EAGERx Documentation

Release 0.1.39

EAGERx Contributors

Sep 04, 2023

TABLE OF CONTENTS

You can use EAGERx (*Engine Agnostic Graph Environments for Robotics*) to easily define new (Gymnasium compatible) environments with modular robot definitions.

It enables users to:

- Define environments as graphs of nodes

- Visualize these graph environments interactively in a GUI

- Use a single graph environment both in reality and with various simulators

EAGERx explicitly addresses the differences in learning between simulation and reality, with native support for essential features such as:

- Safety layers and various other state, action and time-scale abstractions

- Delay simulation & domain randomization

- Real-world reset routines

- Synchronized parallel computation within a single environment

You can find the open-source code on Github.

Sim2Real: Policies trained in simulation and zero-shot evaluated on real systems using EAGERx. On the left the successful transfer of a box-pushing policy is shown, in the middle for the classic pendulum swing-up problem and on the right a task involving the crazyfly drone.

Modular: The modular design of EAGERx allows users to create complex environments easily through composition.

GUI: Users can visualize their graph environment. Here we visualize the graph environment that we built in this tutorial. See the documentation for more information.

VIDEO

ENGINES

EAGERx enables a unified pipeline for real-world and simulated learning across various simulators. The following engines/simulators are already available for training and evaluation:

- RealEngine for real-world experiments

- PybulletEngine for PyBullet simulations

- OdeEngine for simulations based on ordinary differential equations (ODEs)

Users can easily add their own engines by implementing the *Engine* interface.

2.1 Getting Started

2.1.1 Installing EAGERx

There are four installation options:

- Using *pip*

- From source

- Using docker

- Using conda and robostack

Installation using *pip*

You can do a minimal installation of `EAGERx` with:

```
pip3 install eagerx
```

Note: To make use of EAGERx's distributed capabilities (e.g. running on different physical machines), *ROS1* should be installed and sourced.

Installation from source

Prerequisites: Install *Poetry*.

Clone the eagerx repository and go to its root:

```
git clone git@github.com:eager-dev/eagerx.git
cd eagerx
```

Install EAGERx:

```
poetry install
```

Verify installation:

```
poetry run python examples/example_openai.py
```

Note: To make use of EAGERx's distributed capabilities (e.g. running on different physical machines), *ROS1* should be installed and sourced.

Installation using Docker (with distributed support)

Prerequisites: Install Docker and for GPU dockers nvidia-docker.

In total, four docker images are available with EAGERx installed, i.e. two with a minimal installation of EAGERx and its dependencies (CPU and GPU) and two with Stable Baselines 3 installed as well (CPU and GPU). The dockers with Stable Baselines 3 also come with tutorials on EAGERx.

Note: All docker images natively support EAGERx's distributed capabilities (e.g. running on different physical machines).

GPU Dockers

The GPU dockers require nvidia-docker and can be pulled as follows:

```
sudo docker pull eagerx/eagerx
```

or with Stable Baselines 3 and the tutorials:

```
sudo docker pull eagerx/eagerx-sb
```

The docker image can be run as follows:

```
sudo docker run -it --rm --gpus all [image]
```

where [image] should be replaced with *eagerx/eagerx* or *eagerx/eagerx-sb*.

Verify that EAGERx is installed:

```
python -c 'import eagerx'
```

CPU Dockers

The CPU only dockers can be pulled as follows:

```
sudo docker pull [image]
```

where image should be replaced with *eagerx/eagerx-cpu* or *eagerx/eagerx-sb-cpu*.

Run the image with the command

```
sudo docker run -it --rm [image]
```

where image should be replaced with *eagerx/eagerx-cpu* or *eagerx/eagerx-sb-cpu*.

Verify that EAGERx is installed:

```
python -c 'import eagerx'
```

Installation Using Conda (with distributed support)

You first need to download and install Conda (we recommend the miniforge distribution).

Then, follow the instructions of RoboStack to install ROS1:

```
# if you don't have mamba yet, install it first (not needed when using mambaforge):
conda install mamba -c conda-forge

# now create a new environment
mamba create -n ros_env python=3.8
conda activate ros_env

# this adds the conda-forge channel to the new created environment configuration
conda config --env --add channels conda-forge
# and the robostack channels
conda config --env --add channels robostack
conda config --env --add channels robostack-experimental

# Install the version of ROS you are interested in:
mamba install ros-noetic-desktop

# optionally, install some compiler packages if you want to e.g. build packages in a
→colcon_ws:
mamba install compilers cmake pkg-config make ninja colcon-common-extensions

# on Linux and osx (but not Windows) for ROS1 you might want to:
mamba install catkin_tools

# on Windows, install Visual Studio 2017 or 2019 with C++ support
# see https://docs.microsoft.com/en-us/cpp/build/vscpp-step-0-installation?view=msvc-160

# on Windows, install the Visual Studio command prompt:
# mamba install vs2019_win-64

# note that in this case, you should also install the necessary dependencies with conda/
```

(continues on next page)

```
↪mamba, if possible

# IMPORTANT! reload environment to activate required scripts before running anything
# on Windows, please restart the Anaconda Prompt / Command Prompt!
conda deactivate
conda activate ros_env

# if you want to use rosdep, also do:
mamba install rosdep
rosdep init  # IMPORTANT: do not use sudo!
rosdep update
```

Finally, you can activate your `ros_env` and install EAGERx:

```
conda activate ros_env
pip install eagerx
```

We also provide a Conda environment file which contains ROS1, EAGERx, SB3 and other EAGERx packages. In that case you simply have to do:

```
conda env create -f ros_env.yml
```

2.1.2 Extras: GUI

To install the whole set of features, you will need additional packages. There is for example a package available for visualizing the `Graph` and the `EngineGraph`.

You can install the gui by running:

```
pip3 install eagerx-gui
```

Note: The EAGERx docker images currently do not support gui functionality.

Fig. 1: The visualisation of an environment via the GUI.

2.1.3 Extras: training visualization

In robotics it is crucial to monitor the robot's behavior during the learning process. Luckily, all inter-node communication within EAGERx can be listened to externally, so that any relevant information stream can be trivially monitored on-demand (e.g. with `rqt_plot`). For this, the user must select the `Ros1` *Backend*.

Note: `rqt_plot` is included in the `desktop` or `desktop-full` ROS1 installation. See *here* for installation instructions. The docker images do not support visualization using `rqt_plot`.

Fig. 2: Live plot of the x, y, and z coordinate of the end effector using `rqt_plot`.

2.1.4 Other Dependencies

Below you find instructions for installing dependencies (optionally) required by EAGERx.

Poetry

Poetry is a tool for dependency management and packaging in Python. It allows you to declare the libraries your project depends on and it will manage (install/update) them for you. We advise contributors to use this tool when developing an EAGERx package to leverage the pre-build CI workflow we have setup in the template package. However, this is **not** a requirement and a simple *pip install* to install all eagerx package dependencies into your project's (virtual) Python environment will also work.

For installation on osx / linux / bashonwindows, simply run:

```
curl -sSL https://raw.githubusercontent.com/python-poetry/poetry/master/get-poetry.py |␣
→python -
```

For more installation instructions, see here.

ROS1

See the ROS1 Installation Options, or do the following. By replacing <DISTRO> with the supported ROS1 distributions (`noetic`, `melodic`), and <PACKAGE> with the installation type (`ros-base`, `desktop`, `desktop-full`), a minimal ROS1 installation can be installed with:

> **Warning:** Currently, eagerx only supports ROS1. ROS2 support will be added in future versions.

```
sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_release -sc) main" > /etc/
→apt/sources.list.d/ros-latest.list'
sudo apt install curl # if you haven't already installed curl
curl -s https://raw.githubusercontent.com/ros/rosdistro/master/ros.asc | sudo apt-key␣
→add -
sudo apt update
sudo apt install ros-<DISTRO>-<PACKAGE>
sudo apt-get install ros-<DISTRO>-cv-bridge
```

Make sure to source /opt/ros/<DISTRO>/setup.bash in the environment where you intend to `eagerx` in. It can be convenient to automatically source this script every time a new shell is launched. These commands will do that for you if you:

```
echo "source /opt/ros/<DISTRO>/setup.bash" >> ~/.bashrc
source ~/.bashrc
```

In case you make use of a virtual environment, move to the directory containing the `.venv` and add `source /opt/ros/<DISTRO>/setup.bash` to the activation script before activating the environment with this line:

```
echo "source /opt/ros/<DISTRO>/setup.bash" >> .venv/bin/activate
```

2.2 Tutorials

A set of tutorials is created to showcase some of the key features of EAGERx and to guide users through the process of using EAGERx for robot learning tasks. Most of them are available in the form of Google Colabs in the eagerx_tutorials package. We will briefly introduce these tutorials in the following sections. Furthermore, a tutorial on how to visualize training using EAGERx is available.

2.2.1 Colabs

Introduction to EAGERx

The best way to get introduced to EAGERx is to play around with the tutorials that are available. They also contain exercises that address common challenges of robotic reinforcement learning and how to overcome them using EAGERx.

The following introductory tutorials are available:

- Tutorial 1: Getting started
- Tutorial 2: Advanced usage

The solutions are available in here.

Fig. 3: In the advanced usage tutorial you will learn a quadruped to walk in circles within four minutes of training.

