.system.**Gain**(***kwargs*)
Bases: *pyctrl.block.Filter*, *pyctrl.block.BufferBlock*

Gain multiplies input by a constant gain, that is

$y = au$,

where a is the gain.

Parameters **gain** – multiplier (default *1*)

set(*exclude=()*, ***kwargs*)
Set properties of *pyctrl.block.system.Gain* block.

Parameters

- **gain** – multiplier (default *1*)
- **kwargs** (*kwargs*) – other keyword arguments

write(**values*)
Writes product of `gain` times current input to the private `buffer`.

Parameters **values** (*vararg*) – values

class pyctrl.block.system.**Affine**(***kwargs*)
Bases: *pyctrl.block.system.Gain*

Affine multiplies and offset input by a constant gain and offset, that is

$y = au + b$,

where a is the gain and b is the offset.

Parameters

- **gain** (*float*) – multiplier (default *1*)
- **offset** (*float*) – offset (default *0*)

set(*exclude=()*, ***kwargs*)
Set properties of *pyctrl.block.system.Affine* block.

Parameters

- **gain** (*float*) – multiplier (default *1*)
- **offset** (*float*) – offset (default *0*)
- **kwargs** (*kwargs*) – other keyword arguments

write(**values*)
Writes product of `gain` times current input plus `offset` to the private `buffer`.

Parameters **values** (*vararg*) – values

class pyctrl.block.system.**Differentiator**(***kwargs*)
Bases: *pyctrl.block.Filter*, *pyctrl.block.BufferBlock*

Differentiator differentiates the input, that is

$$y = \frac{u_k - u_{k-1}}{t_k - t_{k-1}} \approx \dot{u}.$$

The first signal must be a clock.

write(**values*)
Writes finite difference derivative to the private `buffer`.

The first signal must be a clock.

Parameters **values** (*vararg*) – values

class pyctrl.block.system.**Feedback**(***kwargs*)
Bases: *pyctrl.block.Filter*, *pyctrl.block.BufferBlock*

`pyctrl.system.Feedback` creates a general feedback connection for a given `block`, that is

$$y = Ge, \quad e = \gamma u[m:] + \rho u[:m],$$

where G represents the `block`, γ and ρ are a constant gains, and m is the number of inputs and references.

For the default configuration the first m signals are measurements and the last m signals are references under standard unit negative feedback.

Parameters

- **block** – an instance of *pyctrl.block.Block*
- **gamma** – a constant gain (default *1*)
- **rho** – a constant gain (default *-1*)
- **m** – number of inputs (default *1*)

reset()
Reset *Feedback* block.

Calls *block.reset()*.

write(**values*)
Writes feedback signal to the private `buffer`.

Parameters **values** (*vararg*) – values

class pyctrl.block.system.**Average**(***kwargs*)
Bases: *pyctrl.block.Filter*, *pyctrl.block.BufferBlock*

pyctrl.block.system.Average calculates the (weighted) average of all inputs, that is

$y = \sum_{i=0}^{m-1} w[i]u[i]$,

where w is a vector of weights

Parameters **weights** – weights (default $1/m$)

set(*exclude=()*, ***kwargs*)
Set properties of *Sum* block.

Parameters **weights** – multiplier

write(**values*)
Writes average of the current input to the private `buffer`.

Parameters **values** (*vararg*) – list of values

Returns tuple with scaled input

class pyctrl.block.system.**Sum**(***kwargs*)
Bases: *pyctrl.block.system.Gain*

pyctrl.block.system.Sum adds all inputs and multiplies the result by a constant gain, that is

$y = a \sum_{i=0}^{m-1} u[m]$,

where a is the gain.

Parameters **gain** (*float*) – multiplier (default *1*)

write(**values*)
Writes product of *gain* times the sum of the current input to the private *buffer*.

Parameters **values** (*vararg*) – list of values

Returns tuple with scaled input

class pyctrl.block.system.**Subtract**(***kwargs*)
Bases: *pyctrl.block.system.Gain*

pyctrl.block.system.Subtract subtracts first input as in

$y = a \sum_{i=1}^{m-1} u[m] - u[0],$

where a is the gain.

Parameters **gain** (*float*) – multiplier (default *1*)

write(**values*)
Writes product of *gain* times the difference of the current input to the private *buffer*.

Parameters **values** (*vararg*) – list of values

Returns tuple with scaled input

2.9 Module *pyctrl.block.nl*

class pyctrl.block.nl.**ControlledCombination**(***kwargs*)
Bases: *pyctrl.block.Filter*, *pyctrl.block.BufferBlock*

pyctrl.block.nl.ControlledCombination implements the combination:

$y = \alpha\, u[1 : m+1] + (1-\alpha)\, u[m+1 :], \quad \alpha = \frac{u[0]}{K}$

where K is a gain multiplier.

