micropythonlibs

Brian Pugh

Apr 23, 2023

CONTENTS:

CHAPTER

ONE

MICROPYTHON-LIBS

This repository contains a collection of single-file libraries intended for a micropython target.

All micropython libraries are located in the `lib/` folder of this repo.

CHAPTER

TWO

GENERIC LIBRARIES

These libraries generally don't depend on specific hardware and primarily contain just software algorithms/abstractions.

- `configstore` - Persistent auto-write key-value store.
- `controlhal` - Abstractions for controlling a dynamic system. Easy PID control loops.
- `debouncedpin` - Debounced `Pin` drop-in that automatically handles switch debouncing. Can also simultaneously drive an LED using the same pin with `DebouncedLedPin`.
- `interp1d` - One dimensional interpolation functions.
- `oversample` - Oversample a sensor to improve the SNR and measurement resolution at the cost of increased CPU utilization and reduced throughput.
- `pid` - PID controller. Not recommended for fast processes (not for quadcopters). Requires: `controlhal`.
- `pidautotune` - Autotune for PID controllers. Requires `controlhal`.
- `ringbuffer` - RingBuffer with builtin statistical methods.
- `uprofiler` - Code profiling tools.

CHAPTER THREE

HARDWARE DRIVERS

These libraries contain drivers for specific hardware. Whenever possible, these drivers abide by the standards in docs/hardware.

- `max6675` - MAX6675 Cold-Junction-Compensated K-Thermocouple-to-Digital Converter (0°C to +1024°C).

CHAPTER

FOUR

INSTALLATION

We recommend using the Belay Package Manager for installing desired libraries. To install Belay on your computer, run:

```
pip install belay
```

Then, define your project name and dependencies in your project's `pyproject.toml` file. Belay assumes you have a python package in your project with the same name as `tool.belay.name`:

```
[tool.belay]
name = "my_project_name"

[tool.belay.dependencies]
ringbuffer = "https://github.com/BrianPugh/micropython-libs/blob/main/lib/ringbuffer.py"

[tool.pytest.ini_options]
pythonpath = ".belay/dependencies/main"
```

Then, to actually download the dependencies (and update them if already downloaded), run the following in your project's root directory:

```
belay update
```

Finally, to actually get the code onto your device, run:

```
belay install [DEVICE-PORT]
```

You can specify other argument to `belay install`, including cross-compiling the python code.

CHAPTER

FIVE

REPO FOLDER STRUCTURE

- `lib/` - Micropython modules.
- `tests/` - Tests for micropython modules
- `demos/` - For code that primarily interacts with hardware, these serve as minimal scripts for demonstrating their use.
- `tools/` - Auxiliary scripts that aid in project maintenance or experiments to drive design decisions.

5.1 Installation

These libraries are intended for a micropython target. We suggest using the Belay Package Manager

Install Belay:

```
pip install belay
```

Then, in your project's `pyproject.toml`, add a section like:

```
[tool.belay.dependencies]
interp1d = "https://github.com/BrianPugh/micropython-libs/blob/main/lib/interp1d.py"
```

Next, run `belay update` to pull down the latest changes.

Finally, to transfer the libraries to your device:

```
belay install <PORT>
```

While developing, it can be useful then to also run a file after innstalling your project:

```
belay install <PORT> --run my_script.py
```

5.2 ConfigStore

Persistent key-value store with automatic write-to-disk and schema enforcement.

5.2.1 Dependencies

No dependencies.

5.2.2 ConfigStore

In most cases, the resulting `ConfigStore` object can just be treated exactly like a dictionary.

```python
from configstore import ConfigStore

config = ConfigStore("settings.json")  # Will read/write to "settings.json"
config["foo1"] = "bar1"  # Simple key/value store. Write is automatically performed.
config["list_example"] = [1, 2, 3]  # lists work
config["nested"] = {"foo2": "bar2"}  # So do nested dictionaries.

config_reload = ConfigStore.load("settings.json")  # Load an existing config
assert config_reload == config  # They should be the same.

with config:  # defer writes until the contextmanager exits.
    config["a"] = 1
    config["b"] = 2
    config["c"] = 3
```

Supplying Dictionary Defaults

Like a dictionary, the `config` object has an `update` method to merge in a dictionary. The `update` method will overwrite existing keys:

```python
config["foo"] = "bar"
config["value"] = 123
config.update({"name": "alice", "value": 456})

assert config["foo"] == "bar"
assert config["value"] == 456
assert config["name"] == "alice"
```