1. Getting Started

This tutorial covers:

- constructing a `Graph` and an environment using `BaseEnv`,
- switching between different `Engine`,
- performing domain randomization.

2. Advanced Usage

In this notebook, you will learn to use EAGERx to create a gym-compatible environment. This tutorial covers:

- how to initialize a robot (Go 1 Quadruped Robot).
- how to add pre-processing nodes (i.e. low-level controllers).
- how to fine-tune low-level controllers to achieve the desired behavior.
- how to (de)select various sensors to investigate its effect on the learning performance.

Developer Tutorials

Next to the introduction tutorials, a set of developer tutorials is also available:

- Tutorial 1: Environment Creation and Training with EAGERx
- Tutorial 2: Reset and Step
- Tutorial 3: Space and Processors
- Tutorial 4: Nodes and Graph Validity
- Tutorial 5: Adding Engine Support for an Object
- Tutorial 6: Defining a new Object
- Tutorial 7: More Informative Rendering
- Tutorial 8: Reset Routines

The solutions are available in here.

Fig. 4: The tutorials cover common challenges of robotic reinforcement learning and how to overcome them using EAGERx. The classic control problem of swinging up an underactuated pendulum is used as an example.

1. Environment Creation and Training

This tutorial covers:

- Creating a `Graph` with an `Object`.
- How to use this `Graph` and a `Engine` to create an `BaseEnv`.
- How to train a policy with the `BaseEnv`.

2. Reset and Step

This tutorial covers:

- Extracting observations in the `step`
- Resetting states using `reset()`
- The *window* argument of the `connect()` method
- Simulating delays using the *delay* argument of the `connect()` method

3. Space and Processors

This tutorial covers:

- How to specify a *Space*
- Creating a custom *Processor*
- How to add a *Processor*

4. Nodes and Graph Validity

This tutorial covers:

- Creating a *Node*
- Adding a *Node* to the *Graph*
- Checking the validity of the *Graph*
- How to make the *Graph* valid (DAG)

5. Adding Engine Support for an Object

This tutorial covers:

- Adding an engine-specific implementation to an *Object*
- Initializing the corresponding *Engine*
- Train with the newly added engine-specific implementation

6. Defining a new Object

This tutorial covers:

- Defining a new *Object*

7. More Informative Rendering

- Create a layover *Node* that augments a raw image sensors
- Connect the layover *Node* and use it for rendering
- Demonstrate that rendering is agnostic to the selected physics-engine

8. Reset Routines

- Defining the reset routine with a *ResetNode*
- Reset the *Object*'s with the reset routine.

2.2.2 Visualizing your environment

In this tutorial we will demonstrate how you can use EAGERx to visualize parts of your environment.

EAGERx has a built-in GUI to visualize your environment. Moreover, as EAGERx is build on top of ROS you can use many of the support ROS tools. These tools can give valuable insights on the workings of your environment.

Note: The ROS tools we cover in this tutorial (e.g. `rqt_plot`) are per default included in the `desktop` and `desktop-full` ROS installation.

The tools can be manually installed with the lines below. Replace <DISTRO> with the supported ROS distributions (`noetic`, `melodic`).

```
sudo apt-get install ros-<DISTRO>-rqt
sudo apt-get install ros-<DISTRO>-rqt-common-plugins
```

Graphical user interface

After creating the *Graph* for our environment, we can inspect it using the GUI. Note that we need to install it first if you haven't done so yet:

```
pip install eagerx-gui
```

Next, we can open it by calling *gui()*:

```
graph.gui()
```

By clicking on *Show Graph*, we can inspect the graph in the GUI. The output you will see should look something like this:

The GUI also provides functionalities for constructing a *Graph*. So we could also have created the exact same *Graph* from scratch using the GUI.

This is demonstrated in the video below:

Live-plotting

Note: Live-plotting is currently only supported when the `Ros1` *Backend* is selected.

In robotics it is crucial to monitor the robot's behavior during the learning process. Luckily, inter-node communication within EAGERx can always be listened to externally, so that any relevant information stream can be trivially monitored on-demand.
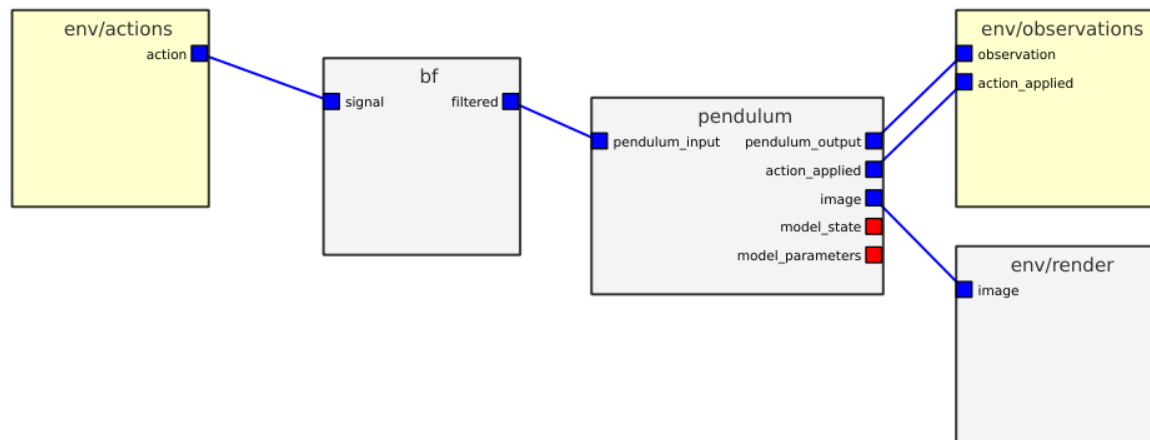
Fig. 5: Screenshot of the EAGERx GUI.

Fig. 6: The construction of an environment via the GUI.

Inter-node communication within EAGERx is always advertised as a topic that can be listened to externally, even when nodes are interconnected within the same process. Therefore, we can leverage existing tools from ROS such as `rqt_plot`. `rqt_plot` provides a GUI plugin visualizing numeric values in a 2D plot using different plotting backends. See here for more details on this tool.

Topic addresses for outputs follow the naming convention:

- `<env_name>/<node_name>/outputs/<cname>`: (e.g. *rx/controller/outputs/reference*).

```
rqt_plot /rx/viper/sensors/ee_pos/data[0]:data[1]:data[2]
```

This will open a live-plot of the x, y, and z coordinate of the end effector similar to the one below.

Fig. 7: Live plot of the x, y, and z coordinate of the end effector using `rqt_plot`.

Note: The computational overhead of publishing all node outputs as topics is minimal when there are no subscribers. In other words, there is only computational overhead when external source (e.g. rqt_plot) is listening to the advertised topics. Once the external source unsubscribes, the overhead is again reduced.

Computation graph

`rqt_graph` is a ROS tool that provides a GUI plugin for visualizing what's going in the ROS computation graph that EAGERx creates for you based on the nodes, objects, and their interconnections.

To visualize the graph, you can run the following command in a separate terminal while your environment is running:

```
rosparam set enable_statistics true
rqt_graph
```

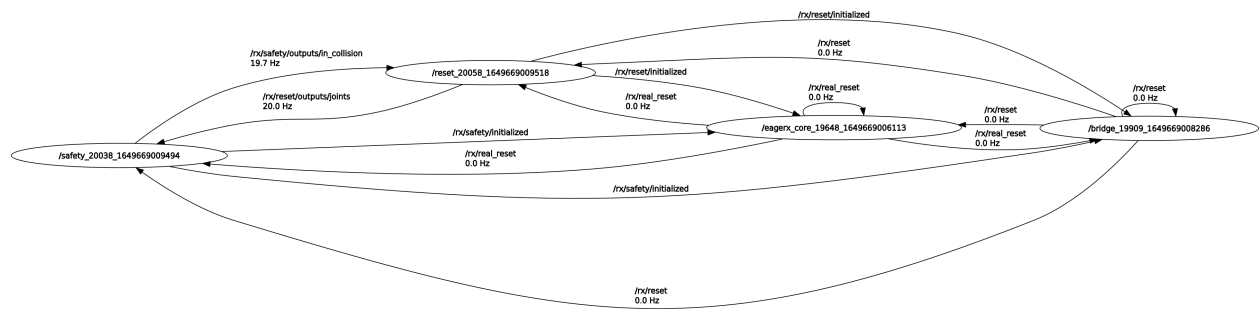This will provide you with an overview similar to the one below:

Fig. 8: The ROS computation graph that EAGERx creates for you.

In the top left, you can refresh to update statistics about the messages that are passed in the graph. Also you can select what to visualize:

- *Nodes only*: This will only show the communication (i.e. topics) between nodes that were launched as a `NEW_PROCESS`.

- *Nodes/Topics (active)*: This will show all communication (i.e. topics) that are currently active.

- *Nodes/Topics (all)*: This will show all communication (i.e. topics).

2.2.3 Distributed

To launch a node or engine externally on, for example, a different physical machine, you must set its process to
`EXTERNAL`. See *process* for more info. In this case, you as a user are responsible for launching the node/engine.

Note: When using the `Ros1` *Backend* for running across multiple machines, please make sure that the
`ROS_MASTER_URI` is correctly configured on every machine. See here for more info.