Parameters

- **gain** (*float*) – multiplier (default *1*)
- **m** (*int*) – number of inputs to combine (default *1*)

set(*exclude=()*, ***kwargs*)
Set properties of *pyctrl.block.nl.ControlledCombination*.

Parameters

- **gain** (*float*) – multiplier (default *1*)
- **m** (*int*) – number of inputs to combine (default *1*)
- **kwargs** (*kwargs*) – other keyword arguments

write(**values*)
Writes combination of inputs to the private `buffer`.

Parameters **values** (*vararg*) – list of values

class pyctrl.block.nl.**Product**(***kwargs*)
Bases: *pyctrl.block.Filter*, *pyctrl.block.BufferBlock*

pyctrl.block.nl.Product implements the product:

$y = u[:m]u[m:]$

Parameters **m** (*int*) – number of inputs to control (default *1*)

set (*exclude=()*, ***kwargs*)
Set properties of *pyctrl.block.nl.Product*.

Parameters

- **m** (*int*) – number of inputs to combine (default *1*)
- **kwargs** (*kwargs*) – other keyword arguments

write (**values*)
Writes product of inputs to the private `buffer`.

Parameters **values** (*vararg*) – list of values

class pyctrl.block.nl.**DeadZone** (***kwargs*)
Bases: *pyctrl.block.Filter*, *pyctrl.block.BufferBlock*

pyctrl.block.nl.DeadZone implements the piecewise function:

$$y = f_{XY}(u) = \begin{cases} au + b, & u > X, \\ au - b, & u < -X, \\ cu, & -X \leq u \leq X \end{cases}$$

where

$$a = \frac{100 - Y}{100 - X}$$
$$b = 100\frac{Y - X}{100 - X}$$
$$c = \frac{Y}{X}$$

This is a generalized dead-zone nonlinearity.

The classic dead-zone nonlinearity has $Y = 0$.

The inverse can be obtained by swapping the arguments, that is $f_{XY}^{-1} = f_{YX}$.

When $X = 0$ then c is `NaN`.

Parameters

- **X** (*float*) – parameter X (default *1*)
- **Y** (*float*) – parameter Y (default *0*)

set (*exclude=()*, ***kwargs*)
Set properties of *pyctrl.block.nl.DeadZone*.

Parameters

- **X** (*float*) – parameter X
- **Y** (*float*) – parameter Y
- **kwargs** (*kwargs*) – other keyword arguments

write (**values*)
Writes the intput transformed by the dead-zone to the private `buffer`.

Parameters **values** (*vararg*) – list of values

2.10 Module *pyctrl.block.logic*

This module provide logic blocks.

class pyctrl.block.logic.**State**
Bases: enum.IntEnum

An enumeration.

class pyctrl.block.logic.**Compare**(***kwargs*)
Bases: *pyctrl.block.Filter*, *pyctrl.block.BufferBlock*

pyctrl.block.logic.Compare compares inputs.

Produces an output with the result of the test

$y[:m] = (u[m:] >= u[:m] + \gamma)$

Output is 1 if test return True and 0 otherwise.

Parameters

- **threshold** (*float*) – the threshold γ (default *0*)
- **m** (*int*) – number of inputs to test (default *1*)

set(*exclude=()*, ***kwargs*)
Set properties of *pyctrl.block.logic.Compare*.

Parameters

- **threshold** (*float*) – threshold
- **m** (*int*) – number of inputs to combine
- **kwargs** (*kwargs*) – other keyword arguments

write(**values*)
Writes result of comparison to the private buffer.

Parameters **values** (*vararg*) – values

class pyctrl.block.logic.**CompareWithHysterisis**(***kwargs*)
Bases: *pyctrl.block.logic.Compare*

pyctrl.block.logic.CompareWithHysterisis compares inputs with histerisis.

Produces an output with the result of the test

$$y[:m] = \begin{cases} (u[m:] >= u[:m] + \gamma + \mu), & \text{if state}[:m] \\ (u[m:] >= u[:m] + \gamma - \mu), & \text{if !state}[:m] \end{cases}$$

and update the state as follows:

$$\text{state}[:m] = \begin{cases} 1, & \text{if state} \,\&\, (u[m:] < u[:m] + \gamma + \mu) \\ 0, & \text{if state} \,\&\, (u[m:] >= u[:m] + \gamma + \mu) \\ 0, & \text{if !state} \,\&\, (u[m:] >= u[:m] + \gamma - \mu) \\ 1, & \text{if !state} \,\&\, (u[m:] < u[:m] + \gamma - \mu) \end{cases}$$

Output is 1 if test return True and 0 otherwise.