To supply default values, use the `merge` method. Existing values will not be overwritten.

```python
config["foo"] = "bar"
config["value"] = 123
config.merge({"name": "alice", "value": 456})

assert config["foo"] == "bar"
assert config["value"] == 123
assert config["name"] == "alice"
```

Freezing Schema

Often, once configured, the json schema should be fixed to prevent accidental misconfigurations. This includes things like typoing keys, or assigning incorrect value datatypes. Call the `freeze_schema` to freeze the `ConfigStore` object.

```
config["foo"] = "bar"
config["my_list"] = [1, 2, 3]
config.freeze_schema()
```

For dictionaries, attempting to assign a different datatype to a key will result in a `FrozenError` exception.

```
config["foo"] = 123  # raises FrozenError
```

For lists, values can no longer be appended to (list will have fixed length). When replacing an element, the new element must have the same datatype.

```
config["my_list"].append(4)  # raises FrozenError
config["my_list"][0] = "foo"  # raises FrozenError
config["my_list"][1] = 42  # this is OK
```

Freezing only specific children types is possible:

```
config.freeze_schema(dicts=True, lists=True, recursive=True)  # Default values
```

5.3 ControlHAL

Control Hardware Abstraction Layer.

This library provides inheritable base classes that simplifies reading inputs (sensors) and controlling outputs (actuators). The interfaces to these classes relies heavily on both inheritance and composition so that peripherals can be easily swapped and combined. Sensor reads and actuator writes are self-caching and self-limiting, meaning they can be efficiently in quick succession without worrying about physical implications. This allows code to be very loosely coupled, for example:

```
# Conventional: need to pass around a cached value.
while True:
    temperature = read_temperature()
    print(f"temperature: {temperature}°C")
    if temperature > 100:
        print("Boiling")
    sleep(0.5)

# ControlHAL: read sensor as much as your want.
# Rapid repeated reads will return a cached value.
while True:
    print(f"temperature: {sensor()}°C")
    if sensor() > 100:
        print("Boiling")
```

See the source files for more details on function and class APIs. Anytime that a `machine.Pin` is referenced, a `machine.Signal` or `signal.Signal` may be a more appropriate choice.

5.3.1 Dependencies

No Dependencies

Optional

- `ringbuffer` - Only used in the `Derivative` virtual sensor.

5.3.2 Protocol Summary

The methods for each class in ControlHAL is summarizes in the table below:

	Sensor	Actuator	Controller	ControlLoop
`__call_`	Read sensor (SI).	Read setpoint (%).	N/A	Predict actuator % from sensor (SI).
`__call_`	`NotImplementedError`	Write setpoint & device (%).	Predict actuator value (%) from sensor(s) value (SI).	Args/Kwargs are passed to `controller.__call__`.
`read()`	Read sensor (SI).	Read setpoint (%).	Read setpoint (SI).	Read sensor (SI).
`write(v`	`NotImplementedError`	Write setpoint (%) & device.	Write setpoint (SI).	Write setpoint to controller (SI).
`estop()`	Prevents future reads. Subsequent reads return last cached value.	Writes `0.0` to setpoint & device. Disables future writes.	No effect.	Propagate `estop` call to all attributes and controllers.
get `setpoir`	Always returns 0.	Get setpoint (%).	Get setpoint (SI).	Get controller's setpoint (SI).
set `setpoir`	`NotImplementedError`	Set setpoint (%).	Set setpoint (SI).	Set controller's setpoint (SI).

In this table:

- SI - sane metric values appropriate for the sensor.
- % - A floating point value ranging from [0, 1] representing 0% ~ 100%.

All classes described inherit from the `Peripheral` base class.

5.3.3 Sensor

Abstract input sensor class.

Input devices should inherit from `Sensor` and implement the `_raw_read` method. Optionally the `_convert` may also be implemented. The default `_convert` method is an identity operation.

```python
def _raw_read(self) -> float:
    """Read sensor.

    The returned value **may** be in standard units; or may be in a fast
    intermediate format for ``self._convert`` to post process into standard
    units. It is recommended to put fast, sensor reading logic into
```

(continues on next page)

(continued from previous page)

```
    ``_raw_read``, and put expensive deferred logic into ``_convert``.
    This way, the sensor can be oversampled with minimal overhead.

    Returns
    -------
    float
    """


def _convert(self, val: float) -> float:
    """Convert raw-value from ``_raw_read`` to a SI base unit float.

    Used to only call conversion once per oversample, rather than once per sample.

    Reference:
        https://en.wikipedia.org/wiki/SI_base_unit

    Parameters
    ----------
    val : float

    Returns
    -------
    float
    """
```

Sensor can be oversampled by providing an integer value `samples` to `__init__`. Defaults to 1 sample per read (i.e. no oversampling).