You will have to pass the following arguments

- Path to the appropriate executable python script (`executable_node.py` for nodes, `executable_engine.py`
  for engines).

- `--backend`: Backend that was selected for the environment (e.g. `eagerx.backends.ros1/Ros1` or `eagerx.`
  `backends.single_process/SingleProcess`).

- `--loglevel`: The desired log level (as an integer). See `constants` for more info.

- `--env`: The environment name.

- `--name`: The name of the node/engine. For engines, the name is always `engine`. If the node is part of an
  engine-specific implementation of an object, the node name is `<object_name>/<node_name>`.

For nodes, an example would look like:

```
python3 <path>/<to>/<package>/eagerx/core/executable_node.py --backend eagerx.backends.
→ros1/Ros1 --loglevel 20 --env CamEnv --name obj/camera_api
```

For an engine, an example would look like:

```
python3 <path>/<to>/<package>/eagerx/core/executable_engine.py --backend eagerx.backends.
→ros1/Ros1 --loglevel 20 --env CamEnv --name engine
```

2.3 API Reference

2.3.1 Engine

class eagerx.core.entities.**Engine**(*sync*, *real_time_factor*, *params*, *target_addresses*, *node_names*, **args*,
**kwargs)

> Baseclass for engines.
>
> Use this baseclass to implement an engine that interfaces the simulator.
>
> Users must call *make()* to make the engine subclass' specification.
>
> Subclasses must implement the following methods:
>
> - *make()*
> - *initialize()*
> - *add_object()*
> - *pre_reset()*
> - *reset()*
> - *callback()*

- *shutdown()* (optional)

abstract add_object(*name*, **args*, ***kwargs*)

> Adds an object to the simulator that is interfaced by the engine.
>
> > **Parameters**
> >
> > - **name** (str) – The name of the `Object` that is to be added.
> >
> > - **args** (Union[bool, int, float, str, List, Dict]) – The engine-specific parameters that are required to add the `Object`.
> >
> > - **kwargs** (Union[bool, int, float, str, List, Dict]) – The engine-specific parameters that are optional to add the `Object`.
> >
> > **Return type**
> >   None

abstract callback(*t_n*)

> The engine callback that is performed at the specified rate.
>
> This callback is steps the simulator by 1/*rate*.
>
> ---
>
> **Note:** The engine does not have any outputs. If you wish to broadcast other output messages based on properties of the simulator, a separate `EngineNode` should be created.
>
> ---
>
> > **Parameters**
> >   **t_n** (float) – Time passed (seconds) since last reset. Increments with 1/*rate*.
> >
> > **Return type**
> >   None

classmethod info(*method=None*)

> A helper method to get info on a method of the specified subclass.
>
> > **Parameters**
> >   **method** (Union[List[str], str, None]) – The registered method we would like to receive info on. If no method is specified, it provides info on the class itself.
> >
> > **Return type**
> >   str
> >
> > **Returns**
> >   Info on the subclass' method.

abstract initialize(*spec*)

> An abstract method that initializes the node at run-time.
>
> > **Parameters**
> >   **spec** (Union[*NodeSpec*, *EngineSpec*, *ResetNodeSpec*]) – Specification of the node/engine.
> >
> > **Return type**
> >   None

abstract classmethod make(**args*, ***kwargs*)

> An abstract method that makes the specification (also referred to as *spec*) of this entity.
>
> > **Parameters**

- **args** (Any) – Arguments to the subclass' make function.

- **kwargs** (Any) – Optional Arguments to the subclass' make function.

> **Returns**
> A (mutable) spec that can be used to build and subsequently initialize the entity (e.g. node, engine, . . .).

abstract **pre_reset**(***states*)

An abstract method that resets the engine to its initial state before the start of an episode.

Note: This method is called **before** every *EngineNode* and *EngineState* has performed its reset, but after all reset routines, implemented with *ResetNode*, have reached their target.

- Can be useful for performing some preliminary actions on the simulator such as pausing before resetting every *EngineNode* and *EngineState*.

- Reset the simulator state so that this state can be used in the reset of every *EngineNode* and *EngineState*.

> **Parameters**
> **states** (Any) – States that were registered (& selected) with the *eagerx.core.register.states()* decorator by the subclass. The state messages are sent by the environment and can be used to reset the engine at the start of an episode. This can be anything, such as the dynamical properties of the simulator (e.g. friction coefficients).
>
> **Return type**
> None

abstract **reset**(***states*)

An abstract method that resets the engine to its initial state before the start of an episode.

This method should be decorated with *eagerx.core.register.states()* to register the states.

Note: This method is called **after** every *EngineNode* and *EngineState* has finished its reset.

- Can be useful for performing some final actions on the simulator such as unpausing after every *EngineNode* and *EngineState* have reset.

> **Parameters**
> **states** (Any) – States that were registered (& selected) with the *eagerx.core.register.states()* decorator by the subclass. The state messages are sent by the environment and can be used to reset the engine at the start of an episode. This can be anything, such as the dynamical properties of the simulator (e.g. friction coefficients).
>
> **Return type**
> None

shutdown()

A method that can be overwritten to cleanly shutdown (e.g. release resources).

> **Return type**
> None

backend: *Backend*

> Responsible for all I/O communication within this process. Nodes inside the same process share the same message broker. Cannot be modified.

entity_id: str

> A unique entity_id with the structure <module>/<classname>.

log_level: int

> Specifies the log level for this node: {0: SILENT, 10: DEBUG, 20: INFO, 30: WARN, 40: ERROR, 50: FATAL}. Can be set in the subclass' `spec()`.

log_memory: int

> Specifies the log level for logging memory usage over time for this node: {0: SILENT, 10: DEBUG, 20: INFO, 30: WARN, 40: ERROR, 50: FATAL}. Note that *log_level* has precedent over the memory level set here. Can be set in the subclass' `spec()`.

name: str

> User specified node name. Can be set in `spec()`.

ns: str

> Namespace of the environment. Can be set with the *name* argument to *BaseEnv*.

objects: dict

> Parameters for all objects.

process: int

> Process in which this node is launched. See *process* for all options. Can be set in the subclass' `spec()`.

rate: float

> Rate (Hz) at which the callback is called. Can be set in the subclass' `spec()`.

real_time_factor: float

> A specified upper bound on the real_time factor. Wall-clock rate=real_time_factor*rate. If real_time_factor < 1 the simulation is slower than real time. Can be set in the engine's `spec()`.

simulate_delays: bool

> Flag that specifies whether input delays are simulated. You probably want to set this to False when running in the real-world. Can be set in the engine's `spec()`.

simulator: Any

> The simulator object. The simulator depends on the engine and should be initialized in the *initialize()* method. Oftentimes, engine nodes require a reference in *callback()* and/or *reset()* to this simulator object to simulate (e.g. apply an action, extract a sensor measurement). Engine nodes only have this reference if the node was launched inside the engine process. See *process* for more info.

states: dict

> Parameters for all selected states.

sync: bool

> Flag that specifies whether we run reactive or asynchronous. Can be set in the engine's `spec()`.

2.3.2 Backend

class eagerx.core.entities.**Backend**(*ns*, *backend_type*, *entity_id*, *log_level*, *main=False*, *sync=None*,
real_time_factor=None, *simulate_delays=None*, ***kwargs*)

Baseclass for backends.

Use this baseclass to implement backends that implement the communication.

Users must use *make()* to make the registered subclass' specification.

Subclasses must implement the following methods:

- *make()*
- *initialize()*
- *Publisher()*
- *Subscriber()*
- *register_environment()*
- *delete_param()*
- *upload_params()*
- *get_param()*
- *spin()*

Subclasses must set the following static class properties:

- *BACKEND*
- *DISTRIBUTED_SUPPORT*
- *MULTIPROCESSING_SUPPORT*
- *COLAB_SUPPORT*

abstract Publisher(*address*, *dtype*)

Creates a publisher.

> **Parameters**
>
> - **address** (str) – Topic name.
>
> - **dtype** (str) – Dtype of message in string format (e.g. *float32*).
>
> **Return type**
> Publisher

abstract Subscriber(*address*, *dtype*, *callback*, *header=False*, *callback_args=()*)

Creates a subscriber.

> **Parameters**
>
> - **address** (str) – Topic name.
>
> - **dtype** (str) – Dtype of message in string format (e.g. *float32*).
>
> - **callback** – Function to call (fn(data)) when data is received. If callback_args is set, the function must accept the callback_args as positional args, i.e. fn(data, header, *callback_args).
>
> - **header** (bool) – Set to True if the callback accepts the header as the second positional argument.

- **callback_args** (Optional[Tuple]) – Additional arguments to pass to the callback.

>   **Return type**
>   Subscriber

abstract **delete_param**(*param*, *level=1*)

>   Deletes params from the parameter server.

>   **Parameters**

> - **param** (str) – Parameter name.

> - **level** (int) – Determines what to do when the param does not exist:

>   - 0=error: Raises a BackendException.

>   - 1=warn: logs a warning and returns *None*.

>   - 2=pass: passes silently and returns *None*.

>   **Return type**
>   None

static **deserialize_time**(*secs*, *nsecs*)

>   Convert a secs and nsecs time instance into float time in seconds .

>   Should be used when manually setting secs/nsecs slot values for deserialization.