Parameters

- **threshold** (*float*) – the threshold γ (default *0*)

- **hysterisis** (*float*) – the hysterisis μ (default *0.1*)
- **m** (*int*) – number of inputs to test (default *1*)

set(*exclude=()*, ***kwargs*)
Set properties of *pyctrl.block.logic.CompareWithHysterisis*.

Parameters

- **threshold** (*float*) – threshold
- **m** (*int*) – number of inputs to combine
- **hysterisis** (*float*) – hysterisis
- **kwargs** (*kwargs*) – other keyword arguments

write(**values*)
Writes result of comparison to the private `buffer`.

Parameters **values** (*vararg*) – values

class pyctrl.block.logic.**CompareAbs**(***kwargs*)
Bases: *pyctrl.block.Filter*, *pyctrl.block.BufferBlock*

pyctrl.block.logic.CompareAbs compares the absolute value of its inputs.

Produces an output with the result of the test

$y[:] = (|u[:]| <= \gamma)$

Output is `1` if test return True and `0` otherwise.

If `invert` is True performs the test:

$y[:] = (|u[:]| >= \gamma)$

Parameters

- **threshold** (*float*) – the threshold γ (default *0.5*)
- **invert** (*bool*) – whether to invert the sign of the test

set(*exclude=()*, ***kwargs*)
Set properties of *pyctrl.block.logic.CompareAbs*.

Parameters

- **threshold** (*float*) – the threshold γ (default *0.5*)
- **invert** (*bool*) – whether to invert the sign of the test
- **kwargs** (*kwargs*) – other keyword arguments

write(**values*)
Writes result of comparison to the private `buffer`.

Parameters **values** (*vararg*) – values

class pyctrl.block.logic.**CompareAbsWithHysterisis**(***kwargs*)
Bases: *pyctrl.block.logic.CompareAbs*

pyctrl.block.logic.CompareAbsWithHysterisis compares the absolute value of its inputs with Hysterisis.

Produces an output with the result of the test:

$$y[:m] = \begin{cases} (|u[:] - \alpha| <= \gamma - \mu), & \text{if state[:]} \\ (|u[:] - \alpha| <= \gamma + \mu), & \text{if !state[:]} \end{cases}$$

and update the state as follows:

$$\text{state}[:m] = \begin{cases} 1, & \text{if state } \& \ (u[:] < \gamma + \mu) \\ 0, & \text{if state } \& \ (u[:] >= \gamma + \mu) \\ 0, & \text{if !state } \& \ (u[:] >= \gamma - \mu) \\ 1, & \text{if !state } \& \ (u[:] < \gamma - \mu) \end{cases}$$

Output is `1` if test return True and `0` otherwise.

If `invert` is True performs the test:

$y[:] = (|u[:]| >= \gamma)$

Parameters

- **threshold** (*float*) – the threshold γ (default *0.5*)
- **offset** (*float*) – the offset α (default *0.0*)
- **hysterisis** (*float*) – the hysterisis μ (default *0.1*)
- **invert** (*bool*) – whether to invert the sign of the test

set (*exclude=()*, ***kwargs*)
Set properties of `pyctrl.block.logic.CompareAbsWithHisterisis`.

Parameters

- **threshold** (*float*) – the threshold γ (default *0.5*)
- **hysterisis** (*float*) – the hysterisis μ (default *0.1*)
- **invert** (*bool*) – whether to invert the sign of the test
- **kwargs** (*kwargs*) – other keyword arguments

write (**values*)
Writes result of comparison to the private `buffer`.

Parameters **values** (*vararg*) – values

class pyctrl.block.logic.**Trigger**(***kwargs*)
Bases: *pyctrl.block.Filter*, *pyctrl.block.BufferBlock*

pyctrl.block.logic.Trigger can be used to switch signals on or off.

Produces the output

$y[:-1] = \gamma\, u[1:]$

where $\gamma = 1$ or $\gamma = 0$.

The value of γ depends on the attribute `state` as follows

$$\gamma = \begin{cases} 1, & \text{if state } \& \ f(u[0]) \\ 0, & \text{if !state} \end{cases}$$

Once $f(u[0])$ becomes True `state` is set to True and must be reset manually.

Parameters

- **function** – test function (default identity)
- **state** (*bool*) – initial state (default State.LOW)

reset()
Reset *pyctrl.block.logic.Trigger* attribute `state` to False.

write(**values*)
Writes result of trigger to the private `buffer`.

