aiorate Documentation

Release 1.0.1

Stéphane Caron

Jun 12, 2022

Contents

Loop frequency regulator for asyncio, with an API similar to rospy.Rate.

1 Installation

```
pip install aiorate
```

2 Rate limiter

This module provides a non-blocking loop frequency regulator in the `Rate` class.

Note that there is a difference between a (non-blocking) rate limiter and a (blocking) synchronous clock, which lies in the behavior when skipping cycles. A rate limiter does nothing if there is no time left, as the caller's rate does not need to be limited. On the contrary, a synchronous clock waits for the next tick, which is by definition in the future, so it always waits for a non-zero duration.

class aiorate.rate.**Rate**(*frequency*, *name='rate_limiter'*)

Loop frequency regulator.

Calls to *sleep()* are non-blocking most of the time but become blocking close to the next clock tick to get more reliable loop frequencies.

This rate limiter is in essence the same as in the one from pymanoid. It relies on the event loop time never jumping backwards nor forwards, so that it does not handle such cases contrary to e.g. rospy.Rate.

measured_period

Actual period in seconds measured at the end of the last call to *sleep()*.

Type float

name

Human-readable name used for logging.

Type str

period

Desired loop period in seconds.

Type float

slack

Duration in seconds remaining until the next tick at the beginning of the last call to *sleep()*.

Type float

async remaining()

Get the time remaining until the next expected clock tick.

Return type float

Returns Time remaining, in seconds, until the next expected clock tick.

async sleep(*block_duration=0.0005*)

Sleep the duration required to regulate the loop frequency.

This function is meant to be called once per loop cycle.

Parameters **block_duration** (float) – the coroutine blocks the event loop for this duration (in seconds) before the next tick. It is non-blocking before that.

Note: A call to this function will be non-blocking *except* for the last block_duration seconds of the limiter period.

The block duration helps trim period overshoots and brings the measured period much closer to the desired one (< 2% average error vs. 8-12% average error with a single asyncio.sleep). Empirically a block duration of 0.5 ms gives good behavior at 400 Hz or lower.

3 Example

In short, a loop is rate-regulated this way:

```
import asyncio
import aiorate

async def main():
    rate = aiorate.Rate(400.0)  # Hz
    while True:
        loop_time = asyncio.get_event_loop().time()
        print(f"Hello from loop at {loop_time:.3f} s")
        await rate.sleep()

if __name__ == "__main__":
    asyncio.run(main())
```

You can await `rate.sleep()` anywhere inside the loop.

You can download the full documentation as a PDF document.

Python Module Index

a

Index