ADCSensor

Sensor using an ADC input.

```python
from controlhal import ADCSensor
from machine import ADC

sensor = ADCSensor(ADC(0))
```

Derivative

A virtual sensor that acts as the time-derivative of another sensor.

```python
from machine import ADC
from controlhal import Derivative

position_sensor = ADCSensor(ADC(0))
velocity_sensor = Derivative(position_sensor)
velocity = velocity_sensor.read()
```

Internally uses the five-point stencil to compute the derivative over a series of input measurements. The returned derivative will be `0` until the internal buffer of length 5 fills up.

5.3.4 Actuator

Abstract output actuator class.

Output devices should inherit from `Actuator` and implement the `_raw_write` method.

```
def _raw_write(self, val: float):
    """Perform actual write ``val`` to actuator.

    Parameters
    ----------
    val : float
        Value to write in range ``[0., 1.]``.
    """
```

Attempting to read from an actuator will return the current `setpoint` in range `[0., 1.]`. This value is also available via the read-only `setpoint` attribute.

TimeProportionalActuator

Varies an output actuator via pulse-width-modulation.

Uses an internal virtual timer and intended for relatively slow processes like controlling a heater (period > 1 second).

```
from controlhal import TimeproportionalActuator

heater = TimeProportionalActuator(Pin(1, Pin.OUT), period=10.0)
heater.write(0.75)  # Heater will be on for 7.5 seconds, then off for 2.5 seconds.
```

PWMActuator

Varies an output actuator via pulse-width-modulation.

Similar to a `TimeProportionalActuator`, but requires a supplied configured `PWM` object. Intended for more rapid output devices, like LEDs or motors.

```
from controlhal import PWMActuator
from machine import Pin, PWM

pwm = PWM(Pin(12))
pwm.freq(500)  # Set frequency to 500Hz
actuator = PWMActuator(pwm)  # The PWMActuator class will handle setting duty-cycle
```

Multi

Collect a set of peripherals into a single class. Can be used for more complex controllers while re-using other classes in this library.

```
from controlhal import Multi

multi_sensor = Multi(sensor1, sensor2, sensor3)
# Will read and return all 3 sensors
sensor1_val, sensor2_val, sensor3_val = multi_sensor.read()
```

Multi can be subclassed to provide more structure/order to the constructor:

```python
class MotorSensor(MultiSensor):
    def __init__(self, position, current, temperature):
        super().__init__(position, current, temperature)
```

5.3.5 Controller

Abstract base class for predictive models that consume sensor data and produce actuator predictions.

At the very least, needs to implement the following methods:

```python
class MyController(Controller):
    @property
    def parameters(self) -> Any:
        """Internal parameters that a controller can be constructed from.

        e.g. for a PID controller, this would be ``(k_p, k_i, k_d)``
        """

    def __call__(self, *args, **kwargs) -> float:
        """Given some sensor input, predict what the actuator value
        should be to drive the system to ``setpoint``.
        """
```

The controller setpoint can be written to either by directly writing to `controller.setpoint` or by calling the `controller.write(val)` method.

For a more indepth example, see the `pid` library for a PID controller.

5.3.6 ControlLoop

A self-contained control loop system for single-input/single-output systems. For example, controlling a heating element based on feedback from a temperature sensor. The example below uses the `pid` library.

```python
from controlhal import ADCSensor, ControlLoop
from machine import ADC, Pin
from pid import PID
from time import sleep

temperature_sensor = ADCSensor(
    ADC(0), 100 / 65535
)  # Hypothetical analog sensor [0, 100] °C
heater = TimeProportionalActuator(Pin(1, Pin.OUT))
pid = PID(0.05, 0.0001)

control_loop = ControlLoop(heater, temperature_sensor, pid)

while True:
    control_loop()  # Reads sensor, invokes controller, and updates actuator.
    sleep(0.25)
```

5.4 DebouncedPin

Automatic input button debouncing.

Buttons and other mechanical switches often generate rapid open/close signals when actuated. Naively, a microcontroller may read these rapid open/closing signals as actual multiple button presses. Button debouncing can be implemented either in hardware or in software; this module implements debouncing in software.

