nyquist

Release 2021

Marco Miretti

Sep 04, 2022

CONTENTS:

nyquist is a library for programming control-systems experiments and execute them in remote laboratories.

Here is an example of a bang-bang control loop with nyquist:

```
import time
import numpy as np
import matplotlib.pyplot as plt

from nyquist.lab import System
from nyquist.control import Experiment


# we create an inherited class from Experiment
class MyExperiment(Experiment):

    # the three fundamental functions should be defined
    def before_the_loop(self):
        # we will use aero to comunicate with the system
        self.aero = System("aeropendulum")
        self.aero.propeller.pwm.status.post("initialized")
        self.aero.telemetry.period.post(20)
        self.aero.logger.level.post("LOG_INFO")
        self.angle = []
        self.time = []
        self.aero.sensors.encoder.angle.get()
        self.start_ts = time.monotonic()

    def in_the_loop(self):
        if self.aero.sensors.encoder.angle._Endpoint__resourcer.new_message:
            angle = self.aero.sensors.encoder.angle.get()
            self.time.append(time.monotonic() - self.start_ts)
            self.angle.append(angle)
            print(angle)
            if angle is not None:
                if angle < self.setpoint_deg:
                    self.aero.propeller.pwm.duty.post(self.duty_high)
                else:
                    self.aero.propeller.pwm.duty.post(self.duty_low)

    # the after script will be executed even on failure
    def after_the_loop(self):
        self.aero.propeller.pwm.duty.post(0)
        self.aero.propeller.pwm.status.post("disabled")
        self.aero.propeller.pwm.status.post("initialized")


# instance experiment
exp = MyExperiment()

# set constants
exp.setpoint_deg = 40
exp.duty_high = 35
```

(continues on next page)

(continued from previous page)

```
exp.duty_low = 25
exp.set_loop_frequency(20)
exp.set_run_time(6)
exp.set_before_loop_time(1)
# and run!
exp.run()

plt.figure()
print("---")
print(exp.time[-1] - exp.time[0])
print(len(exp.angle))
plt.plot(exp.time, np.sin(np.deg2rad(exp.angle)))
plt.show()
```

CHAPTER

ONE

TUTORIAL

Get started with `nyquist`!

1.1 Getting started

1.1.1 Requirements

`nyquist` requires:

- Python 3.6
- websockets python package

1.1.2 Installation

Install `nyquist` with:

```
pip install nyquist
```

1.1.3 Basic example

A basic bang-bang control example with aeropendulum:

```
import time
import numpy as np
import matplotlib.pyplot as plt

from nyquist.lab import System
from nyquist.control import Experiment


# we create an inherited class from Experiment
class MyExperiment(Experiment):

    # the three fundamental functions should be defined
    def before_the_loop(self):
        # we will use aero to comunicate with the system
        self.aero = System("aeropendulum")
```

(continues on next page)

(continued from previous page)

```
        self.aero.propeller.pwm.status.post("initialized")
        self.aero.telemetry.period.post(20)
        self.aero.logger.level.post("LOG_INFO")
        self.angle = []
        self.time = []
        self.aero.sensors.encoder.angle.get()
        self.start_ts = time.monotonic()

    def in_the_loop(self):
        if self.aero.sensors.encoder.angle._Endpoint__resourcer.new_message:
            angle = self.aero.sensors.encoder.angle.get()
            self.time.append(time.monotonic() - self.start_ts)
            self.angle.append(angle)
            print(angle)
            if angle is not None:
                if angle < self.setpoint_deg:
                    self.aero.propeller.pwm.duty.post(self.duty_high)
                else:
                    self.aero.propeller.pwm.duty.post(self.duty_low)

    # the after script will be executed even on failure
    def after_the_loop(self):
        self.aero.propeller.pwm.duty.post(0)
        self.aero.propeller.pwm.status.post("disabled")
        self.aero.propeller.pwm.status.post("initialized")


# instance experiment
exp = MyExperiment()

# set constants
exp.setpoint_deg = 40
exp.duty_high = 35
exp.duty_low = 25
exp.set_loop_frequency(20)
exp.set_run_time(6)
exp.set_before_loop_time(1)
# and run!
exp.run()

plt.figure()
print("---")
print(exp.time[-1] - exp.time[0])
print(len(exp.angle))
plt.plot(exp.time, np.sin(np.deg2rad(exp.angle)))
plt.show()
```

CHAPTER

TWO

REFERENCE

Find all the details about the API.

2.1 API

2.1.1 Lab Client

The System Class

class nyquist.lab.**System**(*description*, *ip=None*, *http_resources=None*, *ws_resources=None*, *http_port=None*, *ws_port=None*, *http_timeout=None*, *ws_timeout=None*, *ws_get_mode=None*)

Generates an object with the complete resource tree as attributes.