>   **Return type**
>   float

abstract **get_param**(*name*, *default=<eagerx.core.constants.Unspecified object>*)

>   Retrieve a parameter from the param server

>   **Parameters**

> - **name** (str) – Parameter name.

> - **default** (Any) – Default value to return.

>   **Return type**
>   Union[Dict, List, bool, float, int, str]

classmethod **info**(*method=None*)

>   A helper method to get info on a method of the specified subclass.

>   **Parameters**
>   **method** (Union[List[str], str, None]) – The registered method we would like to receive
>   info on. If no method is specified, it provides info on the class itself.

>   **Return type**
>   str

>   **Returns**
>   Info on the subclass' method.

abstract **initialize**(*spec*)

>   An abstract method to initialize the backend.

>   **Parameters**
>   **spec** (*BackendSpec*) – Specification of the node/engine.

>   **Return type**
>   None

abstract classmethod make(*args, **kwargs)

An abstract method that makes the specification (also referred to as *spec*) of this entity.

Parameters

- **args** (Any) – Arguments to the subclass' make function.

- **kwargs** (Any) – Optional Arguments to the subclass' make function.

Returns

A (mutable) spec that can be used to build and subsequently initialize the entity (e.g. node, engine, . . .).

now()

Get the current times according to the simulated and wall clock

Return type

Tuple[float, float]

abstract register_environment(*name*, *force_start*, *fn*)

Checks if environment already exists and shuts it down if *force_restart* is set. Then, it registers the remote shutdown procedure for the newly created environment.

Parameters

- **name** (str) – Environment name (i.e. namespace of the environment).

- **force_start** (bool) – Whether to shutdown any environment with the same name if it already exists.

- **fn** (Callable) – Function with zero args to be called on remote shutdown.

Return type

ShutdownService

static serialize_time(*t*)

Convert a float time instance (in seconds) into secs and nsecs.

Should be used when manually setting secs/nsecs slot values for serialization.

Return type

Tuple[int, int]

shutdown()

Shuts down the backend

Return type

None

abstract spin()

Blocks until node is shutdown. Yields activity to other threads.

Return type

None

abstract upload_params(*ns*, *values*, *verbose=False*)

Upload params to the parameter server.

Parameters

- **ns** (str) – Namespace to load parameters into, str.

- **values** (Dict[str, Union[Dict, List, bool, float, int, str]]) – Key/value dictionary, where keys are parameter names and values are parameter values, dict.

> • **verbose** (bool) – Verbosity level.

> **Return type**
>> None

abstract property BACKEND: str

> Backend name in string format.

abstract property COLAB_SUPPORT: bool

> Whether the backend supports running on Google colab.

abstract property DISTRIBUTED_SUPPORT: bool

> Whether nodes can be launched on external platforms (i.e. distributed communication).

abstract property MULTIPROCESSING_SUPPORT: bool

> Whether nodes can be launched as separate processes.

backend_type: str

> The class definition of the subclass. Follows naming convention *<module>/<BackendClassName>*. Cannot be modified.

entity_id: str

> A unique entity_id with the structure <module>/<classname>.

log_level: int

> Specifies the effective log level: {0: SILENT, 10: DEBUG, 20: INFO, 30: WARN, 40: ERROR, 50: FATAL}. Can be set in the subclass' `spec()`.

main: bool

> If True, the backend is the 'main` backend that corresponds to the environment process.

ns: str

> Namespace of the environment. Can be set with the *name* argument to *BaseEnv*.

real_time_factor: float

> A specified upper bound on the real_time factor. Wall-clock rate=real_time_factor*rate. If real_time_factor < 1 the simulation is slower than real time.

simulate_delays: bool

> Flag that specifies whether input delays are simulated. You probably want to set this to False when running in the real-world.

sync: bool

> Flag that specifies whether we run synchronous or asynchronous.

2.3.3 Processor

class eagerx.core.entities.**Processor**

> Baseclass for processors.

> Use this baseclass to implement processor that preprocess an input/output message.

> This baseclass only supports one-way processing.

> Users must call *make()* to make the subclass' specification.

> Subclasses must implement the following methods:

> • *make()*

- *initialize()*

- *convert()*

abstract convert(*msg*)

An abstract method to preprocess messages.

> **Parameters**
> > **msg** (Any) – Raw message.
>
> **Return type**
> > Any
>
> **Returns**
> > Preprocessed message.

classmethod info(*method=None*)

A helper method to get info on a method of the specified subclass.

> **Parameters**
> > **method** (Union[List[str], str, None]) – The registered method we would like to receive info on. If no method is specified, it provides info on the class itself.
>
> **Return type**
> > str
>
> **Returns**
> > Info on the subclass' method.

abstract initialize(*spec*)

An abstract method to initialize the processor.

> **Parameters**
> > **spec** (*ProcessorSpec*) – Specification of the processor.
>
> **Return type**
> > None

abstract classmethod make(**args*, ***kwargs*)

An abstract method that makes the specification (also referred to as *spec*) of this entity.

> **Parameters**
> - **args** (Any) – Arguments to the subclass' make function.
> - **kwargs** (Any) – Optional Arguments to the subclass' make function.
>
> **Returns**
> > A (mutable) spec that can be used to build and subsequently initialize the entity (e.g. node, engine, ...).

2.3.4 Engine State

class eagerx.core.entities.**EngineState**(*ns*, *name*, *simulator*, *backend*, *params*)

Baseclass for engine states.

Use this baseclass to implement engine states for an *Object*.

Users must call *make()* to make the subclass' specification.

Subclasses must implement the following methods:

- *make()*

- *initialize()*

- *reset()*

classmethod info(*method=None*)

> A helper method to get info on a method of the specified subclass.
>
> > **Parameters**
> > > **method** (Union[List[str], str, None]) – The registered method we would like to receive info on. If no method is specified, it provides info on the class itself.
> >
> > **Return type**
> > > str
> >
> > **Returns**
> > > Info on the subclass' method.

abstract initialize(*spec*, *simulator*)

> An abstract method to initialize the engine state.
>
> > **Parameters**
> >
> > - **spec** (*EngineStateSpec*) – The engine state specification.
> >
> > - **simulator** (Any) – A reference to the engine's simulator.
> >
> > **Return type**
> > > None

abstract classmethod make(**args*, ***kwargs*)

> An abstract method that makes the specification (also referred to as *spec*) of this entity.
>
> > **Parameters**
> >
> > - **args** (Any) – Arguments to the subclass' make function.
> >
> > - **kwargs** (Any) – Optional Arguments to the subclass' make function.
> >
> > **Returns**
> > > A (mutable) spec that can be used to build and subsequently initialize the entity (e.g. node, engine, . . .).

abstract reset(*state*)

> An abstract method to reset the engine state of an *Object*.
>
> > **Parameters**
> > > **state** (Any) – The desired state that the user can specify before calling *reset()*.
> >
> > **Return type**
> > > None

backend

> Responsible for all I/O communication within this process.

name

> Name of the state.

ns

> Namespace of the environment. Can be set with the *name* argument to *BaseEnv*.

2.3.5 Nodes

Node

class eagerx.core.entities.**Node**(*ns*, *message_broker*, *sync*, *real_time_factor*, *simulate_delays*, *params*, *call_init=True*)

>   Baseclass for nodes.
>
>   Use this baseclass to implement nodes that will be added to the (agnostic) *Graph*.
>
>   Users must call *make()* to make the node subclass' specification.
>
>   Subclasses must implement the following methods:
>
>   - *make()*
>
>   - *initialize()*
>
>   - *reset()*
>
>   - *callback()*
>
>   - *shutdown()* (optional)
>
>   Use baseclass *EngineNode* instead, for nodes that will be added to *EngineGraph* when specifying an engine implementation for an *Object*.
>
>   Use baseclass *ResetNode* instead, for reset routines.
>
>   abstract **callback**(*t_n*, ***inputs*)
>
>   >   An abstract method that is called at the specified node rate.
>   >
>   >   This method should be decorated with:
>   >
>   >   - *eagerx.core.register.inputs()* to register the inputs.
>   >
>   >   - *eagerx.core.register.outputs()* to register the outputs.
>   >
>   >   **Parameters**
>   >
>   >   >   - **t_n** (float) – Time passed (seconds) since last reset. Increments with 1/*rate*.
>   >   >
>   >   >   - **inputs** (*Msg*) – Inputs that were registered (& selected) with the *eagerx.core.register.inputs()* decorator by the subclass.
>   >
>   >   **Return type**
>   >
>   >   >   Dict[str, Any]
>   >
>   >   **Returns**
>   >
>   >   >   Dictionary with outputs that were registered (& selected) with the *eagerx.core.register.outputs()* decorator by the subclass.
>
>   classmethod **info**(*method=None*)
>
>   >   A helper method to get info on a method of the specified subclass.
>   >
>   >   **Parameters**
>   >
>   >   >   **method** (Union[List[str], str, None]) – The registered method we would like to receive info on. If no method is specified, it provides info on the class itself.
>   >
>   >   **Return type**
>   >
>   >   >   str

> **Returns**
>> Info on the subclass' method.

abstract **initialize**(*spec*)

> An abstract method that initializes the node at run-time.

>> **Parameters**
>>> **spec** (Union[*NodeSpec*, *EngineSpec*, *ResetNodeSpec*]) – Specification of the node/engine.