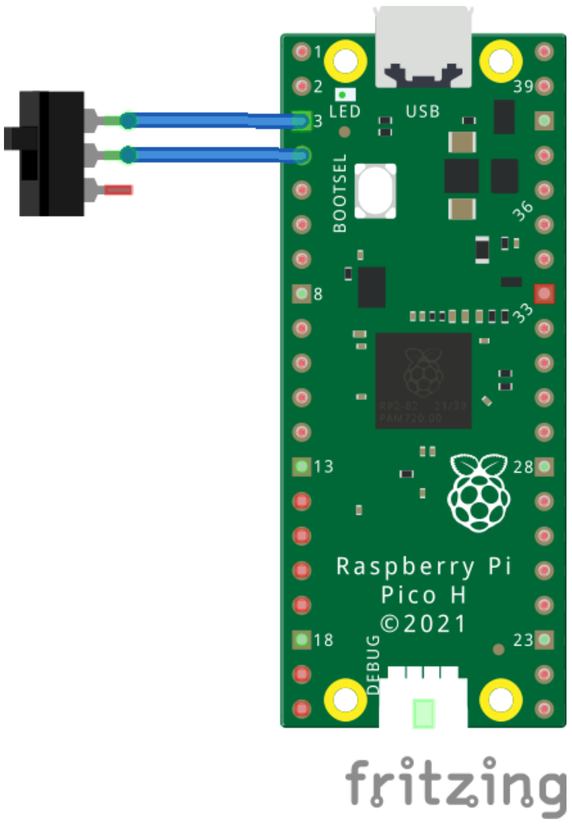
5.4.1 Dependencies

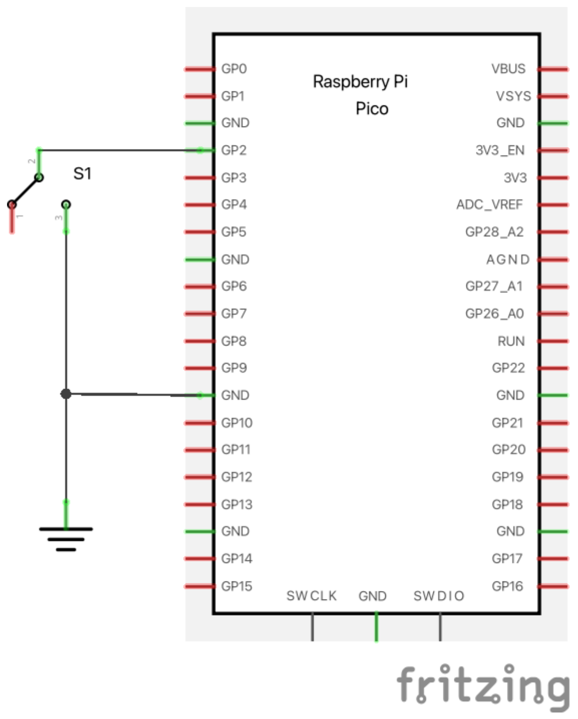
No dependencies.

5.4.2 DebouncedPin

Inherits from micropython's builtin `machine.Pin`.

Typical usage:





```
from debouncedpin import DebouncedPin

pin = DebouncedPin(2, DebouncedPin.PULL_UP)

# Directly call the object to get the current state.
value = pin()

# Alternatively the value can be retrieved via the ``value`` method.
value = pin.value()
```

To automatically execute a function when a change in switch state is detected, configure an interrupt:

```
from debouncedpin import DebouncedPin


def handler(pin):
    print("Button was just pressed!")


pin = DebouncedPin(2, DebouncedPin.PULL_UP)
pin.irq(handler, DebouncedPin.IRQ_FALLING)

while True:
    # Program can be doing other stuff here.
    pass
```