Parameters **values** (*vararg*) – values

class pyctrl.block.logic.**Event**(***kwargs*)
Bases: *pyctrl.block.Sink*, *pyctrl.block.BufferBlock*

pyctrl.block.logic.Event runs actions based on event.

Produces no output but calls `pyctrl.block.logic.Event.rise_event()` if

`state` is `pyctrl.block.logic.State.LOW` and $u[m:] >= high$

or `pyctrl.block.logic.Event.fall_event()` if

`state` is `pyctrl.block.logic.State.HIGH` and $u[m:] <= low$

Parameters

- **low** (*float*) – the low threshold
- **high** (*float*) – the high threshold

set(*exclude=()*, ***kwargs*)
Set properties of *pyctrl.block.logic.Event*.

Parameters

- **low** (*float*) – low threshold
- **high** (*float*) – high threshold
- **state** – state (HIGH or LOW)
- **kwargs** (*kwargs*) – other keyword arguments

write(**values*)
Writes result of comparison to the private `buffer`.

Parameters **values** (*vararg*) – values

class pyctrl.block.logic.**SetBlock**(***kwargs*)
Bases: *pyctrl.block.logic.Event*

pyctrl.block.logic.SetBlock set block based on event.

set(*exclude=()*, ***kwargs*)
Set properties of *pyctrl.block.logic.Event*.

Parameters

- **blocktype** (*pyctrl.BlockType*) – block type (source, sink, filter, timer)
- **kwargs** (*kwargs*) – other keyword arguments

2.11 Module *pyctrl.block.random*

class pyctrl.block.random.**Uniform**(***kwargs*)
Bases: *pyctrl.block.Source*, *pyctrl.block.BufferBlock*

pyctrl.block.random.Uniform produces an output with random entries uniformly distributed between `low` and `high`.

pyctrl.block.random.Uniform uses `numpy.random.uniform()`.

Parameters

- **low** (*float*) – lowest value (default *0*)
- **high** (*float*) – highest value (default *1*)
- **m** (*int*) – number of output (default *1*)
- **seed** – seed (default None)

reset()
Resets *pyctrl.block.random.Uniform* by reseeding generator.

set(*exclude=()*, ***kwargs*)
Set properties of *pyctrl.block.random.Uniform*.

Parameters

- **low** (*float*) – lowest value
- **high** (*float*) – highest value
- **m** (*int*) – number of outputs
- **seed** – seed

read()
Writes random numbers to private `buffer`.

class pyctrl.block.random.**Gaussian**(***kwargs*)
Bases: *pyctrl.block.Source*, *pyctrl.block.BufferBlock*

pyctrl.block.random.Gaussian produces an output with random entries normally distributed with parameters `mu` and `sigma`.

pyctrl.block.random.Gaussian uses `numpy.random.normal()`.

Parameters

- **mu** (*float*) – mean (default *0*)
- **sigma** (*float*) – variance (default *1*)
- **m** (*int*) – number of outputs (default *1*)
- **seed** – seed (default None)

reset()
Resets *pyctrl.block.random.Gaussian* by reseeding generator.

set(*exclude=()*, ***kwargs*)
Set properties of *pyctrl.block.random.Gaussian*.

Parameters

- **mu** (*float*) – mean
- **sigma** (*float*) – variance
- **m** (*int*) – number of outputs
- **seed** – seed

read()
Writes random numbers to private `buffer`.

2.12 Module *pyctrl.rc*

This module supports the hardware on the Robotics Cape running on a Beaglebone Black or a Beaglebone Blue. It also provides a controller suited for the configuration of the Educational MIP (Mobile Inverted Pendulum) kit. For installation instructions see Section *Installation*.

class pyctrl.rc.**Controller**(**kwargs*)

Bases *pyctrl.Controller*

pyctrl.rc.Controller initializes a controller for the Robotics Cape equiped with a clock based on the MPU9250 periodic interrupts.

Parameters

- **period** (*float*) – period in seconds (default 0.01)
- **kwargs** – other keyword parameters

2.13 Module *pyctrl.rc.encoder*

class pyctrl.rc.encoder.**Encoder**(**kwargs*)

Bases *pyctrl.block.Block*

pyctrl.rc.encoder.Encoder provides an interface to the Beaglebone Black and Beaglebone Blue 3 hardware encoder counters.

Thanks to the Robotics Cape library a fourth encoder counter is available through one of the PRUs.

The attribute `ratio` can be used to set to the desired reading units.

Parameters

- **ratio** (*float*) – multiplier (default 48 * 172)
- **encoder** (*int*) – encoder channel, 1 through 4 (default 2)
- **kwargs** – other keyword parameters

reset()
Reset *pyctrl.rc.encoder.Encoder*.