>> **Return type**
>>> None

abstract classmethod **make**(**args*, ***kwargs*)

> An abstract method that makes the specification (also referred to as *spec*) of this entity.

>> **Parameters**

>>> • **args** (Any) – Arguments to the subclass' make function.

>>> • **kwargs** (Any) – Optional Arguments to the subclass' make function.

>> **Returns**
>>> A (mutable) spec that can be used to build and subsequently initialize the entity (e.g. node, engine, …).

abstract **reset**(***states*)

> An abstract method that resets the node to its initial state before the start of an episode.

> This method should be decorated with *eagerx.core.register.states()* to register the states.

>> **Parameters**
>>> **states** (Any) – States that were registered (& selected) with the *eagerx.core.register.states()* decorator by the subclass. The state messages are sent by the environment and can be used to reset the node at the start of an episode. This can be anything from an estimator's initial state to a hyper-parameter (e.g. delay, control gains).

>> **Return type**
>>> None

set_delay(*delay*, *component*, *cname*)

> A method to vary the delay of an input or feedthrough.

>> **Parameters**

>>> • **delay** (float) – A non-negative delay that can be varied at the beginning of an episode (during the reset procedure).

>>> • **component** (str) – Either "inputs" or "feedthroughs".

>>> • **cname** (str) – name of the component.

>> **Return type**
>>> None

shutdown()

> A method that can be overwritten to cleanly shutdown (e.g. release resources).

>> **Return type**
>>> None

backend: *Backend*

> Responsible for all I/O communication within this process. Nodes inside the same process share the same message broker. Cannot be modified.

color: str

> Specifies the color of logged messages & node color in the GUI. Check-out the termcolor documentation for the supported colors. Can be set in the subclass' `spec()`.

entity_id: str

> A unique entity_id with the structure <module>/<classname>.

inputs: dict

> Parameters for all selected inputs.

log_level: int

> Specifies the log level for this node: {0: SILENT, 10: DEBUG, 20: INFO, 30: WARN, 40: ERROR, 50: FATAL}. Can be set in the subclass' `spec()`.

log_memory: int

> Specifies the log level for logging memory usage over time for this node: {0: SILENT, 10: DEBUG, 20: INFO, 30: WARN, 40: ERROR, 50: FATAL}. Note that *log_level* has precedent over the memory level set here. Can be set in the subclass' `spec()`.

name: str

> User specified node name. Can be set in `spec()`.

ns: str

> Namespace of the environment. Can be set with the *name* argument to *BaseEnv*.

outputs: dict

> Parameters for all selected outputs.

process: int

> Process in which this node is launched. See *process* for all options. Can be set in the subclass' `spec()`.

rate: float

> Rate (Hz) at which the callback is called. Can be set in the subclass' `spec()`.

real_time_factor: float

> A specified upper bound on the real_time factor. Wall-clock rate=real_time_factor*rate. If real_time_factor < 1 the simulation is slower than real time. Can be set in the engine's `spec()`.

simulate_delays: bool

> Flag that specifies whether input delays are simulated. You probably want to set this to False when running in the real-world. Can be set in the engine's `spec()`.

states: dict

> Parameters for all selected states.

sync: bool

> Flag that specifies whether we run reactive or asynchronous. Can be set in the engine's `spec()`.

Engine Node

class eagerx.core.entities.**EngineNode**(*params*, **args*, *simulator=None*, *message_broker=None*, ***kwargs*)

Baseclass for nodes that are only to be used in combination with a specific engine.

Users must call *make()* to make the engine node subclass' specification.

Use this baseclass to implement nodes that will be added to an *EngineGraph* when specifying an engine implementation for an *Object*.

These nodes can, optionally, be synchronized with respect to the simulator clock by registering "tick" as an input.

Note: Engine nodes *only* receive a reference to the simulator as an argument to *initialize()* when the engine nodes are launched within the same process as the engine. See *process* for more info.

Subclasses must implement the following methods:

- *make()*
- *initialize()*
- *reset()*
- *callback()*
- *shutdown()* (optional)

Use baseclass *Node* instead, for nodes that will be added to the (agnostic) *Graph*.

Use baseclass *ResetNode* instead, for reset routines.

abstract callback(*t_n*, ***inputs*)

An abstract method that is called at the specified node rate.

This method should be decorated with:

- *eagerx.core.register.inputs()* to register the inputs.
- *eagerx.core.register.outputs()* to register the outputs.

> **Parameters**
> - **t_n** (float) – Time passed (seconds) since last reset. Increments with 1/*rate*.
> - **inputs** (*Msg*) – Inputs that were registered (& selected) with the *eagerx.core.register.inputs()* decorator by the subclass.
>
> **Return type**
> Dict[str, Any]
>
> **Returns**
> Dictionary with outputs that were registered (& selected) with the *eagerx.core.register.outputs()* decorator by the subclass.

classmethod info(*method=None*)

A helper method to get info on a method of the specified subclass.

> **Parameters**
> **method** (Union[List[str], str, None]) – The registered method we would like to receive info on. If no method is specified, it provides info on the class itself.

> **Return type**
> > str
>
> **Returns**
> > Info on the subclass' method.

abstract **initialize**(*spec*, *simulator*)

> An abstract method that initializes the node at run-time.
>
> **Parameters**
> - **spec** (*NodeSpec*) – Specification of the engine node.
> - **simulator** (Any) – A reference to the *simulator*. The simulator type depends on the engine. Only available if the node was launched inside the engine process.
>
> **Return type**
> > None

abstract classmethod **make**(**args*, ***kwargs*)

> An abstract method that makes the specification (also referred to as *spec*) of this entity.
>
> **Parameters**
> - **args** (Any) – Arguments to the subclass' make function.
> - **kwargs** (Any) – Optional Arguments to the subclass' make function.
>
> **Returns**
> > A (mutable) spec that can be used to build and subsequently initialize the entity (e.g. node, engine, . . .).

abstract **reset**(***states*)

> An abstract method that resets the node to its initial state before the start of an episode.
>
> This method should be decorated with *eagerx.core.register.states()* to register the states.
>
> > **Warning:** Avoid defining states for engine nodes, as you risk making your *Object* non-agnostic to the environment. Instead, try to implement object states as an *EngineState* of an *Object*.
>
> **Parameters**
> > **states** (Any) – States that were registered (& selected) with the *eagerx.core.register.states()* decorator by the subclass. The state messages are sent by the environment and can be used to reset the node at the start of an episode. This can be anything from an estimator's initial state to a hyper-parameter (e.g. delay, control gains).
>
> **Return type**
> > None

set_delay(*delay*, *component*, *cname*)

> A method to vary the delay of an input or feedthrough.
>
> **Parameters**
> - **delay** (float) – A non-negative delay that can be varied at the beginning of an episode (during the reset procedure).
> - **component** (str) – Either "inputs" or "feedthroughs".
> - **cname** (str) – name of the component.

> **Return type**
>> None

shutdown()

> A method that can be overwritten to cleanly shutdown (e.g. release resources).
>
>> **Return type**
>>> None

backend: *Backend*

> Responsible for all I/O communication within this process. Nodes inside the same process share the same message broker. Cannot be modified.

color: str

> Specifies the color of logged messages & node color in the GUI. Check-out the termcolor documentation for the supported colors. Can be set in the subclass' `spec()`.

entity_id: str

> A unique entity_id with the structure <module>/<classname>.

inputs: dict

> Parameters for all selected inputs.

log_level: int

> Specifies the log level for this node: {0: SILENT, 10: DEBUG, 20: INFO, 30: WARN, 40: ERROR, 50: FATAL}. Can be set in the subclass' `spec()`.

log_memory: int

> Specifies the log level for logging memory usage over time for this node: {0: SILENT, 10: DEBUG, 20: INFO, 30: WARN, 40: ERROR, 50: FATAL}. Note that *log_level* has precedent over the memory level set here. Can be set in the subclass' `spec()`.

name: str

> User specified node name. Can be set in `spec()`.

ns: str

> Namespace of the environment. Can be set with the *name* argument to *BaseEnv*.

outputs: dict

> Parameters for all selected outputs.

process: int

> Process in which this node is launched. See *process* for all options. Can be set in the subclass' `spec()`.

rate: float

> Rate (Hz) at which the callback is called. Can be set in the subclass' `spec()`.

real_time_factor: float

> A specified upper bound on the real_time factor. Wall-clock rate=real_time_factor*rate. If real_time_factor < 1 the simulation is slower than real time. Can be set in the engine's `spec()`.

simulate_delays: bool

> Flag that specifies whether input delays are simulated. You probably want to set this to False when running in the real-world. Can be set in the engine's `spec()`.

states: dict

> Parameters for all selected states.

sync: bool

> Flag that specifies whether we run reactive or asynchronous. Can be set in the engine's `spec()`.

