CLEAVE

ExPECA Project Team

Sep 13, 2022

TABLE OF CONTENTS

CHAPTER

ONE

CLEAVE - CONTROL BENMARKING SERVICE ON THE EDGE

A framework for testing, benchmarking and evaluating control loop applications on the Edge, written in Python 3.8+.

Note: This project is in early stages of development.

CLEAVE is part of the ExPECA research project at KTH Royal Institute of Technology. It aims at providing a powerful and flexible platform for the study of networked control systems, particularly on Edge Computing architectures.

1.1 Installation

1.1.1 Installation for general usage

NOTE: Although these instructions will eventually be the recommended way of downloading and installing the framework, they are currently a work-in-progress, and do not work yet. For failsafe ways to install the framework, see section *Installation for development* below.

From PyPI

1. Set up a Python `virtualenv` and activate it:

```
$ virtualenv --python=python3.8 ./venv
created virtual environment CPython3.8.3.final.0-64 in 212ms
$ . ./venv/bin/activate
(venv) $
```

2. Next, install the framework from PyPI using `pip`:

```
(venv) $ pip install cleave
```

3. Alternatively, if you already have set up a project with an associated virtualenv, you can add `cleave` to your `requirements.txt` :

```
# example requirements.txt
...
numpy
pandas
cleave
scipy
```

```
(venv) $ pip install -Ur ./requirements.txt
```

From the repository

1. Set up a `virtualenv` and activate it as before:

```
$ virtualenv --python=python3.8 ./venv
created virtual environment CPython3.8.3.final.0-64 in 212ms
$ . ./venv/bin/activate
(venv) $
```

2. Install the package through `pip` by explicitly pointing it toward the repository:

```
(venv) $ pip install -U git+git://github.com/KTH-EXPECA/CLEAVE.git#egg=cleave
```

3. This can also be inserted into a `requirements.txt` file:

```
# example requirements.txt
...
numpy
pandas
-e git://github.com/KTH-EXPECA/CLEAVE.git#egg=cleave
scipy
```

1.1.2 Installation for development

1. Clone the CLEAVE repository:

```
$ git clone git@github.com:KTH-EXPECA/CLEAVE.git
$ cd ./CLEAVE
```

2. Create a Python 3.8+ `virtualenv` and install the development dependencies:

```
$ virtualenv --python=python3.8 ./venv
created virtual environment CPython3.8.3.final.0-64 in 212ms
...

$ . ./venv/bin/activate
(venv) $ pip install -Ur ./requirements.txt
...
```

(Optional) Set up the Sphinx documentation environment

1. Install the documentation dependencies:

```
(venv) $ pip install -Ur requirements_docs.txt
```

2. Document code using the Numpy docstring format (see below).

3. Generate reStructured text files for the code by running `sphinx-apidocs` from the top-level directory and passing it the output directory (`docs/source`) and the `cleave` package directory as arguments:

```
$ sphinx-apidoc -fo docs/source ./cleave
```

4. Finally, to preview what the documentation will look like when published on readthedocs, build it with `GNU Make`:

```
$ cd docs/
$ make html
```

The compiled HTML structure will be output to `docs/build`, from where it can be viewed in a browser.

1.2 Emulating a Networked Control System

Emulations of Networked Control Systems in CLEAVE are centered around two core concepts: Plants and Controller Services. These terms follow the terminology used in Control Systems research: Plants are physical systems we wish to control, whereas Controller Services are the computational elements which perform the necessary computations for the controlling of Plants. CLEAVE provides an easy-to-use API to implement these components, and abstracts away the network code, allowing researchers to focus on the NCS itself.

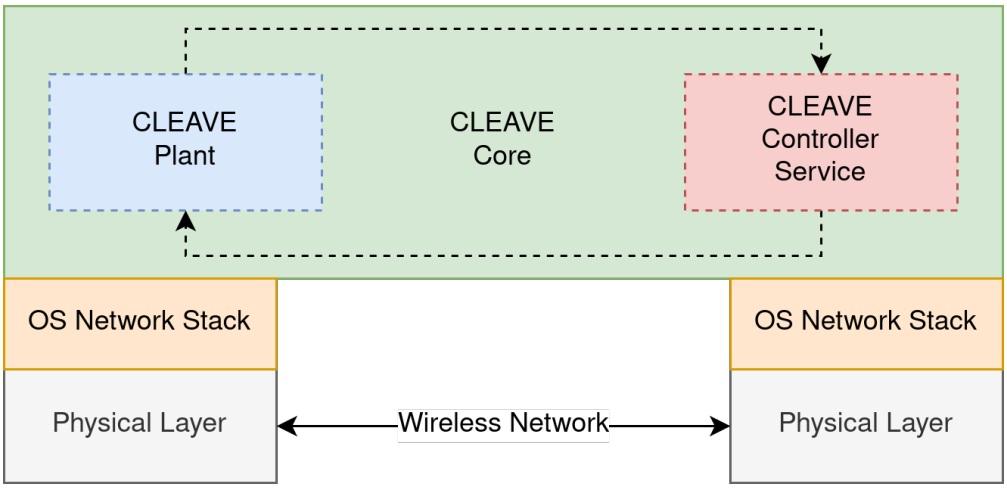

Fig. 1: General overview of CLEAVE's design.

In practical terms, the definitions of Plants and Controller Services are done through configuration files written in pure Python. These are then executed using the `cleave.py` launcher script. A couple of of such configuration files can be found under the `examples/` directory, and we will discuss them in more detail in *Deploying the included NCS emulations*.

To execute a Plant, use the `run-plant` sub-command and provide a Plant configuration file:

```
(venv) $ python cleave.py run-plant examples/plant_config.py
```

To execute a Controller Service, use the `run-controller` subcommand and provide a Controller Service configuration file.

```
(venv) $ python cleave.py run-controller examples/controller_config.py
```

Check `cleave.py --help` for more details and additional options. In particular, the `-v|--verbose` flag is very useful; it increases the verbosity of the logged output, and can be specified multiple times. Examples:

```
(venv) $ python cleave.py run-plant examples/plant_config.py
...
(venv) $ python cleave.py -vv run-plant examples/plant_config.py
2020-11-25T18:50:22.908532+0100 WARNING /mnt/data/workspace/CLEAVE/plant_metrics/
```

(continues on next page)

(continued from previous page)

```
↪simulation.csv will be overwritten with new data.
2020-11-25T18:50:22.909054+0100 WARNING /mnt/data/workspace/CLEAVE/plant_metrics/client.
↪csv will be overwritten with new data.
2020-11-25T18:50:22.909384+0100 WARNING /mnt/data/workspace/CLEAVE/plant_metrics/sensors.
↪csv will be overwritten with new data.
2020-11-25T18:50:22.909749+0100 WARNING /mnt/data/workspace/CLEAVE/plant_metrics/
↪actuators.csv will be overwritten with new data.
2020-11-25T18:50:22.910301+0100 WARNING Target frequency: 200 Hz
2020-11-25T18:50:22.910360+0100 WARNING Target time step: 5.0 ms
...
```

1.2.1 Deploying the included NCS emulations

CLEAVE comes with a number of pre-configured NCS emulations, composed of a Plant simulating an inverted pendulum and a number of Controller Services that interact with this Plant and control the pendulum in different ways.