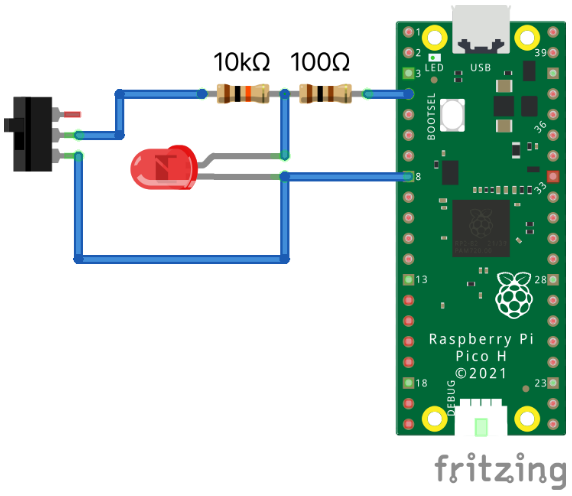
How It Works

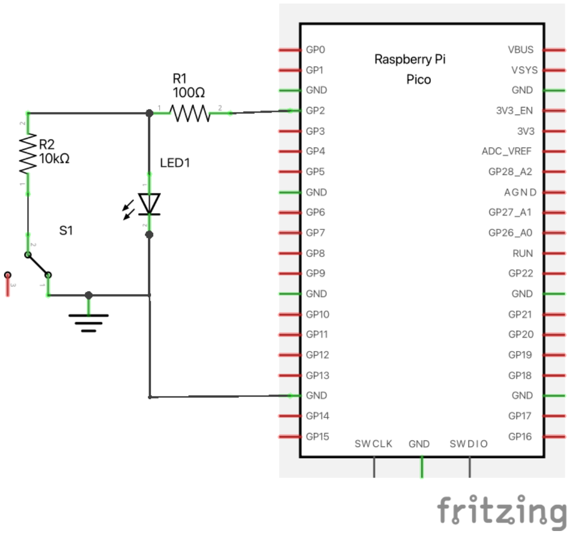
Every 20 milliseconds, the DebouncedPin object will poll the input pin state. If 2 consecutive reads report the same value, the cached switch state is updated. When the user wants to read the switch state, the cached switch state is returned.

5.4.3 DebouncedLedPin

Inherits from DebouncedPin. All DebouncedPin documentation also applies to this class.

DebouncedLedPin can control an LED and read the state of a button using a single microcontroller pin. Compared to a naive switch/LED circuit using 2 microcontroller pins, this setup requires an additional 10k resistor. Sharing a pin can be useful for reducing the number of IO required for a project. If the led/switch are mounted elsewhere, this can also reduce the number of wires required in the cable.





```python
from debouncedpin import DebouncedLedPin

pin = DebouncedLedPin(2, DebouncedPin.PULL_UP)

# Various ways to read the current switch state:
```

(continues on next page)

(continued from previous page)

```
value = pin()
value = pin.value()

# Various ways to turn on LED
pin(True)
pin.value(True)
pin.on()

# Various ways to turn off LED
pin(False)
pin.value(False)
pin.off()
```

5.4.4 Signal

The `Signal` class takes in a Pin-like object as input. Optionally, set the `invert=True` flag to invert physical input/output values.

The builtin `machine.Signal` class doesn't handle pin-like objects well. However, this implementation won't be as fast/resource-efficient, but that's fine for many cases.

If wrapping a vanilla `machine.Pin` object, it's recommended to use the built in `machine.Signal` class. If wrapping an object that implements the pin Protocol, then use the `Signal` class implemented here.

```
from debouncedpin import DebouncedPin
from signal import Signal

pin = DebouncedPin(7)  # This won't work with ``machine.Signal``
signal = Signal(pin, invert=True)  # If the output is active-low

signal.on()  # Turn signal off, setting pin high
signal.off()  # Turn signal off, setting pin low
signal.value(True)  # alternative ways to turn signal on
signal(True)

val = signal()  # Read input; this also abides by ``invert``.
val = signal.value()
```

How It Works

In addition to the explanation of how `DebouncedPin` works, `DebouncedLedPin` will set the pin to be in an output configuration between pin input reads. This results in the pin being in output for the majority of the time, turning the LED on or off. For the very brief moment that the pin is changed to input mode, the LED will be off while the button state is read. This brief moment where the pin is in input mode is imperceptable to the human eye.

5.5 Interp1D

1D interpolation methods.

5.5.1 Dependencies

No dependencies

5.5.2 Interpolater

Base interpolater class. Not to be directly instantiated. All interpolaters have the same interface; linear interpolation is shown below as a demonstration:

```
import interp1d

x = [1, 5, 17]  # Inputs
y = [20, 6, 14]  # Outputs
mapping = interp1d.Linear(x, y)
mapping(11)  # 10.0
```

The input `x` array **must** be sorted.