An instance from System defines a laboratory sistem completely, and allows the user to interact with it.

Given an string with the name of the device to control, the library will load the default values for it's configuration. Said configuration can also be modified via keyword optional arguments.

The last word of each URL, each resource is an `_HTTPEndpoint` or `_WSEndpoint`, that according the methods assigned to the resource, will have a *get()* and/or `post()` method... or *get()* and/or `post()` method. The latter ones are much faster, but require an open websockets stream.

Parameters

- **description** (*str*) – The system name.
- **ip** (*str*) – An IP address or domain.
- **http_resources** (*tuple*) – Set of HTTP resources.
- **ws_resources** (*tuple*) – Set of Websocket resources.
- **http_port** (*int*) – Destination HTTP port.
- **ws_port** (*int*) – Destination Websocket port.
- **http_timeout** (*float*) – Timeout for each HTTP request.
- **ws_timeout** (*str. ("new" or "last").*) – Timeout for each ws request.
- **ws_get_mode** – Strategy to gather messages.

Methods for HTTP endpoints

_HTTPResourcer.**get**(*resource*, *retval_mode='payload'*)

Gets the value of a resource.

Parameters

- **resource** (*string*) – The resource whose value we want. If using any *nyquist.lab.client.System* object, this argument mustn't be passed.
- **retval_mode** (*string*) – The type of return value we expect.

Returns
The value of the resource.

Return type
depends on the resource.

_HTTPResourcer.**post**(*resource*, *value*, *retval_mode='code'*)

Gets the value of a resource.

Parameters

- **resource** (*string*) – The resource whose value we want to set. If using any *nyquist.lab.client.System* object, this argument mustn't be passed.
- **value** (*depends on the resource*) – The resource whose value we want to set.
- **retval_mode** (*string*) – The type of return value we expect.

Returns
The return code.

Return type
int

Methods for WS endpoints

_WSResourcer.**get**(*resource*)

Gets the last telemetry value of given resource.

If the telemetry is not initialized (socket not opened yet) it opens a websocket and starts receiving telemetry asynchronously. Said telemetry will be saved, and can be retrieved by this method.

Is most likely that this method will return None the first time it's called, so it's advised to call it in the after_script first.

Every call of this method will set `new_message` to False, and every time a new message is stored, that attribute will be set to true. This way the user can implement methods to avoid reading the same message twice.

_WSResourcer.**post**(*resource*, *value*)

Sets the value of a resource through a fast channel, asynchronously.

2.1.2 Experiments

Defining the control algorithm

abstract Experiment.**before_the_loop**()

The user should define this method. When *Experiment.run()* is called, this method will run first. Then it will wait for some time that can be set with *Experiment.set_before_loop_time()* and after said delay, it will run the loop.

abstract Experiment.**in_the_loop**()

The user should define this method. When *Experiment.run()* is called, this method will run in a loop, after *Experiment.before_the_loop()*. This method will run in a loop with a frequency established by *Experiment.set_loop_frequency()*.

abstract Experiment.**after_the_loop**()

The user should define this method. When *Experiment.run()* is called, this method will run after the loop, and after a delay that can be configured with *Experiment.set_after_loop_time()*.

Experiment object configuration

Experiment.**set_loop_frequency**(*frequency_hz*)

Define the frequency of the control-loop [Hz].

Parameters
frequency_hz (*float*) – The frequency in hertz.

Experiment.**set_run_time**(*time_s*)

Define the simulation run time [s]. This includes only the time spent inside the control loop.

Parameters
time_s (*float*) – The time in seconds.

Experiment.**set_before_loop_time**(*time_s*)

Set the delay time between the *Experiment.before_the_loop()* and the start of the loop.

Parameters
time_s (*float*) – The time in seconds.

Experiment.**set_after_loop_time**(*time_s*)

Set the delay time between end of the loop and *Experiment.after_the_loop()*.

Parameters
time_s (*float*) – The time in seconds.

Execution

Experiment.**run**(*blocking=True*)

Execute the control experiment, run:

```
before_the_loop()
sleep(_before_loop_time_s)
while _time_spent < _run_time_s:
    in_the_loop()
    sleep(1 /_loop_period_s)
```

(continues on next page)

(continued from previous page)

```
sleep(_after_loop_time_s)
after_the_loop()
```

2.2 External Docs

2.2.1 Laboratories APIs

Aeropendulum

https://marcomiretti.gitlab.io/remote-control-lab/openapi.html

2.2.2 Laboratories firmwares

Aeropendulum

https://marcomiretti.gitlab.io/remote-control-lab/

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search

INDEX