Inverted pendulum Plant

The `examples/inverted_pendulum/plant/` directory contains the configuration files for the inverted pendulum Plant. Currently, this directory contains two files: `config.py` and `config_with_viz.py`. These files define identical Plant simulations, except for the fact that the latter includes a graphical visualization of the Plant in realtime.

Please refer to *Configuring the Plant* for details on customizing the Plant configurations, and to *cleave.impl.inverted_pendulum* for the actual implementations of the Plants.

Inverted pendulum Controller Services

Controller Services for the inverted pendulum NCS can be found under `examples/inverted_pendulum/controller/`. See each file for details on each Controller Service.

Please refer to *Configuring the Controller Service* for details on customizing the Controller Service configurations, and to *cleave.impl.inverted_pendulum* for the actual implementations of the Controllers.

1.2.2 Building a NCS emulation from scratch

In the following sections we will explain how to set up NCS emulations in CLEAVE by developing and configuring Plants and Controller Services from scratch and connecting them.

Plants

These are the representations of the physical systems wich we want to control. Plants in CLEAVE are usually physical simulations of some system we wish to monitor and act upon. Correspondingly, a Plant is composed of three subcomponents:

- A `State`, which implements the discrete-time behavior of the simulated system.
- A collection of `Sensor` objects, which measure specific properties of the `State`, potentially transforming them, and send them to the Controller Service.
- A collection of `Actuator` objects, which receive inputs from the Controller Service, potentially transform or distort them, and finally act upon specific properties of the `State`.

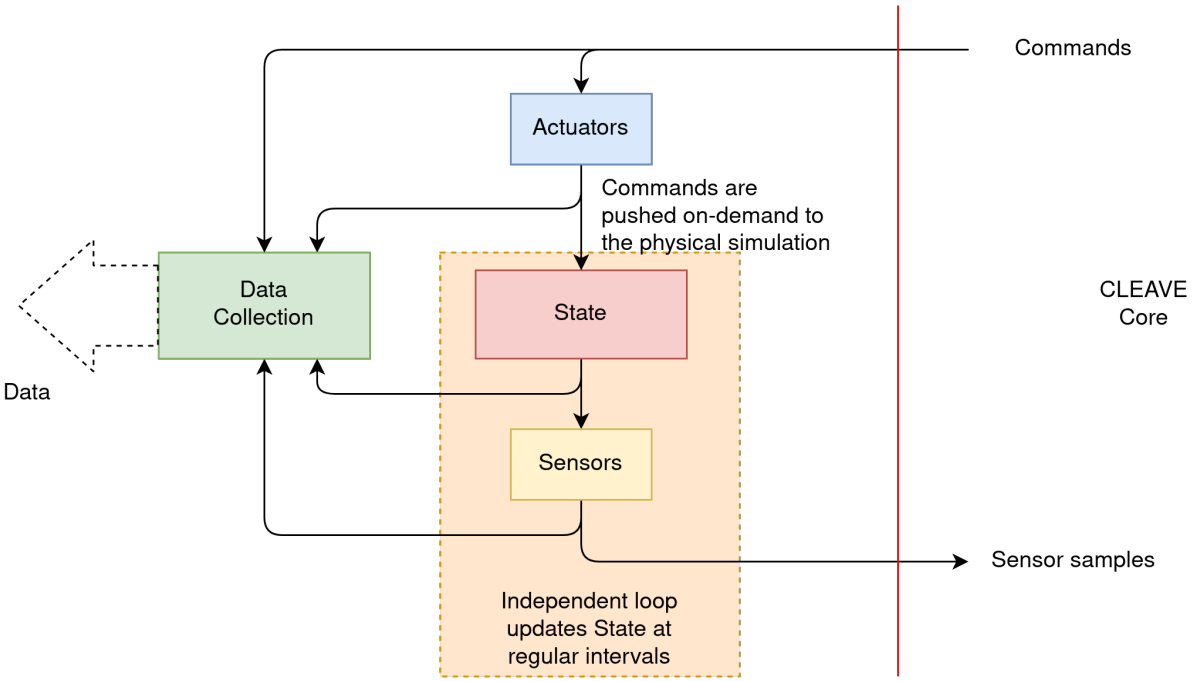

Fig. 2: General overview of the structure of, and flow of data in a Plant in CLEAVE.

State

`State` objects in CLEAVE are simply instances of classes which extend from the abstract base class *cleave.api.plant.State*. This base class defines a single required method as well as two optional ones:

```
class State:
    @abstractmethod
    def advance(self, dt: float) -> None:
        ...

    def initialize(self) -> None:
        ...

    def shutdown(self) -> None:
        ...
```

The *cleave.api.plant.State.advance()* method *must* be extended by inheriting classes. It is called by the framework on every iteration of the simulation, and thus users should implement their discrete-time plant logic here. The parameter `dt` corresponds to the number of seconds elapsed since the last invocation of the method.

Methods *cleave.api.plant.State.initialize()* and *cleave.api.plant.State.shutdown()* can optionally also be extended or overridden. They are called by the framework at the very beginning of the Plant execution and right before shutdown, respectively. It is in these methods users should put their initialization and shutdown logic.

`State` objects also need to expose the properties which that will be provided as inputs to the Controller and the properties the Controller acts upon. This is done by defining special *semantic* variables in the constructor of the `State`:

- *cleave.api.plant.SensorVariable* objects represent properties that will be measured by sensors and subsequently pushed to the Controller Service.

- *cleave.api.plant.ActuatorVariable* objects represent properties that will be modified by the actuation commands generated by the Controller Service. The values of these variables will be directly modified by the framework as commands come in.
- *cleave.api.plant.ControllerParameter* objects represent parameters passed to the Controller Service at the beginning of the emulation **(WIP, not implemented yet)**.

These objects are simply used to track the values during execution, and thus are completely transparent, allowing unrestricted access to the underlying raw values at all times. This means that after initialization, these variables can simply be used as normal "raw" values without having to consider the semantic variable object around it.

Furthermore, an optional "sanity check" may be attached to each semantic variable. This simply corresponds to a callable which receives the current value of the semantic variable and returns a boolean indicating if the current value is within acceptable ranges or not. This check will be executed at each time step of the Plant simulation, and if it at any point returns `False` the framework will record the corresponding variable and then halt the emulation.

An example skeleton of a `State` with a single input variable and a single output variable could then look something like the following:

```
class ExampleState(State):
    def __init__(self):
        super(ExampleState, self).__init__()

        self.accel = ActuatorVariable(0.0)
        self.speed = SensorVariable(0.0, sanity_check=lambda s: s < 200.0)
        # shuts down if speed ever exceeds 200.0 m/s

    def advance(self, dt: float) -> None:
        # To update the state, we simply read the ActuatorVariable, as it will always
        # hold the latest value from the controller at the beginning of each timestep,
        # and we write to the SensorVariable, as its value will automatically be sampled
        # by the corresponding Sensor.

        self.speed += dt * self.accel
```

As mentioned in *Deploying the included NCS emulations*, more complex example implementations of `State` classes representing Inverted Pendulum systems are included in the module *cleave.impl.inverted_pendulum*.