By default, querying a value outside the range of input datapoints will raise a `ValueError`.

```
mapping(18)  # raises ValueError
```

We can control this behavior with the `fill_value` argument by either specifying the string `"clip"` or by specifying a tuple of 2 floats representing `(low, high)`:

```
import interp1d

x = [1, 5, 17]
y = [20, 6, 14]

mapping = interp1d.Linear(x, y, fill_value="clip")
mapping(0)  # 20
mapping(18)  # 14

mapping = interp1d.Linear(x, y, fill_value=(-100, 100))
mapping(0)  # -100
mapping(18)  # 100
```

5.5.3 Linear

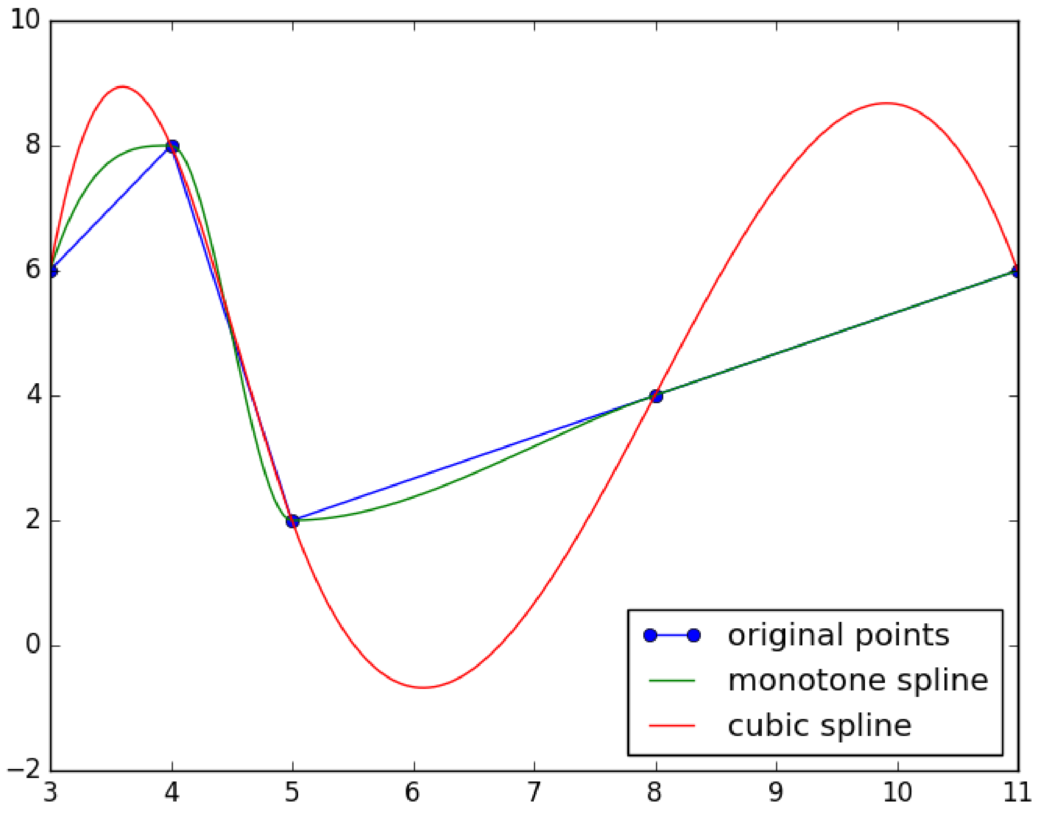
Linear interpolation.

5.5.4 Cubic

Cubic interpolation.

5.5.5 MonoSpline

Monotone preserving cubic interpolation. Unlike `Cubic`, `MonoSpline` guarantees interpolated values to be in-between it's supporting points.



The above graphic comes from antdvid's implementation, which this implementation is also based on.

5.5.6 searchsorted

Not an interpolation function, but may be externally useful. For a given sorted iterable, finds the index where a value should be inserted to maintain order.

```
data = [1, 5, 10]
searchsorted(data, 0)  # 0
searchsorted(data, 1)  # 0
searchsorted(data, 2)  # 1
searchsorted(data, 8)  # 2
searchsorted(data, 11)  # 3
```

5.6 MAX6675

Cold-Junction-Compensated K-Thermocouple-to-Digital Converter (0°C to +1024°C).

5.6.1 Dependencies

- `controlhal`

5.6.2 Max6675

The `Max6675` class uses a `SPI` object and a chip-select pin to handle communications:

```
from machine import Pin, SPI
from max6675 import Max6675

spi = SPI(0, mosi=Pin(3, Pin.OUT), miso=Pin(4, Pin.IN), sck=Pin(2, Pin.OUT))
max6675 = Max6675(spi, cs=Pin(5, Pin.OUT))
temperature_celsius = max6675.read()
```

The MAX6675 can be polled with a minimum period of 0.22 seconds. `Max6675` inherits from the `controlhal.Sensor` class, so it inherits some of it's benefits, such as caching the previous temperature value if reads are attempted faster than the 0.22 second period.