Reset Node

class eagerx.core.entities.**ResetNode**(*params*, **args*, ***kwargs*)

> Baseclass for nodes that perform a reset routine.
>
> Use this baseclass to implement reset nodes that will be added to the (agnostic) *Graph*.
>
> Users must call *make()* to make the reset node subclass' specification.
>
> ---
> **Note:** Subclasses must always have at least one target registered with the *eagerx.core.register.targets()* decorator.
>
> ---
>
> Subclasses must implement the following methods:
>
> - *make()*
> - *initialize()*
> - *reset()*
> - *callback()*
> - *shutdown()* (optional)
>
> Use baseclass *EngineNode* instead, for nodes that will be added to *EngineGraph* when specifying an engine implementation for an *Object*.
>
> Use baseclass *Node* instead, for regular nodes that will be added to the agnostic *Graph*.
>
> abstract **callback**(*t_n*, ***inputs_and_targets*)
>
> > An abstract method that is called at the specified node rate during the environment reset.
> >
> > This method should be decorated with:
> >
> > - *eagerx.core.register.inputs()* to register the inputs.
> > - *eagerx.core.register.outputs()* to register the outputs.
> > - *eagerx.core.register.targets()* to register the targets.
> >
> > ---
> > **Note:** This callback is skipped until the user calls *reset()*. Until then, the messages coming in via the connected *feedthroughs* are fed through as the outputs instead. For every registered output that was registered (& selected) with the *eagerx.core.register.outputs()* decorator by the subclass, there must be a connected *feedthrough*.
> >
> > ---
> >
> > **Parameters**
> >
> > - **t_n** (float) – Time passed (seconds) since last reset. Increments with 1/*rate*.
> > - **inputs_and_targets** (*Msg*) – Inputs and targets that were registered (& selected) with the *eagerx.core.register.inputs()* and *eagerx.core.register.targets()* decorators by the subclass.
> >
> > **Return type**
> > > Dict[str, Any]
> >
> > **Returns**
> > > Dictionary with outputs that were registered (& selected) with the *eagerx.core.register.outputs()* decorator by the subclass. In addition, the dictionary must contain message of type bool that specifies whether the requested *target* was reached.

classmethod **info**(*method=None*)

A helper method to get info on a method of the specified subclass.

> **Parameters**
>> **method** (Union[List[str], str, None]) – The registered method we would like to receive info on. If no method is specified, it provides info on the class itself.
>
> **Return type**
>> str
>
> **Returns**
>> Info on the subclass' method.

abstract **initialize**(*spec*)

An abstract method that initializes the node at run-time.

> **Parameters**
>> **spec** (Union[*NodeSpec*, *EngineSpec*, *ResetNodeSpec*]) – Specification of the node/engine.
>
> **Return type**
>> None

abstract classmethod **make**(**args*, ***kwargs*)

An abstract method that makes the specification (also referred to as *spec*) of this entity.

> **Parameters**
>> - **args** (Any) – Arguments to the subclass' make function.
>> - **kwargs** (Any) – Optional Arguments to the subclass' make function.
>
> **Returns**
>> A (mutable) spec that can be used to build and subsequently initialize the entity (e.g. node, engine, …).

abstract **reset**(***states*)

An abstract method that resets the node to its initial state before the start of an episode.

This method should be decorated with *eagerx.core.register.states()* to register the states.

> **Parameters**
>> **states** (Any) – States that were registered (& selected) with the *eagerx.core.register. states()* decorator by the subclass. The state messages are sent by the environment and can be used to reset the node at the start of an episode. This can be anything from an estimator's initial state to a hyper-parameter (e.g. delay, control gains).
>
> **Return type**
>> None

set_delay(*delay*, *component*, *cname*)

A method to vary the delay of an input or feedthrough.

> **Parameters**
>> - **delay** (float) – A non-negative delay that can be varied at the beginning of an episode (during the reset procedure).
>> - **component** (str) – Either "inputs" or "feedthroughs".
>> - **cname** (str) – name of the component.

> > **Return type**
> > None

shutdown()

> A method that can be overwritten to cleanly shutdown (e.g. release resources).
>
> > **Return type**
> > None

backend: *Backend*

> Responsible for all I/O communication within this process. Nodes inside the same process share the same message broker. Cannot be modified.

color: str

> Specifies the color of logged messages & node color in the GUI. Check-out the termcolor documentation for the supported colors. Can be set in the subclass' `spec()`.

entity_id: str

> A unique entity_id with the structure <module>/<classname>.

feedthroughs: dict

> Parameters for all feedthroughs corresponding to the selected outputs.

inputs: dict

> Parameters for all selected inputs.

log_level: int

> Specifies the log level for this node: {0: SILENT, 10: DEBUG, 20: INFO, 30: WARN, 40: ERROR, 50: FATAL}. Can be set in the subclass' `spec()`.

log_memory: int

> Specifies the log level for logging memory usage over time for this node: {0: SILENT, 10: DEBUG, 20: INFO, 30: WARN, 40: ERROR, 50: FATAL}. Note that *log_level* has precedent over the memory level set here. Can be set in the subclass' `spec()`.

name: str

> User specified node name. Can be set in `spec()`.

ns: str

> Namespace of the environment. Can be set with the *name* argument to *BaseEnv*.

outputs: dict

> Parameters for all selected outputs.

process: int

> Process in which this node is launched. See *process* for all options. Can be set in the subclass' `spec()`.

rate: float

> Rate (Hz) at which the callback is called. Can be set in the subclass' `spec()`.

real_time_factor: float

> A specified upper bound on the real_time factor. Wall-clock rate=real_time_factor*rate. If real_time_factor < 1 the simulation is slower than real time. Can be set in the engine's `spec()`.

simulate_delays: bool

> Flag that specifies whether input delays are simulated. You probably want to set this to False when running in the real-world. Can be set in the engine's `spec()`.

states: dict

> Parameters for all selected states.

sync: bool

> Flag that specifies whether we run reactive or asynchronous. Can be set in the engine's `spec()`.

targets: dict

> Parameters for all selected targets.

2.3.6 Object

class eagerx.core.entities.**Object**

> Baseclass for objects.
>
> Use this baseclass to implement objets that consist of sensors, actuators, and/or engine states.
>
> Users must call *make()* to make the object subclass' specification.
>
> Subclasses must implement the following methods:
>
> * *make()*
>
> For every supported *Engine*, an implementation method must be added. This method must have the same signature as *example_engine()*:
>
> * pybullet() (example)
>
> * ode_engine() (example)
>
> * …
>
> **example_engine**(*spec*, *graph*)
>
> > An example of an engine-specific implementation of an object's registered sensors, actuators, and/or states.
> >
> > See *engine* how engine specific parameters can be set/get.
> >
> > This method must be decorated with *eagerx.core.register.engine()* to register the engine implementation of the object.
> >
> > ---
> >
> > **Note:** This is an example method for documentation purposes only.
> >
> > ---
> >
> > **Parameters**
> >
> > * **spec** (*ObjectSpec*) – A (mutable) specification.
> >
> > * **graph** (*EngineGraph*) – A graph containing the object's registered (disconnected) sensors & actuators. Users should add nodes that inherit from *EngineNode*, and connect them to the sensors & actuators. As such, the engine nodes become the *engine-specific implementation* of the agnostic sensors & actuator definition.
> >
> > **Return type**
> > None
>
> **classmethod** **info**(*method=None*)
>
> > A helper method to get info on a method of the specified subclass.
> >
> > **Parameters**
> > **method** (Union[List[str], str, None]) – The registered method we would like to receive info on. If no method is specified, it provides info on the class itself.

> **Return type**
>> str
>
> **Returns**
>> Info on the subclass' method.

abstract classmethod make(**args*, ***kwargs*)

> An abstract method that makes the specification (also referred to as *spec*) of this object.
>
> See *ObjectSpec* how sensor/actuator/engine state parameters can be set.
>
> This method should be decorated with:
>
> - *eagerx.core.register.sensors()* to register sensors.
> - *eagerx.core.register.actuators()* to register actuators.
> - *eagerx.core.register.engine_states()* to register engine states.
>
> > **Parameters**
> >
> > - **args** (Any) – Arguments to the subclass' make function.
> > - **kwargs** (Any) – Optional Arguments to the subclass' make function.
> >
> > **Returns**
> >> A (mutable) spec that can be used to build and subsequently initialize the entity (e.g. node, object, ...).

2.3.7 Specs

Engine

class eagerx.core.specs.**EngineSpec**(*params*)

> A parameter specification that specifies how *BaseEnv* should initialize the engine.

> **add_object**(*name*, ***kwargs*)
>
> > Adds an object to the simulator that is interfaced by the engine.
> >
> > > **Parameters**
> > >> **kwargs** (Union[bool, int, float, str, List, Dict]) – Other arguments of *add_object()*.
> > >
> > > **Return type**
> > >> None

> **property config: SpecView**
>
> > Provides an API to set/get the parameters to initialize.
> >
> > The default parameters are:
> >
> > - **Spec.config.rate: float**
> >
> >   Rate (Hz) at which the *callback()* is called.
> >
> > - **Spec.config.process: int = 0**
> >
> >   Process in which the engine is launched. See *process* for all options.
> >
> > - **Spec.config.sync: bool = True**
> >
> >   Flag that specifies whether we run reactive or asynchronous.

- `Spec.config.real_time_factor: float = 0`

    A specified upper bound on the real-time factor. *Wall-clock-rate `=` `real_time_factor` `*` rate*. If *real_time_factor* < 1 the simulation is slower than real time.

- `Spec.config.simulate_delays: bool = True`

    Flag that specifies whether input delays are simulated. You probably want to set this to *False* when running in the real-world.