Sensors

Similarly to `State`, a `Sensor` in CLEAVE corresponds to an object instance of a subclass of *cleave.api.plant.Sensor* implementing the required method *cleave.api.plant.Sensor.process_sample()*. The *cleave.api.util.PhyPropType* typing variable in the signature simply represents the type of variables that can be measured in a Plant, currently `int`, `float`, `bool` and `bytes`.

```
class Sensor:
    def __init__(self, prop_name: str, sample_freq: int):
        ...

    @abstractmethod
    def process_sample(self, value: PhyPropType) -> PhyPropType:
        ...
```

As can be observed above, the `Sensor` base class constructor takes two parameters:

- `prop_name`: Corresponds to a string holding the name of the semantic variable the `Sensor` samples from.

- `sample_freq`: An integer representing the sampling frequency of this `Sensor` in Hz.

`Sensor` objects in the framework can be conceptualized as attaching to a semantic variable defined in the `State`. Whenever it is time for the `Sensor` to sample the value of this variable, `process_sample(value)` is called with its latest value, and whatever is returned is passed on to the Controller Service. Thus, users should extend `process_sample(value)` with any procedure to add noise or distortion to the measured variable they desire.

An example simple `Sensor` class which simply adds a bias to the measured value could be implemented as follows:

```
class BiasSensor(Sensor):
    def __init__(self, bias: float, prop_name: str, sample_freq: int):
        super(BiasSensor, self).__init__(prop_name, sample_freq)
        self._bias = bias

    def process_sample(self, value: PhyPropType) -> PhyPropType:
        return value + self._bias
```

Actuators

`Actuator` objects follow a similar logic as `Sensor` objects, in the sense that they "attach" to a semantic variable in the `State` and modify its value at each iteration following commands from the Controller Service.

In practical terms, `Actuator` objects correspond to instances of subclasses of *cleave.api.plant.Actuator*:

```
class Actuator:
    def __init__(self, prop_name: str):
        ...

    @abstractmethod
    def set_value(self, desired_value: PhyPropType) -> None:
        ...

    @abstractmethod
    def get_actuation(self) -> PhyPropType:
        ...
```

Again, the `prop_name` parameter in the constructor corresponds to the name of the semantic variable the `Actuator` attaches to. The *cleave.api.plant.Actuator.set_value()* and *cleave.api.plant.Actuator.get_actuation()* methods correspond to the required methods users should implement:

- `set_value(self, desired_value: PhyPropType) -> None` will be called by the framework whenever a new value for the actuated semantic variable is received from the Controller.
- `get_actuation(self) -> PhyPropType` will be called by the framework at the beginning of each simulation time step.

Note that due to the fact that commands from the Controller Service are received asynchronously, there are no guarantees regarding the order in which `set_value()` and `get_actuation()` are called with respect to each other. In fact, depending on the frequency of the plant simulation updates, the sensor sampling rates, network latency, and/or the time the Controller takes to process each input, either of these methods may be called *multiple* repeated times before the other. Users need to account for this when implementing new `Actuator` classes.

CLEAVE includes implementations for a number of different `Actuator` subclasses. For example, *cleave.api.plant.SimpleConstantActuator* implements an `Actuator` which remembers the last value set by the Controller Service and applies it on every simulation time step. This can be thought of as, for instance, an electrical motor maintaining a specific RPM until explicitly changed:

```
class SimpleConstantActuator(Actuator):
    def __init__(self, initial_value: PhyPropType, prop_name: str):
        super(SimpleConstantActuator, self).__init__(prop_name)
        self._value = initial_value

    def set_value(self, desired_value: PhyPropType) -> None:
        self._value = desired_value

    def get_actuation(self) -> PhyPropType:
        return self._value
```

Configuring the Plant

As discussed before, setting up Plants in CLEAVE is done through the use of configuration files written in pure Python. These configuration files may contain any valid Python code, be split up into multiple files, and even use external libraries. The only requirement is that the following top-level variables are defined:

- `host`: String containing the IP address of the Controller Service.
- `port`: Integer representing the UDP port on which the Controller Service is listening.
- `tick_rate`: Integer representing the update frequency of the Plant in Hertz. In other words, this number represents the number of iterations per second of the discrete-time simulation involving the `State`.
- `State`: A variable pointing to a valid instance of a subclass of `State`.
- `sensors`: A collection (list, tuple, set, etc) of instances of subclasses of `Sensor`.
- `actuators`: A collection of instances of subclasses of `Actuator`.
- *(Optional)* `output_dir`: This string should contain a path to a directory where the output metrics of the Plant will be written to (see *Plant output files* for details on the output files). If omitted, this variable defaults to `./plant_metrics/`.

Putting together our examples from the previous subsections, an example configuration file for the simple dummy `ExampleState` discussed previously would look something like the following:

Simulation of the Plant can then be initialized using the `cleave.py` launcher script:

```
(venv) $ python cleave.py run-plant dummy_plant_config.py
...
```

Plant output files

TODO

Controller Services

As discussed previously, a Controller Service correspond to the element in the NCS emulation which implements the necessary logic and computations to achieve the desired control of the Plant. In CLEAVE, Controller Services are implemented as stateful microservices paired with a specific Plant that receive samples of the Plant `State` semantic sensor variables over a UDP socket and return new values for the `State` semantic actuator variables over the same socket. Controller Services currently have a single user-defined component: a `Controller` which implements the control strategy.