This function writes `0` to the current encoder count register.

write(*values)
Write to Encoder.

Set encoder count register to `value[0]` multiplied by `ratio`.

Parameters **values** (*varag*) – values

read()
Read from Encoder.

When encoder is enabled, returns the current encoder count register divided by `ratio`.

2.14 Module *pyctrl.rc.motor*

class pyctrl.rc.motor.**Motor**(**kwargs*)

Bases *pyctrl.block.Block*

pyctrl.rc.motor.Motor provides an interface to the Robotics Cape 4 hardware PWM motor channels.

With the default `ratio`, PWM values are from -100 to 100 but can be changed by modifying the attribute `ratio`.

Parameters

- **ratio** (*float*) – ratio factor by which PMW is divided (default 100)
- **motor** (*int*) – motor channel, 1 through 4 (default 2)
- **kwargs** – other keyword parameters

set_enabled(*enabled = True*)
Set *enabled* state.

This function writes `0` to the corresponding PWM channel when `enabled` is `False`.

Parameters **enabled** – `True` or `False` (default `True`)

write(**values*)
Write to Motor.

When motor is enabled, sets motor PWM `value[0]` divided by `ratio`.

Parameters **values** (*varag*) – values

2.15 Module *pyctrl.rc.mpu9250*

class pyctrl.rc.mpu9250.**MPU9250**(**kwargs*)

Bases *pyctrl.block.clock.Clock*

pyctrl.rc.mpu9250.MPU9250 provides an interface to the Robotics Cape MPU9250 IMU.

It inherits from *pyctrl.block.clock.Clock* and can be used as a clock.

Parameters

- **accel_fsr** (*int*) – accelerometer full scale resolution
- **gyro_fsr** (*int*) – gyroscope full scale resolution
- **accel_dlpf** (*int*) – acellerometer low pass cutoff filter
- **gyro_dlpf** (*int*) – gyroscope low pass cutoff filter
- **orientation** (*int*) – orientation
- **compass_time_constant** (*float*) – compass time constant
- **dmp_interrupt_priority** (*int*) – DMP interrupt priority
- **period** (*int*) – DMP period in seconds
- **enable_magnetometer** (*bool*) – enable magnetometer
- **enable_dmp** (*bool*) – enable Digital Motion Processor
- **enable_fusion** (*bool*) – enable fusion algorithm

- **show_warnings** (*bool*) – show warnings
- **kwargs** – other keyword parameters

get_data():
Return the data read by the IMU.

Returns the IMU data

read()
Read MPU9250.

WARNING: This function does not return any IMU data, just a timestamp. Use `get_data()` to get IMU data.

Returns time

class pyctrl.rc.mpu9250.**Raw**(**kwargs)

Bases *pyctrl.block.BufferBlock*

pyctrl.rc.mpu9250.Raw reads raw accelerometer and gyroscope angular velocity from the MPU9250 IMU.

read()
Read from IMU.

When enabled, returns the current IMU linear acceleration and gyroscope angular velocities.

Returns tuple with IMU acceleration and gyroscope angular velocities.

class pyctrl.rc.mpu9250.**Inclinometer**(**kwargs)

Bases *pyctrl.rc.mpu9250.Raw*

pyctrl.rc.mpu9250.Inclinometer reads the output from *pyctrl.rc.mpu9250.Inclinometer* and returns the angle and angualar velocity with respect to the vertical.

Parameters

- **turns** (*int*) – current number of turns (default 0)
- **threshold** (*float*) – used to detect that the IMU has been fliped more than 1 turn (360 degrees) (default 0.25 turns (90 deg))

read()
Read inclinometer.

When enabled, returns the current angle and angular velocity measured with respect to the vertical.

Returns tuple with angle and angular velocity.

2.16 Module *pyctrl.rc.mip*

class pyctrl.rc.mip.**Controller**(**kwargs)

Bases *pyctrl.rc.Controller*

pyctrl.rc.Controller initializes a controller to be used with the Educational MIP connected to a Robotics Cape.

The following devices are installed:

1. Clock, `clock`, based on interrupts generated by the MPU9250; installed as a *source*; output is the `clock`;

2. Inclinometer, `inclinometer`, based on the MPU9250 IMU; installed as a *source*; outputs are the *signals* `theta` and `theta_dot`, which correspond to the angular position and velocity of the MIP; units are cycles and cycles/s;
3. Two encoders, `encoder1`, and `encoder2`, installed as *sources*; outputs are the *signals* `encoder1` and `encoder2`, which correspond to relative angular displacement between the body of MIP and the axis of the left and right motors; units are cycles;
4. Two motors, `motor1`, and `motor2`; installed as *sinks*; inputs are the PWM signals `pwm1` and *py:data:'pwm2* driving the left and right motors of the MIP; range is from -100 to 100.