- `Spec.config.color: str = grey`

    Specifies the color of logged messages. Check-out the termcolor documentation for the supported colors.

- `Spec.config.print_mode: int = 1`

    Specifies the different modes for printing: *{1: TERMCOLOR, 2: ROS}*.

- `Spec.config.log_level: int = 30`

    Specifies the log level for the engine: *{0: SILENT, 10: DEBUG, 20: INFO, 30: WARN, 40: ERROR, 50: FATAL}*.

    Returns
        API to get/set parameters.

property inputs: SpecView

Provides an API to set/get the parameters of registered *eagerx.core.register.inputs()*.

The mutable parameters are:

- `Spec.inputs.<name>.window: int = 1`

    A non-negative number that specifies the number of messages to pass to the node's *callback()*.
    – *window* = 1: Only the last received input message.
    – *window* = *x* > 1: The trailing last *x* received input messages.
    – *window* = 0: All input messages received since the last call to the node's *callback()*.

    Note: With *window* = 0, the number of input messages may vary and can even be zero.

- `Spec.inputs.<name>.processor: ProcessorSpec = None`

    A processor that preprocesses the received input message before passing it to the node's *callback()*.

- `Spec.inputs.<name>.space: dict = None`

    This space defines the format of valid messages.

- `Spec.inputs.<name>.delay: float = 0.0`

    A non-negative simulated delay (seconds). This delay is ignored if *simulate_delays* = True in the engine's `spec()`.

- `Spec.inputs.<name>.skip: bool = False`

    Skip the dependency on this input during the first call to the node's *callback()*. May be necessary to ensure that the connected graph is directed and acyclic.

    Returns
        API to get/set parameters.

property objects: SpecView

> Provides an API to set/get the parameters to add an object to the engine.
>
> To add a new object, please use add_object().
>
> Arguments correspond to the signature of *add_object()*.
>
> > **Returns**
> >
> > > API to get/set parameters.

property outputs: SpecView

> Provides an API to set/get the parameters of registered *eagerx.core.register.outputs()*.
>
> The mutable parameters are:
>
> > • **Spec.outputs.<name>.processor: ProcessorSpec = None**
> >
> > > A processor that preprocesses the output message, returned by *callback()*, before publishing it.
> >
> > • **Spec.outputs.<name>.space: dict = None**
> >
> > > This space defines the format of valid messages.
>
> > **Returns**
> >
> > > API to get/set parameters.

property states: SpecView

> Provides an API to set/get the parameters of registered *eagerx.core.register.states()*.
>
> The mutable parameters are:
>
> > • **Spec.states.<name>.space: dict = None**
> >
> > > This space defines the format of valid messages.
>
> > **Returns**
> >
> > > API to get/set parameters.

Backend

class eagerx.core.specs.**BackendSpec**(*params*)

> A parameter specification that specifies how *BaseEnv* should initialize the selected backend.
>
> **property config: SpecView**
>
> > Provides an API to get/set the parameters to initialize.
> >
> > > **Returns**
> > >
> > > > (mutable) API to get/set parameters.

Processor

class eagerx.core.specs.**ProcessorSpec**(*params*)

> A parameter specification that specifies how *BaseEnv* should initialize the processor.
>
> **property config: SpecView**
>
> > Provides an API to get/set the parameters to initialize.
> >
> > > **Returns**
> > >
> > > > (mutable) API to get/set parameters.

Engine State

class eagerx.core.specs.**EngineStateSpec**(*params*)

A parameter specification that specifies how *BaseEnv* should initialize the engine state.

property config: SpecView

Provides an API to get/set the parameters to initialize.

> **Returns**
>
> > API to get/set parameters.

Node

class eagerx.core.specs.**NodeSpec**(*params*)

A parameter specification that specifies how *BaseEnv* should initialize the node.

property config: SpecView

Provides an API to set/get the parameters to initialize.

The default parameters are:

- **Spec.config.name: str**

  User specified unique node name.

- **Spec.config.rate: float**

  Rate (Hz) at which the *callback()* is called.

- **Spec.config.process: int = 0**

  Process in which the node is launched. See *process* for all options.

- **Spec.config.color: str = grey**

  Specifies the color of logged messages & node color in the GUI. Check-out the termcolor documentation for the supported colors.

- **Spec.config.print_mode: int = 1**

  Specifies the different modes for printing: *{1: TERMCOLOR, 2: ROS}*.

- **Spec.config.log_level: int = 30**

  Specifies the log level for the engine: *{0: SILENT, 10: DEBUG, 20: INFO, 30: WARN, 40: ERROR, 50: FATAL}*

> **Returns**
>
> > API to get/set parameters.

property inputs: SpecView

Provides an API to set/get the parameters of registered *eagerx.core.register.inputs()*.

The mutable parameters are:

- **Spec.inputs.<name>.window: int = 1**

  A non-negative number that specifies the number of messages to pass to the node's *callback()*.
  - *window* = 1: Only the last received input message.
  - *window* = *x* > 1: The trailing last *x* received input messages.
  - *window* = 0: All input messages received since the last call to the node's *callback()*.

  Note: With *window* = 0, the number of input messages may vary and can even be zero.

- **Spec.inputs.<name>.processor: ProcessorSpec = None**

    A processor that preprocesses the received input message before passing it to the node's *callback()*.

- **Spec.inputs.<name>.space: dict = None**

    This space defines the format of valid messages.

- **Spec.inputs.<name>.delay: float = 0.0**

    A non-negative simulated delay (seconds). This delay is ignored if *simulate_delays* = True in the engine's `spec()`.

- **Spec.inputs.<name>.skip: bool = False**

    Skip the dependency on this input during the first call to the node's *callback()*. May be necessary to ensure that the connected graph is directed and acyclic.

    Returns

    API to get/set parameters.

property outputs: SpecView

Provides an API to set/get the parameters of registered *eagerx.core.register.outputs()*.

The mutable parameters are:

- **Spec.outputs.<name>.processor: ProcessorSpec = None**

    A processor that preprocesses the output message, returned by *callback()*, before publishing it.

- **Spec.outputs.<name>.space: dict = None**

    This space defines the format of valid messages.

    Returns

    API to get/set parameters.

property states: SpecView

Provides an API to set/get the parameters of registered *eagerx.core.register.states()*.

The mutable parameters are:

- **Spec.states.<name>.space: dict = None**

    This space defines the format of valid messages.

    Returns

    API to get/set parameters.

Reset Node

class eagerx.core.specs.**ResetNodeSpec**(*params*)

A parameter specification that specifies how *BaseEnv* should initialize the node.

property config: SpecView

Provides an API to set/get the parameters to initialize.

The default parameters are:

- **Spec.config.name: str**

    User specified unique node name.

- **Spec.config.rate: float**

    Rate (Hz) at which the *callback()* is called.

- **Spec.config.process: int = 0**

    Process in which the node is launched. See *process* for all options.

- **Spec.config.color: str = grey**

    Specifies the color of logged messages & node color in the GUI. Check-out the termcolor documentation for the supported colors.

- **Spec.config.print_mode: int = 1**

    Specifies the different modes for printing: *{1: TERMCOLOR, 2: ROS}*.

- **Spec.config.log_level: int = 30**

    Specifies the log level for the engine: *{0: SILENT, 10: DEBUG, 20: INFO, 30: WARN, 40: ERROR, 50: FATAL}*

    Returns

    API to get/set parameters.

property feedthroughs: SpecView

Provides an API to set/get the parameters of a feedthrough corresponding to registered *eagerx.core. register.outputs()*.

The mutable parameters are:

- **Spec.feedthroughs.<name>.processor: ProcessorSpec = None**

    A processor that preprocesses the received input message before passing it to the node's *callback()*.

- **Spec.feedthroughs.<name>.space: dict = None**

    This space defines the format of valid messages.

- **Spec.feedthroughs.<name>.delay: float = 0.0**

    A non-negative simulated delay (seconds). This delay is ignored if *simulate_delays* = True in the engine's spec().

    Returns

    API to get/set parameters.

property inputs: SpecView

Provides an API to set/get the parameters of registered *eagerx.core.register.inputs()*.

The mutable parameters are:

- **Spec.inputs.<name>.window: int = 1**

    A non-negative number that specifies the number of messages to pass to the node's *callback()*.
    – *window* = 1: Only the last received input message.
    – *window* = x > 1: The trailing last *x* received input messages.
    – *window* = 0: All input messages received since the last call to the node's *callback()*.

    Note: With *window* = 0, the number of input messages may vary and can even be zero.

- **Spec.inputs.<name>.processor: ProcessorSpec = None**

    A processor that preprocesses the received input message before passing it to the node's *callback()*.

- `Spec.inputs.<name>.space: dict = None`

    This space defines the format of valid messages.

- `Spec.inputs.<name>.delay: float = 0.0`

    A non-negative simulated delay (seconds). This delay is ignored if *simulate_delays* = True in the engine's `spec()`.

- `Spec.inputs.<name>.skip: bool = False`

    Skip the dependency on this input during the first call to the node's *callback()*. May be necessary to ensure that the connected graph is directed and acyclic.

    Returns
    API to get/set parameters.

`property outputs:  SpecView`

Provides an API to set/get the parameters of registered *eagerx.core.register.outputs()*.