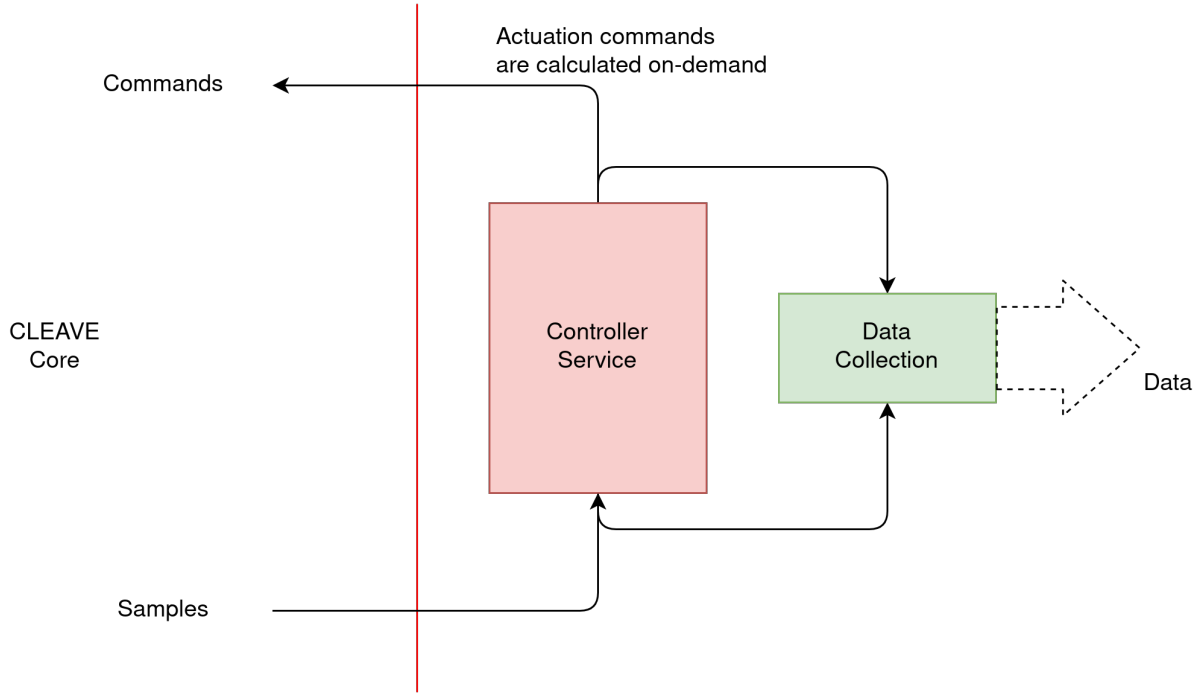

Fig. 3: General overview of the structure of, and flow of data in a Controller Service in CLEAVE.

Controllers

In practical terms, `Controller` objects are instances of subclasses of *`cleave.api.controller.Controller`*:

```
class Controller:
    @abstractmethod
    def process(self, sensor_values: PhyPropMapping) -> PhyPropMapping:
        ...
```

As seen above, this abstract base class defines a single required *`cleave.api.controller.Controller.process()`* method subclasses must implement. This method takes as argument a `Mapping` from sensor variable names to values, as is invoked whenever a new sample is received from the Plant. In turn, it must return a `Mapping` of actuator variable names to new values, which will subsequently be sent to the Plant.

Below we present an example `Controller` for our example Plant that operates on the `speed` and `accel` variables:

Configuring the Controller Service

Controller Service config files work the same way as Plant config files, the only difference being in the required top-level variables:

- `port`: Integer defining the UDP port on which the Controller Service listens.
- `controller`: Variable pointing to a valid `Controller` instance.
- *(Optional)* `output_dir`: This string should contain a path to a directory where the output metrics of the Controller will be written to (see *Controller output files* for details on the output files). If omitted, this variable defaults to `./controller_metrics/`.

The full example configuration file for our dummy Controller Service would then be:

Use `cleave.py` launcher script together with the config file to start listening for samples:

```
(venv) $ python cleave.py run-controller dummy_controller_config.py
...
```

Controller output files

TODO

1.3 Contributing (WIP)

This section contains information on how to contribute to this project.

1.3.1 Code style and standards

When developing on this project, please configure your IDE to adhere to the following guidelines.

The code in this repository should be PEP8 coding style guide compliant, with one exception: maximum line length. PEP8 specifies a maximum line length of 79 characters, a relic of a time where widescreen monitors didn't exist. In this project, we extend the maximum line length to 120 characters.

Furthermore, code documentation in this project should follow the Numpy docstring format as detailed here.

Finally, every Python module in this project should include an Apache License v2.0 statement at the top:

```
#  Copyright (c) 2020 KTH Royal Institute of Technology
#
#  Licensed under the Apache License, Version 2.0 (the 'License');
#  you may not use this file except in compliance with the License.
#  You may obtain a copy of the License at
#
#         http://www.apache.org/licenses/LICENSE-2.0
#
#  Unless required by applicable law or agreed to in writing, software
#  distributed under the License is distributed on an 'AS IS' BASIS,
#  WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
#  See the License for the specific language governing permissions and
#   limitations under the License.
```

CHAPTER

TWO

TEAM

- Manuel Olguín Muñoz: PhD student in Networking for Cyberphysical Systems; lead developer and maintainer for the CLEAVE project.
- Seyed Samie Mostafavi: PhD student in Networking for Cyberphysical Systems.
- Dr. James Gross: Professor in Networking for Cyberphysical Systems; PI for the ExPECA project.

CHAPTER

THREE

LICENSE

3.1 Indices and tables

- genindex
- modindex
- search

3.1.1 CLEAVE

cleave package

Subpackages

cleave.api package

Submodules

cleave.api.cli module

cleave.api.controller module

class cleave.api.controller.**Controller**

Bases: ABC

Base class for controllers, which defines a a simple interface that extending subclasses need to implement.

abstract process(*sensor_values: Mapping[str, Union[int, float, bool]]*) → Mapping[str, Union[int, float, bool]]

Processes samples and produces a control command.

Samples arrive in the form of a dictionary of mappings from sensor property name to measured value. Actuator commands should be returned in the same fashion, in a dictionary of mappings from actuated property name to actuated value.

This method needs to be implemented by extending subclasses with their respective logic.

Parameters
sensor_values – A mapping of sensor property names to measured values.

Returns
A mapping of actuated property names to desired values.

Return type
PhyPropMapping

cleave.api.plant module

exception cleave.api.plant.**UnrecoverableState**(*prop_values: Mapping[str, Union[int, float, bool]]*)

Bases: Exception

class cleave.api.plant.**ControllerParameter**(*value: Any*, *record: bool = False*)

Bases: *BaseSemanticVariable*

Note: Parameter from plant to controller is not yet implemented.

A semantically significant variable corresponding to an initialization parameter for a controller interacting with this plant.

Variables of this type will automatically be provided to the controller on initialization

Parameters

- **value** – The initial value for this variable.
- **record** – Whether this variable should be recorded or not (WIP) TODO: record variables
- **sanity_check** – A callable which takes a single input value and returns a boolean. If this value is not None, the callable give will be invoked with the updated value of the semantic variable after every tick of the plant. If the callable returns false, the simulation will fail with an UnrecoverableState error.

class cleave.api.plant.**SensorVariable**(*value: Any*, *record: bool = True*, *sanity_check: Optional[Callable[[Any], bool]] = None*)

Bases: *BaseSemanticVariable*

A semantically significant variable corresponding to a property measured by a sensor. Variables of this type will automatically be paired with the corresponding Sensor during emulation.

Parameters

- **value** – The initial value for this variable.
- **record** – Whether this variable should be recorded or not (WIP) TODO: record variables
- **sanity_check** – A callable which takes a single input value and returns a boolean. If this value is not None, the callable give will be invoked with the updated value of the semantic variable after every tick of the plant. If the callable returns false, the simulation will fail with an UnrecoverableState error.

class cleave.api.plant.**ActuatorVariable**(*value: Any*, *record: bool = True*, *sanity_check: Optional[Callable[[Any], bool]] = None*)

Bases: *BaseSemanticVariable*

A semantically significant variable corresponding to a property measured by an actuator. Variables of this type will automatically be paired with the corresponding Actuator during emulation.