Parameters

- **period** (*float*) – period in seconds (default 0.01)
- **kwargs** – other keyword parameters

2.17 Module *pyctrl.system*

This module provides objects that can be used to represent time-invariant and time-varying dynamic systems.

class pyctrl.system.**System**

Bases: `object`

Base class for dynamic time-invariant systems.

set_output(*yk*)

Sets the internal state of the *pyctrl.system.System* so that a call to *pyctrl.system.System.update()* with `uk = 0` yields `yk`.

Parameters **yk** – scalar desired `yk`

shape()

Shape of a *pyctrl.system.System*

Returns tuple with number of inputs, number of outputs and order

update(*uk*)

Time update *pyctrl.system.System* model.

Parameters **uk** – the input `uk`

Returns the output of the model

class pyctrl.system.**TVSystem**

Bases: *pyctrl.system.System*

Base class for dynamic time-varying systems.

set_output(*tk*, *yk*)

Sets the internal state of the *pyctrl.system.TVSystem* so that a call to *pyctrl.system.System.update()* with `uk = 0` and `tk` yields `yk`.

Parameters **yk** – scalar desired `yk`

shape()

Shape of a *pyctrl.system.System*

Returns tuple with number of inputs, number of outputs and order

update(*tk*, *uk*)

Time update *pyctrl.system.System* model.

Parameters

- **tk** – the time `tk`
- **uk** – the input `uk`

Returns the output of the model

exception pyctrl.system.**SystemException**
Bases: `Exception`

Exception class for module *pyctrl.system*.

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

2.18 Module *pyctrl.system.tf*

class pyctrl.system.tf.**DTTF**(*num=array([1])*, *den=array([1])*, *state=None*)
Bases: *pyctrl.system.System*

`pyctrl.system.DTTF` implements a single-input-single-output (SISO) transfer-function.

The underlying model is of the form:

$$den[0]y_k + den[1]y_{k-1} + \cdots + den[n]y_{k-n} = num[0]uk + num[1]u_{k-1} + \cdots + num[m]u_{k-m}$$

which corresponds to the transfer-function:

$$G(z) = \frac{\sum_{i=0}^{m} z^{-i} num[i]}{\sum_{i=0}^{n} z^{-i} den[i]}$$

Denominator is always normalized so that $den[0] = 1$.

Model is implementated in terms of the auxiliary variable z as follows. Let

$$z_k + den[1]z_{k-1} + \cdots + den[n]z_{k-n} = u_k$$

By linearity:

$$y_k = num[0]z_k + num[1]z_{k-1} + \cdots + den[n]z_{k-n}$$

Parameters

- **num** – numpy m-dimensional 1D-vector numerator (default [1])
- **den** – numpy n-dimensional 1D-vector denominator (default [1])
- **state** – numpy n-dimensional 1D-vector representing vector z (default *None*)

set_output(*yk*)
Sets the internal state of the `pyctrl.system.DTTF` so that a call to *update* with *uk = 0* yields *yk*.

It is calculated as follows. With $u_k = 0$

$$z_k + den[1]z_{k-1} + \cdots + den[n]z_{k-n} = 0$$

and

$$\begin{aligned} y_k &= num[0]z_k + num[1]z_{k-1} + \cdots + num[n]z_{k-n} \\ &= num[1]z_{k-1} + \cdots + num[n]z_{k-n} - num[0](den[1]z_{k-1} + \cdots + den[n]z_{k-n}) \end{aligned}$$

and $y_k = yk$ if $num[1] \neq num[0]den[1]$ and

$$z_{k-1} = \frac{y_k - \sum_{i=2}^{n}(num[i] - num[0]den[i])z_{k-i}}{num[1] - num[0]den[1]}$$

TODO: if $num[1] \neq num[0]den[1]$ then choose next nonzero coefficient.

Parameters **yk** – scalar desired *yk*

update(*uk*)

Update `pyctrl.system.DTTF` model. Implements the recursion:

$$z_k + den[1]z_{k-1} + \cdots + den[n]z_{k-n} = u_k$$
$$y_k = num[0]z_k + num[1]z_{k-1} + \cdots + den[n]z_{k-n}$$

Parameters **uk** (*numpy.array*) – input at time k

as_DTSS()

Returns a state-space representation (`pyctrl.system.DTSS`) of the `pyctrl.system.DTTF`.

`pyctrl.system.tf.`**zDTTF**(*num*, *den*, *state=None*)

`pyctrl.system.zDTTF` implements a single-input-single-output (SISO) transfer-function.