If the thermocouple becomes disconnected during operation, the next read will cause a `OpenThermocouple` exception to be raised:

```
from max6675 import OpenThermocouple

try:
    temperature_celsius = max6675.read()
except OpenThermocouple:
    print("Thermocouple disconnected!")
```

The MAX6675 must be physically reset after the thermocouple has been reconnected.

5.7 OverSample

5.7.1 Dependencies

No dependencies

5.7.2 Oversample

OOP oversampling of a callable function.

```python
from oversample import Oversample


def foo():  # takes no arguments
    return 42


oversample_foo = Oversample(foo, 64)  # Will sample foo 64 times each call
oversampled_value = oversample_foo()
```

5.7.3 oversample

Functional oversampling a callable function.

```python
from oversample import oversample


def foo():  # takes no arguments
    return 42


oversampled_value = oversample(foo, 64)  # Sample foo 64 times
```

5.8 PID

Proportional-Integral-Derivative Controller.

A PID controller predicts how much an actuator should be given a relevant feedback signal. A common example is in a water boiler where the PID controller predicts how much the heating element should be turned on given a feedback temeperature signal from a temperature sensor.

5.8.1 Dependencies

- controlhal

5.8.2 PID

For most use-cases, the PID class takes in up to 3 inputs:

```
from pid import PID

controller = PID(0.05)  # A "P" controller
controller = PID(0.05, 0.001)  # A "PI" controller
controller = PID(0.05, 0.001, 3.2)  # A "PID" controller
```

By default, the output is limited to the floating-point range [0, 1], which should be interpreted as 0 to 100%. This output normalization makes it easier to hook up different actuators to a PID controller.

```
controller = PID(0.5, setpoint=80.0)  # Setpoint can be configured at initialization.
controller.setpoint = 80  # Can directly set the attribute whenever.
```

The PID controller should be called at a fix interval. If called faster than the specified period (defaults to 0.01 seconds), then the internal state will **not** be updated, and the previous actuator prediction value will be returned.

```
while True:  # Main application loop
    temperature = sensor.read()
    actuator_power = controller(temperature)
    actuator.write(actuator_power)
```

This sensor-read, actuator-write code is wrapped up in controlhal.ControlLoop.

5.8.3 Acknowledgements

This library is modified from m-lundberg's implementation. In turn, his library was inspired by Brett Beauregard's arduino library, PIDLibrary.

I added the following modifications:

1. Fix some micropython incompatibilities.
2. Don't accumulate the integral component while the acutator is in saturation. This helps prevent integral windup.
3. Change a few parameter names to be consistent with some other libraries.
4. Set default output limits to (0, 1). The error_map feature has been removed in favor of a normalized output.

5.9 PIDAutotune

Proportional-Integral-Derivative autotuning using the Relay Method.

5.9.1 Dependencies

- `controlhal`
- `ringbuffer`

While not a direct dependency, this library is intended to be used in conjunction with the `pid` library.

5.9.2 PIDAutotune

The main class, `PIDAutotune`, is itself a `controlhal.Controller`. For typical usage, you can temporarily swap out

```
from controlhal import ControlLoop, AutotuneSuccess, AutotuneFailure
from pid import PID
from pidautotune import PIDAutotune
from time import sleep

heater = get_actuator()  # Just some controhal.Actuator
thermometer = get_sensor()  # Just some controlhal.Sensor

pid = PID(0.05, 0.001)
control_loop = ControlLoop(heater, thermometer, pid)

# Set the hysterisis to be a small value, but larger than random peak-to-peak
# sensor noise.
autotuner = PIDAutotune(control_loop.setpoint, hysterisis=1.5)

# Could do normal control_loop things, but here we will temporarily
# swap out the PID controller for the PIDAutotune controller.

with control_loop.use(autotuner):
    try:
        while True:
            control_loop()
            sleep(0.5)
    except AutotuneSuccess as e:
        print(f"PID parameters: {e.parameters}")
        pid.parameters = e.parameters
    except AutotuneFailure:
        print("Autotuner failed to converge")

# Continues on with ``pid``
```

By default, parameters are computed using the `"some-overshoot"` method. Available tuning formulas include:

- `"some-overshoot"` - Default
- `"ziegler-nichols-p"`
- `"ziegler-nichols-pi"`

- "ziegler-nichols-pid"
- "no-overshoot"
- "tyreus-luyben"

The parameter computation technique can be specified by setting `method` when creating the autotuner. Alternatively, a tuning rule can be specified

```
autotuner = PIDAutotune(80.0, hysterisis=1.5, method="ziegler-nichols-pi")
# after running until AutotuneSuccess:
k_p, k_i, k_d = autotuner.compute_tunings("ziegler-nichols-pid")
```

5.9.3 Acknowledgements

This library is heavily modified from hirschmann's implementation. In turn, his library was inspired by Brett Beauregard's arduino library, PIDAutotuneLibrary. Large portions of the code have been rewritten to favor readability (like using floating point) over microcontroller performance.