Parameters

- **value** – The initial value for this variable.
- **record** – Whether this variable should be recorded or not (WIP) TODO: record variables
- **sanity_check** – A callable which takes a single input value and returns a boolean. If this value is not None, the callable give will be invoked with the updated value of the semantic variable after every tick of the plant. If the callable returns false, the simulation will fail with an UnrecoverableState error.

class cleave.api.plant.**State**

Bases: *StateBase*, ABC

Abstract core class defining an interface for Plant state evolution over the course of a simulation. Implementing classes need to extend the advance() method to implement their logic, as this method will be called by the plant on each emulation time step.

abstract advance(*delta_t: float*) → None

Called by the plant on every time step to advance the emulation. Needs to be implemented by subclasses.

Parameters

delta_t – Time elapsed since the previous call to this method. This value will be 0 the first time this method is called.

initialize() → None

Called by the plant at the beginning of the emulation.

shutdown() → None

Called by the plant on shutdown.

class cleave.api.plant.**Sensor**(*prop_name: str*, *sample_freq: int*)

Bases: ABC

This class defines an interface for sensors attached to a simulated plant. Implementations should override the process_sample() method with their logic.

Parameters

- **prop_name** – Name of the property this sensor measures.
- **sample_freq** – Sampling frequency of this sensor.

property measured_property_name: str

Returns

Name of the property monitored by this sensor.

Return type

str

property sampling_frequency: int

Returns

Sampling frequency of this sensor, expressed in Hertz.

Return type
int

abstract process_sample(*value: Union[int, float, bool]*) → Union[int, float, bool]

Processes the measured value. This method should be implemented by subclasses to include sensor-specific behaviors.

Parameters
value – The latest measurement of the monitored property.

Returns
A possibly transformed value of the monitored property, according to the internal parameters of this sensor.

Return type
PhyPropType

class cleave.api.plant.**SimpleSensor**(*prop_name: str*, *sample_freq: int*)

Bases: *Sensor*

Simplest implementation of a sensor, which performs no processing on the read value and returns it as-is.

Parameters

- **prop_name** – Name of the property this sensor measures.
- **sample_freq** – Sampling frequency of this sensor.

process_sample(*value: Union[int, float, bool]*) → Union[int, float, bool]

Processes the measured value. This method should be implemented by subclasses to include sensor-specific behaviors.

Parameters
value – The latest measurement of the monitored property.

Returns
A possibly transformed value of the monitored property, according to the internal parameters of this sensor.

Return type
PhyPropType

class cleave.api.plant.**Actuator**(*prop_name: str*)

Bases: ABC

Abstract core class for actuators. Implementations should override the set_value() and get_actuation() methods with their logic.

Parameters
prop_name – Name of the property actuated upon by this actuator.

property actuated_property_name: str

Returns
Name of the property actuated upon by this actuator.

Return type
str

abstract set_value(*desired_value: Union[int, float, bool]*) → None

Called to set the target value for this actuator. This method should be implemented by extending classes.

Parameters
desired_value – Target value for this actuator.

abstract get_actuation() → Union[int, float, bool]

Returns the next value for the actuation processed governed by this actuator. This method should be implemented by extending classes.

Returns
A value for the actuated property.

Return type
PhyPropType

class cleave.api.plant.**SimpleConstantActuator**(*initial_value: Union[int, float, bool]*, *prop_name: str*)

Bases: *Actuator*

Implementation of a perfect actuator which keeps its value after being read (i.e. can be thought of as applying a constant force/actuation on the target variable).

Parameters
prop_name – Name of the property actuated upon by this actuator.

set_value(*desired_value: Union[int, float, bool]*) → None

Sets the value of the actuated property governed by this actuator.

Parameters
desired_value – The value of the actuated property.

get_actuation() → Union[int, float, bool]

Returns
The current value of the actuated property.

Return type
PhyPropType

class cleave.api.plant.**SimpleImpulseActuator**(*prop_name: str*, *default_value: Union[int, float, bool]*)

Bases: *Actuator*

Implementation if a perfect actuator which resets its value after being read (i.e. can be thought as an actuator which applies impulses to the target variable).

Parameters
prop_name – Name of the property actuated upon by this actuator.

set_value(*desired_value: Union[int, float, bool]*) → None

Sets the next value returned by this actuator.

Parameters
desired_value – Value returned in the next call to get_actuation().

get_actuation() → Union[int, float, bool]

Returns the internally stored value, and then resets it to the default value.

Returns
The actuation value.

Return type
PhyPropType

class cleave.api.plant.**GaussianConstantActuator**(*prop_name: str*, *g_mean: float*, *g_std: float*, *initial_value: Optional[Union[int, float, bool]] = None*, *prealloc_size: int = 1000000*)

Bases: *SimpleConstantActuator*

Implementation of an actuator with Gaussian noise in its output.

Parameters
prop_name – Name of the property actuated upon by this actuator.

set_value(*desired_value: Union[int, float, bool]*) → None

Sets the value of the actuated property governed by this actuator.

Parameters
desired_value – The value of the actuated property.

cleave.api.util module

cleave.api.util.**PhyPropType**

Type of properties that can be handled by sensors and actuators.

alias of Union[int, float, bool]

cleave.core package

Subpackages

cleave.core.client package

Submodules

cleave.core.client.actuator module

exception cleave.core.client.actuator.**RegisteredActuatorWarning**

Bases: Warning

exception cleave.core.client.actuator.**UnregisteredPropertyWarning**

Bases: Warning

class cleave.core.client.actuator.**ActuatorArray**(*actuators: Collection[*Actuator*]*, *control:* BaseControllerInterface)

Bases: *Recordable*

Internal utility class to manage a collection of Actuators attached to a Plant.

get_actuation_inputs() → Mapping[str, Union[int, float, bool]]

Fetches raw commands from the controller, processes them and returns.

Returns
A mapping from actuated property names to output values from the corresponding actuators.

Return type
PhyPropMapping

cleave.core.client.control module

class cleave.core.client.control.**BaseControllerInterface**

Bases: ABC

Defines the core interface for interacting with controllers.

abstract put_sensor_values(*prop_values: Mapping[str, Union[int, float, bool]]*) → None

Send a sample of sensor values to the controller.

Parameters
prop_values – Mapping from property names to sensor values.

abstract get_actuator_values() → Mapping[str, Union[int, float, bool]]

Waits for incoming data from the controller and returns a mapping from actuated property names to values.

Returns
Mapping from actuated property names to values.

Return type
Mapping

class cleave.core.client.control.**DummyControllerInterface**

Bases: *BaseControllerInterface*

put_sensor_values(*prop_values: Mapping[str, Union[int, float, bool]]*) → None

Send a sample of sensor values to the controller.

Parameters
prop_values – Mapping from property names to sensor values.

get_actuator_values() → Mapping[str, Union[int, float, bool]]

Waits for incoming data from the controller and returns a mapping from actuated property names to values.

Returns
Mapping from actuated property names to values.

Return type
Mapping

cleave.core.client.physicalsim module

class cleave.core.client.physicalsim.**PhysicalSimulation**(*state:* State, *tick_rate: int*)

Bases: *Recordable*

advance_state(*input_values: Mapping[str, Union[int, float, bool]]*) → Mapping[str, Union[int, float, bool]]

Performs a single step update using the given actuation values as inputs and returns the updated values for the sensed variables.

Parameters
input_values – Actuation inputs.

Returns
Mapping from sensed property names to values.