The underlying model is of the form corresponds to the transfer-function:

$$G(z) = \frac{\sum_{i=0}^{m} z^i num[i]}{\sum_{i=0}^{n} z^i den[i]}$$

This is a convinience constructor that transforms a model in the form *zDTTF* into a *pyctrl.system.DTTF*

class `pyctrl.system.tf.`**PID**(*Kp*, *Ki=0*, *Kd=0*, *period=0*, *state=None*)

Bases: *pyctrl.system.tf.DTTF*

`pyctrl.system.PID` implements a Proportional-Integral-Derivative (PID) controller.

A continuous-time PID implements the following function:

$$u(t) = K_p e(t) + K_i \int_0^t e(t)\,dt + K_d \frac{d}{dt} e(t)$$

We provice the following discrete-time version:

$$x_k = x_{k-1} + \frac{T}{2}(e_k + e_{k-1})$$
$$u_k = K_p e_k + K_i x_k + \frac{K_d}{T}(e_k - e_{k-1})$$

In terms of a transfer-function

$$X(z) = \frac{T}{2}\frac{1 + z^{-1}}{1 - z^{-1}} E(z)$$

so that

$$\begin{aligned} U(z) &= KpE(z) + K_i X(z) + \frac{K_d}{T}(1 - z^{-1})E(z) \\ &= \left(K_p + \frac{K_i T}{2}\frac{1 + z^{-1}}{1 - z^{-1}} + \frac{K_d}{T}(1 - z^{-1})\right) E(z) \\ &= \frac{K_p(1 - z^{-1}) + \frac{K_i T}{2}(1 + z^{-1}) + \frac{K_d}{T}(1 - z^{-1})^2}{1 - z^{-1}} E(z) \\ &= G(z)E(z) \end{aligned}$$

where

$$G(z) = \frac{K_p + \frac{K_i T}{2} + \frac{Kd}{T} + z^{-1}\left(\frac{K_i T}{2} - K_p - 2\frac{K_d}{T}\right) + z^{-2}\left(\frac{K_d}{T}\right)}{1 - z^{-1}}$$

A PD controller is the special case when Ki = 0:

$$\begin{aligned} U(z) &= \left(K_p + \frac{Kd}{T}(1 - z^{-1})\right) E(z) \\ &= \frac{K_p + + \frac{Kd}{T} - z^{-1}\frac{Kd}{T}}{1} E(z) \end{aligned}$$

This is a convinience constructor that returs a *PID* as a *pyctrl.systems.DTTF*.

Parameters

- **Kp** – proportional gain
- **Ki** – integral gain (default = *0*)
- **Kd** – derivative gain (default = *0*)
- **period** – sampling period (default = *0*)
- **state** – internal state representation (default = *None*)

class pyctrl.system.tf.**LPF**(*fc*, *period*, *gain=1*, *order=1*, *state=None*)

Bases: *pyctrl.system.tf.DTTF*

`pyctrl.system.LPF` implements a low pass-filter based on `pyctrl.system.DTTF`.

The first-order filter is a discretized version os the continuous-time filter with transfer-function:

$$T(s) = \frac{K\omega_c}{s + \omega_c}, \quad \omega_c = 2\pi f_c$$

where f_c is the cutoff frequency and K is the filter gain. The zero-order hold equivalent transfer-function is:

$$T(z) = \frac{K(1-a)}{z - a}, \quad a = e^{-\omega_c T_s}$$

where T_s is the sampling period.

TODO: filters of order higher than *1*

Parameters

- **fc** – cuttof frequency in Hz
- **Ts** – sampling period in seconds
- **gain** – gain (default = *1*)
- **order** – order (default = *1*)
- **state** – initial state (default = *None*)

2.19 Module *pyctrl.system.ss*

class pyctrl.system.ss.**DTSS**(*A=array([], shape=(1, 0), dtype=float64), B=array([[0]]), C=array([], shape=(1, 0), dtype=float64), D=array([[1]]), state=None*)

Bases: *pyctrl.system.System*

DTSS implements a discrete-time state-space model of the form:

$$x_{k+1} = Ax_k + Bu_k$$
$$y_k = Cx_k + Du_k$$

Parameters

- **A** (*numpy.array*) – state space matrix A (Default = [])
- **B** (*numpy.array*) – state space matrix B (Default = [[0]])
- **C** (*numpy.array*) – state space matrix C (Default = [])
- **D** (*numpy.array*) – state space matrix D (Default = [[1]])
- **state** (*numpy.array*) – initial value of the state vector

set_output (*yk*)

Sets the internal state of the `pyctrl.system.DTSS` so that a call to *update* with *uk = 0* yields *yk*.