5.10 RingBuffer

Simple ring/circular buffer for storing numbers. Commonly used for moving-window statistics.

5.10.1 Dependencies

No dependencies.

5.10.2 RingBuffer

Demo code showing off most of RingBuffer's functionality:

```
from ringbuffer import RingBuffer

buf = RingBuffer(3)

len(buf)  # 0, there are 0 elements currently in the RingBuffer
buf.max_size  # 3, The RingBuffer can hold up to 3 elements.

buf.append(5)
buf.append(10)
buf.append(25)

# Common statistics
buf.mean()  #  13.3333
buf.median()  # 10.0
buf.min()  # 5.0
buf.max()  # 25.0

# Finite Differences
# ``buf.diff()`` returns a generator.
```

(continues on next page)

(continued from previous page)

```
list(buf.diff())  # [5.0, 15.0],  Will be 1-shorter than ``len(buf)``.

# Indexing
buf.append(50)  # Will overwrite the first element, that used to be ``5``
buf[0]  # 10.0, the 0th index will contain the oldest value
buf[-1]  # 50.0, the -1th index will contain the newest value

buf.full  # True, the buffer is currently full

list(
    buf
)  # [10.0, 25.0, 50.0], iterating over RingBuffer will go from oldest to newest.

buf.clear()  # Resets the RingBuffer
buf.full  # False, the RingBuffer has been cleared and is currently empty.
```

The underlying element size can be configured by specifying the `dtype` argument:

```
buf = RingBuffer(3, "b")  # signed character
buf = RingBuffer(3, dtype="d")  # double
```

All `dtype` specifiers are the same as for `array.array` objects.

dtype	C Type	Size
b	signed char	1
B	unsigned char	1
h	signed short	2
H	unsigned short	2
i	signed int	2
I	unsigned int	2
l	signed long	4
L	unsigned long	4
q	signed long long	8
Q	unsigned long long	8
f	float (DEFAULT)	4
d	double	8

5.11 uProfiler

Tools to help identify slow parts of your code.

5.11.1 Dependencies

No dependencies

5.11.2 profile

The `profile` decorator measures how long the function/method takes to execute.

If a `name` is not provided, then it is set to the decorated function name. Decorated functions with the same name will overwrite each other's summary results.

The optional `print_period` will control how often the decorator will print the call timings. Set to 0 to suppress all prints. The **default** global print period can be configured via `uprofiler.print_period`. Defaults to `1` (every call).

```python
from time import sleep
import uprofiler

uprofiler.print_period = 1  # Modifies global default


@uprofiler.profile
def foo():
    sleep(0.25)


@uprofiler.profile(name="changed_bar_name")
def bar():
    sleep(0.6)


@uprofiler.profile(print_period=3)
def baz():
    sleep(0.1)


foo()

bar()
bar()

baz()
baz()
baz()
```

5.11.3 print_results

Prints total execution time, as well as a summary of all `profile` calls. `Total-Time` is computed as the elapsed time between initial `uprofiler` import and the `print_results` function call.

Place this at the end of your script.

```
import uprofiler

uprofiler.print_results()
```

The printed output table in this case is empty, since no `profile` calls were made.

```
Total-Time:  0.983ms
Name                        Calls    Total (%)     Total (ms)   Average (ms)
----------------------------------------------------------------------------
```

5.11.4 Demo

The demo `demos/uprofiler.py` prints the following output:

```
foo                                1 calls      250.225ms total      250.225ms average
changed_bar_name                   1 calls      600.160ms total      600.160ms average
changed_bar_name                   2 calls     1200.310ms total      600.155ms average
baz                                3 calls      300.302ms total      100.101ms average
baz                                6 calls      600.594ms total      100.099ms average

Total-time: 2184.748ms
Name                        Calls    Total (%)     Total (ms)   Average (ms)
----------------------------------------------------------------------------
changed_bar_name                2        54.94        1200.31        600.155
baz                             6        27.49        600.594        100.099
foo                             1        11.45        250.225        250.225
```

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search