Return type
PhyPropMapping

cleave.core.client.plant module

class cleave.core.client.plant.**Plant**(*tick_dt: float*)

Bases: ABC

Interface for all plants.

class cleave.core.client.plant.**BasePlant**(*physim:* PhysicalSimulation, *sensors: Collection[*Sensor*]*, *actuators: Collection[*Actuator*]*, *control_interface:* BaseControllerInterface)

Bases: *Plant*

on_init() → None

Sets up the simulation of this plant

tick(*missed_count: int*) → None

Executes the emulation timestep. Intended use is inside a Twisted LoopingCall, hence why it takes a single integer parameter which specifies the number of calls queued up in a time interval (should be 1).

Parameters
missed_count –

on_shutdown() → None

Called on shutdown of the plant.

class cleave.core.client.plant.**CSVRecordingPlant**(*physim:* PhysicalSimulation, *sensors: Collection[*Sensor*]*, *actuators: Collection[*Actuator*]*, *control_interface:* BaseControllerInterface, *recording_output_dir: Path = PosixPath('.')*)

Bases: *BasePlant*

Plant with built-in CSV recording capabilities of metrics from the the physical properties and the network connection.

on_init() → None

Sets up the simulation of this plant

on_shutdown() → None

Called on shutdown of the plant.

cleave.core.client.sensor module

class cleave.core.client.sensor.**SensorArray**(*plant_tick_rate: int*, *sensors: Collection[*Sensor*]*, *control:* BaseControllerInterface)

Bases: *Recordable*

Internal utility class to manage a collection of Sensors attached to a Plant.

process_and_send_samples(*prop_values: Mapping[str, Union[int, float, bool]]*) → None

Processes measured properties by passing them to the internal collection of sensors and returns the processed values.

Parameters
prop_values – Dictionary containing mappings from property names to measured values.

Returns
A dictionary containing mappings from property names to processed sensor values.

Return type
Dict

exception cleave.core.client.sensor.**IncompatibleFrequenciesError**

Bases: Exception

exception cleave.core.client.sensor.**MissingPropertyError**

Bases: Exception

cleave.core.client.statebase module

class cleave.core.client.statebase.**BaseSemanticVariable**(*value: Any*, *record: bool = True*, *sanity_check: Optional[Callable[[Any], bool]] = None*)

Bases: SupportsFloat, SupportsInt, SupportsBytes, ABC

Base class for semantically significant variables in a State.

Parameters

- **value** – The initial value for this variable.
- **record** – Whether this variable should be recorded or not (WIP) TODO: record variables
- **sanity_check** – A callable which takes a single input value and returns a boolean. If this value is not None, the callable give will be invoked with the updated value of the semantic variable after every tick of the plant. If the callable returns false, the simulation will fail with an UnrecoverableState error.

class cleave.core.client.statebase.**StateBase**

Bases: ABC

Defines the underlying structure for the State interface without having to resort to hacks.

final **get_sensed_prop_names**() → Set[str]

Returns
Set containing the identifiers of the sensed variables.

Return type
Set

final **get_actuated_prop_names**() → Set[str]

Returns
Set containing the identifiers of the actuated variables.

Return type
Set

final **get_record_variables**() → Mapping[str, Union[int, float, bool]]

Returns
A mapping containing the values of the recorded variables in this state.

Return type
PhyPropMapping

final get_controller_parameters() → Mapping[str, Union[int, float, bool]]

Returns
A mapping from strings to values containing the initialization parameters for the controller associated with this physical simulation.

Return type
PhyPropMapping

final check_semantic_sanity() → Dict[str, Any]

Checks that the semantic variables in this state have acceptable values.

Returns
Returns a mapping containing the properties for which the check failed and their associated values.

Return type
Dict[str, Any]

cleave.core.client.timing module

class cleave.core.client.timing.**TimingResult**(*start*, *end*, *duration*, *results*)

Bases: tuple

Create new instance of TimingResult(start, end, duration, results)

start: float

Alias for field number 0

end: float

Alias for field number 1

duration: float

Alias for field number 2

results: Any

Alias for field number 3

class cleave.core.client.timing.**SimClock**

Bases: object

This class provides consistent simulation timing measurements.

get_sim_time() → float

Returns
A float representing the elapsed time in seconds since the initialization of this clock.

Return type
float

get_adjusted_realtime() → float

Returns
A float representing the elapsed time since the UNIX Epoch, adjusted for monotonicity.

Return type
float

time_subroutine(*fn: Callable[[Any], Any]*, **args*, ***kwargs*) → *TimingResult*

Times the execution of a function with respect to this clock.

Parameters

- **fn** – Function to be timed.
- **args** – Args to pass to the timed function.
- **kwargs** – Kwargs to pass to the timed function.

Returns

A TimingResult object containing the start and end timestamps of the function call, the duration of the function call in seconds and the results of the call.

Return type

TimingResult

class cleave.core.client.timing.**Rate**(*tick_count*, *interval_s*)

Bases: tuple

Create new instance of Rate(tick_count, interval_s)

tick_count: int

Alias for field number 0

interval_s: float

Alias for field number 1

class cleave.core.client.timing.**SimTicker**

Bases: object

Utility class to measure the rate of the plant and provide time deltas between ticks.

cleave.core.network package

Submodules

cleave.core.network.backend module

exception cleave.core.network.backend.**BusyControllerException**

Bases: Exception

class cleave.core.network.backend.**BaseControllerService**(*controller: Controller*, *add_delay_s: float = 0.0*)

Bases: *Recordable*, ABC

class cleave.core.network.backend.**UDPControllerService**(*controller: Controller*, *output_dir: Path*, *add_delay_s: float = 0.0*)

Bases: *BaseControllerService*, DatagramProtocol

UDP implementation of a controller service. Receives sensor samples over UDP and pushes them to the controller for processing.

startProtocol() → None

Called when a transport is connected to this protocol.

Will only be called once, even if multiple ports are connected.

stopProtocol() → None

Executed during shutdown.

datagramReceived(*in_dgram: bytes*, *addr: Tuple[str, int]*)

Executed on each datagram received.

cleave.core.network.client module

class cleave.core.network.client.**RecordingUDPControlClient**(*controller_addr: Tuple[str, int]*, *output_dir: Path*)

Bases: DatagramProtocol, ABC

startProtocol()

Called when a transport is connected to this protocol.

Will only be called once, even if multiple ports are connected.

stopProtocol()

Called when the transport is disconnected.

Will only be called once, after all ports are disconnected.

datagramReceived(*datagram: bytes*, *addr: Tuple[str, int]*)

Called when a datagram is received.

@param datagram: the bytes received from the transport. @param addr: tuple of source of datagram.

cleave.core.network.protocol module

class cleave.core.network.protocol.**ControlMsgType**(*value*)

Bases: Enum

An enumeration.

class cleave.core.network.protocol.**ControlMessage**(*msg_type: 'ControlMsgType'*, *seq: 'int'*, *timestamp: 'float'*, *payload: 'Any'*)

Bases: ABC

exception cleave.core.network.protocol.**NoMessage**

Bases: Exception

Submodules

cleave.core.config module

exception cleave.core.config.**ConfigError**

Bases: Exception

class cleave.core.config.**Config**(*config: Mapping[str, Any]*, *defaults: Mapping[str, Any] = {}*)

Bases: object

Helper class to hold configuration variables with optional default values.

get_parameter(*k: str*) → Any

Looks up and returns a named parameter from the configuration. If the parameter is not defined and not required, this will returned its default value instead.