It is calculated as follows:

$$x_k = C^\dagger y_k$$

where $C^\dagger$ is the pseudo-inverse of C, which leads to

$$y_k = Cx_k = y_k$$

when $u_k = 0$

Parameters **yk** (*numpy.array*) – desired *yk*

update (*uk*)

Update `pyctrl.system.SSTF` model. Calculates:

$$y_k = Cx_k + Du_k$$

then updates

$$x_{k+1} = Ax_k + Bu_k$$

Parameters **uk** (*numpy.array*) – input at time k

2.20 Module *pyctrl.system.ode*

class pyctrl.system.ode.**ODEBase**(*shape*, *f*, *g=<function identity>*, *x0=array([0])*, *t0=-1*, *pars=()*)

Bases: *pyctrl.system.TVSystem*

`pyctrl.system.ODEBase` implements a general nonlinear time-varying continuous-time state-space model of the form:

$$\dot{x} = f(t, x, u, *pars)$$
$$y = g(t, x, u, *pars)$$

This base class does not implement any methods.

Parameters

- **shape** (*tuple*) – (m,p) where m is the number of inputs and p is the number of outputs

- **f** – nonlinear state function f
- **g** – nonlinear state function g
- **x0** (*numpy.array*) – initial value of the state vector
- **t0** (*float*) – initial time
- **pars** (*vargs*) – variable positional arguments to be passed to f and g

set_output(*tk*, *yk*)

Sets the internal state of the `pyctrl.system.ODE` so that a call to *update* with *uk = 0* yields *yk*.

Solves the equation:

$$g(x_k) = y_k$$

which leads to

$$y_k = g(x_k) = y_k$$

when $u_k = 0$

Parameters **yk** (*numpy.array*) – desired *yk*

class pyctrl.system.ode.**ODE**(*shape*, *f*, *g=<function identity>*, *x0=0*, *t0=-1*, *pars=()*)

Bases: *pyctrl.system.ode.ODEBase*

`pyctrl.system.ODE` implements a general nonlinear time-varying continuous-time state-space model of the form:

$$\dot{x} = f(t, x, u, *pars)$$
$$y = g(t, x, u, *pars)$$

Integration is performed using `scipy.integrate.ode`.

Parameters

- **shape** (*tuple*) – (m,p) where m is the number of inputs and p is the number of outputs
- **f** – nonlinear state function f
- **g** – nonlinear state function g
- **x0** (*numpy.array*) – initial value of the state vector
- **t0** (*float*) – initial time
- **pars** (*vargs*) – variable positional arguments to be passed to f and g

set_output(*tk*, *yk*)

Sets the internal state of the `pyctrl.system.ODE` so that a call to *update* with *uk = 0* yields *yk*.

Solves the equation:

$$g(x_k) = y_k$$

which leads to

$$y_k = g(x_k) = y_k$$

when $u_k = 0$

Parameters **yk** (*numpy.array*) – desired *yk*

class pyctrl.system.ode.**ODEINT**(*shape*, *f*, *g=<function identity>*, *x0=0*, *t0=-1*, *pars=()*)

Bases: *pyctrl.system.ode.ODEBase*

pyctrl.system.ODEINT implements a general nonlinear time-varying continuous-time state-space model of the form:

$$\dot{x} = f(t, x, u, *pars)$$
$$y = g(t, x, u, *pars)$$

Integration is performed using scipy.integrate.odeint.

Parameters

- **shape** (*tuple*) – (m,p) where m is the number of inputs and p is the number of outputs
- **f** – nonlinear state function f
- **g** – nonlinear state function g
- **x0** (*numpy.array*) – initial value of the state vector
- **t0** (*float*) – initial time
- **pars** (*vargs*) – variable positional arguments to be passed to f and g

set_output(*tk*, *yk*)

Sets the internal state of the pyctrl.system.ODE so that a call to *update* with *uk = 0* yields *yk*.

Solves the equation:

$$g(x_k) = y_k$$

which leads to

$$y_k = g(x_k) = y_k$$

when $u_k = 0$

Parameters **yk** (*numpy.array*) – desired *yk*

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search

BIBLIOGRAPHY

[deO16] M. C. de Oliveira, *Fundamentals of Linear Control: a concise approach*, Cambridge University Press, 2016.

[Zhuo16] Zhu Zhuo, *LQG Controller Design of the Mobile Inverted Pendulum*, M.Sc. Thesis, Department of Mechanical and Aerospace Engineering, University of California San Diego, 2016.

PYTHON MODULE INDEX

p

INDEX

H

I

J

L

M

O

P

R

S

T

U

V

W

Z