Parameters
k – Parameter name.

Returns
The parameter value if defined. If the parameter is not defined but optional, this value will correspond to the parameter's default value.

Return type
Any

Raises
ConfigError – If the parameter is required and has not been defined.

class cleave.core.config.**ConfigFile**(*config_path: str*, *cmd_line_overrides: Mapping[str, Any] = {}*, *defaults: Mapping[str, Any] = {}*)

Bases: *Config*

Helper class to wrap access to a config.py file containing configuration variables for the program.

Parameters

- **config_path** – Path to the config script.
- **cmd_line_overrides** – Overrides for config variables obtained from the command line. Config parameters defined here will always override the config file.
- **defaults** – Mapping of fallback values for missing parameters.

cleave.core.logging module

class cleave.core.logging.**Logger**(*namespace: ~typing.Optional[str] = None*, *source: ~typing.Optional[object] = None*, *observer: ~typing.Optional[<InterfaceClass twisted.logger._interfaces.ILogObserver>] = None*)

Bases: `object`

A L{Logger} emits log messages to an observer. You should instantiate it as a class or module attribute, as documented in L{this module's documentation <twisted.logger>}.

@ivar namespace: the namespace for this logger @ivar source: The object which is emitting events via this logger @ivar observer: The observer that this logger will send events to.

@param namespace: The namespace for this logger. Uses a dotted
notation, as used by python modules. If not L{None}, then the name of the module of the caller is used.

@param source: The object which is emitting events via this
logger; this is automatically set on instances of a class if this L{Logger} is an attribute of that class.

@param observer: The observer that this logger will send events to.
If L{None}, use the L{global log publisher <globalLogPublisher>}.

emit(*level:* LogLevel, *format: Optional[str] = None*, ***kwargs: object*) → None

Emit a log event to all log observers at the given level.

@param level: a L{LogLevel} @param format: a message format using new-style (PEP 3101)

formatting. The logging event (which is a L{dict}) is used to render this format string.

@param kwargs: additional key/value pairs to include in the event.
Note that values which are later mutated may result in non-deterministic behavior from observers that schedule work for later execution.

failure(*format: str*, *failure: ~typing.Optional[~twisted.python.failure.Failure] = None*, *level: ~twisted.logger._levels.LogLevel = <LogLevel=critical>*, ***kwargs: object*) → None

Log a failure and emit a traceback.

For example:

```
try:
    frob(knob)
except Exception:
    log.failure("While frobbing {knob}", knob=knob)
```

or:

```
d = deferredFrob(knob)
d.addErrback(lambda f: log.failure("While frobbing {knob}",
                                   f, knob=knob))
```

This method is generally meant to capture unexpected exceptions in code; an exception that is caught and handled somehow should be logged, if appropriate, via L{Logger.error} instead. If some unknown exception occurs and your code doesn't know how to handle it, as in the above example, then this method provides a means to describe the failure in nerd-speak. This is done at L{LogLevel.critical} by default, since no corrective guidance can be offered to an user/administrator, and the impact of the condition is unknown.

@param format: a message format using new-style (PEP 3101) formatting.
The logging event (which is a L{dict}) is used to render this format string.

@param failure: a L{Failure} to log. If L{None}, a L{Failure} is
created from the exception in flight.

@param level: a L{LogLevel} to use. @param kwargs: additional key/value pairs to include in the event.

Note that values which are later mutated may result in non-deterministic behavior from observers that schedule work for later execution.

debug(*format: Optional[str] = None*, ***kwargs: object*) → None

Emit a log event at log level L{LogLevel.debug}.

@param format: a message format using new-style (PEP 3101) formatting.
The logging event (which is a L{dict}) is used to render this format string.

@param kwargs: additional key/value pairs to include in the event.
Note that values which are later mutated may result in non-deterministic behavior from observers that schedule work for later execution.

info(*format: Optional[str] = None*, ***kwargs: object*) → None

Emit a log event at log level L{LogLevel.info}.

@param format: a message format using new-style (PEP 3101) formatting.
The logging event (which is a L{dict}) is used to render this format string.

@param kwargs: additional key/value pairs to include in the event.
Note that values which are later mutated may result in non-deterministic behavior from observers that schedule work for later execution.

warn(*format: Optional[str] = None*, ***kwargs: object*) → None

Emit a log event at log level L{LogLevel.warn}.

@param format: a message format using new-style (PEP 3101) formatting.
The logging event (which is a L{dict}) is used to render this format string.

@param kwargs: additional key/value pairs to include in the event.
Note that values which are later mutated may result in non-deterministic behavior from observers that schedule work for later execution.

error(*format: Optional[str] = None*, ***kwargs: object*) → None

Emit a log event at log level L{LogLevel.error}.

@param format: a message format using new-style (PEP 3101) formatting.
The logging event (which is a L{dict}) is used to render this format string.

@param kwargs: additional key/value pairs to include in the event.
Note that values which are later mutated may result in non-deterministic behavior from observers that schedule work for later execution.

critical(*format: Optional[str] = None*, ***kwargs: object*) → None

Emit a log event at log level L{LogLevel.critical}.

@param format: a message format using new-style (PEP 3101) formatting.
The logging event (which is a L{dict}) is used to render this format string.

@param kwargs: additional key/value pairs to include in the event.
Note that values which are later mutated may result in non-deterministic behavior from observers that schedule work for later execution.

class cleave.core.logging.**LogLevel**

Bases: Names

Constants describing log levels.

@cvar debug: Debugging events: Information of use to a developer of the
software, not generally of interest to someone running the software unless they are attempting to diagnose a software issue.

@cvar info: Informational events: Routine information about the status of
an application, such as incoming connections, startup of a subsystem, etc.

@cvar warn: Warning events: Events that may require greater attention than
informational events but are not a systemic failure condition, such as authorization failures, bad data from a network client, etc. Such events are of potential interest to system administrators, and should ideally be phrased in such a way, or documented, so as to indicate an action that an administrator might take to mitigate the warning.

@cvar error: Error conditions: Events indicating a systemic failure, such
as programming errors in the form of unhandled exceptions, loss of connectivity to an external system without which no useful work can proceed, such as a database or API endpoint, or resource exhaustion. Similarly to warnings, errors that are related to operational parameters may be actionable to system administrators and should provide references to resources which an administrator might use to resolve them.

@cvar critical: Critical failures: Errors indicating systemic failure (ie.
service outage), data corruption, imminent data loss, etc. which must be handled immediately. This includes errors unanticipated by the software, such as unhandled exceptions, wherein the cause and consequences are unknown.

Classes representing constants containers are not intended to be instantiated.

The class object itself is used directly.

classmethod levelWithName(*name: str*) → NamedConstant

Get the log level with the given name.

@param name: The name of a log level.

@return: The L{LogLevel} with the specified C{name}.

@raise InvalidLogLevelError: if the C{name} does not name a valid log level.

cleave.core.recordable module

class cleave.core.recordable.**Recorder**(*recordable:* Recordable)

Bases: ABC

Interface for Recorder objects. Recorder and Recordable objects together implement the Observable pattern for text records of the plant state.

abstract notify(*latest_record: NamedTuple*)

Notify this recorder of the latest record generated by a recordable it is monitoring.

initialize()

Called on plant startup.

shutdown()

Called on plant shutdown.

flush()

Mainly intended for IO-backed recorders, to force flushing of any potentially memory-stored data.

class cleave.core.recordable.**Recordable**

Bases: ABC

Interface for Recordable objects. Recorder and Recordable objects together implement the Observable pattern for text records of the plant state.

class cleave.core.recordable.**NamedRecordable**(*name: str*, *record_fields: Sequence[str]*, *opt_record_fields: Mapping[str, Any] = {}*)

Bases: *Recordable*

A base implementation of a named recordable object.

class cleave.core.recordable.**CSVRecorder**(*recordable:* Recordable, *output_dir: Path*, *metric_name: str*, *chunk_size: int = 1000*)

Bases: *Recorder*

Implementation of a CSV-file backed Recorder.

initialize() → None

Called on plant startup.

flush() → None

Flushes the internal record buffer to the backing CSV file.

notify(*latest_record: NamedTuple*) → None

Notify this recorder of the latest record generated by a recordable it is monitoring.

shutdown() → None

Called on plant shutdown.

cleave.core.util module

class cleave.core.util.**SingleElementQ**

Bases: object

Utility class to hold a single variable in a thread-safe manner. Subsequent calls to put() without calling pop() will overwrite the stored variable. Conversely, pop() always returns the LATEST value for the stored variable.

put(*value: Any*) → None

Thread-safely store a value. Overwrites any previously stored value.

Parameters
value – The value to store in this container.

pop(*timeout: Optional[float] = None*) → Any

Pop the latest value for the stored variable. If timeout is None (the default), block until a value is available. Otherwise, block for up to timeout seconds, after which an Empty exception is raised if no value was made available within that time.

Parameters
timeout – Number of seconds to block for. If None, blocks indefinitely.

Returns
The stored value.

Return type
Any

Raises
Empty – If timeout is not None and no value is available when it runs out.

pop_nowait() → Any

Pops the latest value for the stored variable without waiting. If no value has been set yet, raises an Empty exception.

Returns
Latest value for the stored variable.

Return type
Any

Raises
Empty – If no value for the stored variable exists.

cleave.impl package

Submodules

cleave.impl.inverted_pendulum module

cleave.impl.inverted_pendulum.**G_CONST = Vec2d(0, -9.8)**

Gravity constants

cleave.impl.inverted_pendulum.**visualization_loop**(*input_q: Queue*, *shutdown_event: Event*, *window_w: int*, *window_h: int*, *caption: str*, *ppm: float = 200.0*) → None

Utility function that executes a Pyglet GUI loop to graphically visualize the inverted pendulum. Should always be called from a separate Process.

Parameters

- **input_q** – Input queue which holds dictionaries describing the figures to be drawn on screen.
- **shutdown_event** – Event to signal a shutdown of the Plant.
- **window_w** – Window width.
- **window_h** – Window height.
- **caption** – Window caption.
- **ppm** – Factor relating number of pixels per meter.

class cleave.impl.inverted_pendulum.**InvPendulumState**(*fail_angle_rad: float = 0.34*, *ground_friction: float = 0.1*, *cart_mass: float = 0.5*, *cart_dims: Vec2d = Vec2d(0.3, 0.2)*, *pend_length: float = 1.2*, *pend_mass: float = 0.2*, *pend_moment: float = 0.001*)

Bases: *State*

Implementation of a discrete-time simulation of an inverted pendulum.

Parameters

- **fail_angle_rad** – Angle at which the pendulum is considered unrecoverable. Negative values disable this check.
- **ground_friction** – Friction factor to apply for the ground.
- **cart_mass** – Mass of the pendulum cart in Kg.
- **cart_dims** – Dimensions of the cart in meters.
- **pend_length** – Length of the pendulum arm.
- **pend_mass** – Mass of the pendulum in Kg.
- **pend_moment** – Moment of the pendulum.

advance(*delta_t: float*) → None

Called by the plant on every time step to advance the emulation. Needs to be implemented by subclasses.

Parameters

delta_t – Time elapsed since the previous call to this method. This value will be 0 the first time this method is called.

class cleave.impl.inverted_pendulum.**InvPendulumStateWithViz**(*fail_angle_rad: float = 0.34*, *ground_friction: float = 0.1*, *cart_mass: float = 0.5*, *cart_dims: Vec2d = Vec2d(0.3, 0.2)*, *pend_length: float = 1.2*, *pend_mass: float = 0.2*, *pend_moment: float = 0.001*, *window_w: int = 1000*, *window_h: int = 700*, *caption: str = 'Inverted Pendulum Simulation'*, *ppm: float = 200.0*)

Bases: *InvPendulumState*

Implementation of a discrete-time simulation of an inverted pendulum, with a graphical visualization using Pyglet.

Parameters

- **fail_angle_rad** – Angle at which the pendulum is considered unrecoverable. Negative values disable this check.
- **ground_friction** – Friction factor to apply for the ground.
- **cart_mass** – Mass of the pendulum cart in Kg.
- **cart_dims** – Dimensions of the cart in meters.
- **pend_length** – Length of the pendulum arm.
- **pend_mass** – Mass of the pendulum in Kg.
- **pend_moment** – Moment of the pendulum.

initialize() → None

Called by the plant at the beginning of the emulation.

shutdown() → None

Called by the plant on shutdown.

advance(*delta_t: float*) → None

Called by the plant on every time step to advance the emulation. Needs to be implemented by subclasses.

Parameters
delta_t – Time elapsed since the previous call to this method. This value will be 0 the first time this method is called.

class cleave.impl.inverted_pendulum.**InvPendulumController**(*ref: float = 0.0*, *max_force: float = 25*)

Bases: *Controller*

Proportional-differential controller for the Inverted Pendulum.

Parameters

- **ref** – Position on the X-axis around which to balance the pendulum.
- **max_force** – Maximum force, in Newtons, allowed to apply to the pendulum.

K = [-57.38901804, -36.24133932, 118.51380879, 28.97241832]

Pendulum parameters

process(*sensor_values: Mapping[str, Union[int, float, bool]]*) → Mapping[str, Union[int, float, bool]]

Processes samples and produces a control command.

Samples arrive in the form of a dictionary of mappings from sensor property name to measured value. Actuator commands should be returned in the same fashion, in a dictionary of mappings from actuated property name to actuated value.

This method needs to be implemented by extending subclasses with their respective logic.

Parameters
sensor_values – A mapping of sensor property names to measured values.

Returns
A mapping of actuated property names to desired values.

Return type
PhyPropMapping

PYTHON MODULE INDEX

C

INDEX

N

O

P

R

S

T

U

V

W