The mutable parameters are:

- `Spec.outputs.<name>.processor: ProcessorSpec = None`

    A processor that preprocesses the output message, returned by *callback()*, before publishing it.

- `Spec.outputs.<name>.space: dict = None`

    This space defines the format of valid messages.

    Returns
    API to get/set parameters.

`property states:  SpecView`

Provides an API to set/get the parameters of registered *eagerx.core.register.states()*.

The mutable parameters are:

- `Spec.states.<name>.space: dict = None`

    This space defines the format of valid messages.

    Returns
    API to get/set parameters.

`property targets:  SpecView`

Provides an API to set/get the parameters of registered *eagerx.core.register.targets()*.

The mutable parameters are:

- `Spec.targets.<name>.processor: ProcessorSpec = None`

    A processor that preprocesses the received state message before passing it to the node's *callback()*.

    Returns
    API to get/set parameters.

Object

class eagerx.core.specs.**ObjectSpec**(*params*)

> A parameter specification of an object.

> **gui**(*engine_cls*, *interactive=True*, *resolution=None*, *filename=None*)

>> Opens a graphical user interface of the object's engine implementation.

>> ---

>> **Note:** Requires *eagerx-gui*:

>> ---

>> ```
>> pip3 install eagerx-gui
>> ```

>> **Parameters**

>>> - **engine_cls** (Type[*Engine*]) – The class engine (not instance!) that was used to register the engine implementation (e.g. "PybulletEngine").

>>> - **interactive** (Optional[bool]) – If *True*, an interactive application is launched. Otherwise, an RGB render of the GUI is returned. This could be useful when using a headless machine.

>>> - **resolution** (Optional[List[int]]) – Specifies the resolution of the returned render when *interactive* is *False*. If *interactive* is *True*, this argument is ignored.

>>> - **filename** (Optional[str]) – If provided, the GUI is rendered to an svg file with this name. If *interactive* is *True*, this argument is ignored.

>> **Return type**
>>> Optional[ndarray]

>> **Returns**
>>> RGB render of the GUI if *interactive* is *False*.

> **property actuators: SpecView**

>> Provides an API to set/get the parameters of registered *eagerx.core.register.actuators()*.

>> The mutable parameters are:

>> - Spec.actuators.<name>.rate: float = 1.0

>>> Rate (Hz) at which the actuator's *callback()* is called.

>> - Spec.actuators.<name>.window: int = 1

>>> A non-negative number that specifies the number of messages to pass to the node's *callback()*.
>>> – *window* = 1: Only the last received input message.
>>> – *window* = *x* > 1: The trailing last *x* received input messages.
>>> – *window* = 0: All input messages received since the last call to the node's *callback()*.

>>> ---

>>> **Note:** With *window* = 0, the number of input messages may vary and can even be zero.

>>> ---

>> - Spec.actuators.<name>.space: dict = None

>>> This space defines the format of valid messages.

>> - Spec.actuators.<name>.delay: float = 0.0

>>> A non-negative simulated delay (seconds). This delay is ignored if *simulate_delays* = True in the engine's spec().

- `Spec.actuators.<name>.skip: bool = False`

    Skip the dependency on this input during the first call to the node's `callback()`. May be necessary to ensure that the connected graph is directed and acyclic.

    Returns

    API to get/set parameters.

property config: SpecView

Provides an API to set/get the parameters to initialize.

The default parameters are:

- Additional parameters registered with the `eagerx.core.register.config()` decorator.

- `Spec.config.name: str`

    User specified unique object name.

- `Spec.config.actuators: list`

    List with selected actuators. Must be a subset of the registered *eagerx.core.register.actuators()*.

- `Spec.config.sensors: list`

    List with selected sensors. Must be a subset of the registered *eagerx.core.register.sensors()*.

- `Spec.config.states: list`

    List with selected engine_states. Must be a subset of the registered *eagerx.core.register.engine_states()*.

    Returns

    API to get/set parameters.

property engine: SpecView

Provides an API to set/get the parameters of an engine-specific implementation.

The mutable parameters are:

- Arguments (excluding spec) of the selected engine's *add_object()* method.

- `Spec.engine.states.<name>: EngineState`

    Link an `EngineState` to a registered state with *eagerx.core.register.states()*.

    Returns

    API to get/set parameters.

property sensors: SpecView

Provides an API to set/get the parameters of registered *eagerx.core.register.sensors()*.

The mutable parameters are:

- `Spec.sensors.<name>.rate: float = 1.0`

    Rate (Hz) at which the sensor's *callback()* is called.

- `Spec.sensors.<name>.space: dict = None`

    This space defines the format of valid messages.

    Returns

    API to get/set parameters.

property states: SpecView

Provides an API to set/get the parameters of registered *eagerx.core.register.engine_states()*.

The mutable parameters are:

- **Spec.states.<name>.space: dict = None**

  This space defines the format of valid messages.

  Returns

  API to get/set parameters.

2.3.8 Graph

Graph

class eagerx.core.graph.**Graph**(*state*)

The Graph API allows users to form a graph of connected nodes and objects.

add(*entities*)

Add nodes/objects to the graph.

> **Parameters**
>
> **entities** (Union[*NodeSpec*, *ResetNodeSpec*, *ObjectSpec*, *EngineSpec*, List[Union[*NodeSpec*, *ResetNodeSpec*, *ObjectSpec*, *EngineSpec*]]]) – Nodes/objects to add.
>
> **Return type**
>
> None

add_component(*entry=None*, *action=None*, *observation=None*)

Selects an available component entry (e.g. input, output, etc. . .) that was not already selected.

> **Parameters**
>
> - **entry** (Optional[SpecView]) – Selects the entry, so that it can be connected.
>
> - **action** (Optional[str]) – Adds a disconnected action entry.
>
> - **observation** (Optional[str]) – Adds a disconnected observation entry.
>
> **Return type**
>
> None

connect(*source=None*, *target=None*, *action=None*, *observation=None*, *window=None*, *delay=None*, *skip=None*)

Connect an action/source (i.e. node/object component) to an observation/target (i.e. node/object component).

> **Parameters**
>
> - **source** (Optional[SpecView]) – Compatible source types are *outputs*, *sensors*, and *states*.
>
> - **target** (Optional[SpecView]) – Compatible target types are *inputs*, *actuators*, *targets*, and *feedthroughs*.
>
> - **action** (Optional[str]) – Name of the action to connect (and add).
>
> - **observation** (Optional[str]) – Name of the observation to connect (and add).

- **window** (Optional[int]) – A non-negative number that specifies the number of messages to pass to the node's *callback()*.

    - *window* = 1: Only the last received input message.

    - *window* = *x* > 1: The trailing last *x* received input messages.

    - *window* = 0: All input messages received since the last call to the node's *callback()*.

    Note: With *window* = 0, the number of input messages may vary and can even be zero.

- **delay** (Optional[float]) – A non-negative simulated delay (seconds). This delay is ignored if *simulate_delays* = True in the engine's `spec()`.

- **skip** (Optional[bool]) – Skip the dependency on this input during the first call to the node's *callback()*. May be necessary to ensure that the connected graph is directed and acyclic.

    Return type
    None

classmethod **create**(*nodes=None*, *objects=None*)

> Create a new graph with nodes and objects.
>
> **Parameters**
>
> - **nodes** (Union[*NodeSpec*, *ResetNodeSpec*, List[Union[*NodeSpec*, *ResetNodeSpec*]], None]) – Nodes to add.
>
> - **objects** (Union[*ObjectSpec*, List[*ObjectSpec*], None]) – Objects to add.
>
> **Return type**
> *Graph*
>
> **Returns**
> The graph.

disconnect(*source=None*, *target=None*, *action=None*, *observation=None*, *remove=False*)

> Disconnects a source/action from a target/observation.
>
> **Parameters**
>
> - **source** (Optional[SpecView]) – Compatible source types are *outputs*, *sensors*, and *states*.
>
> - **target** (Optional[SpecView]) – Compatible target types are *inputs*, *actuators*, *targets*, and *feedthroughs*.
>
> - **action** (Optional[str]) – Name of the action to connect (and add).
>
> - **observation** (Optional[str]) – Name of the observation to connect (and add).
>
> - **remove** (bool) – Flag to also remove observations/actions if they are left disconnected after the entry was disconnected. Actions are only removed if they are completely disconnected.
>
> **Return type**
> None

get(*entry=None*, *action=None*, *observation=None*, *parameter=None*)

> Fetches the parameters of a node/object/action/observation.
>
> **Parameters**

- **entry** (Union[SpecView, EntitySpec, None]) – The entry whose parameters are fetched.

- **action** (Optional[str]) – Action name whose parameters are fetched.

- **observation** (Optional[str]) – observation name whose parameters are fetched.

- **parameter** (Optional[str]) – If only a single parameter needs to be fetched.

> **Return type**
> > Any

> **Returns**
> > Parameters

get_spec(*name*)

> Get Spec from the graph

> > **Parameters**
> > > **name** (str) – Name

> > **Return type**
> > > EntitySpec

> > **Returns**
> > > The specification of the entity.

gui(*interactive=True*, *resolution=None*, *filename=None*)

> Opens a graphical user interface of the graph.

> **Note:** Requires *eagerx-gui*:
>
> ```
> pip3 install eagerx-gui
> ```

> > **Parameters**
> >
> > - **interactive** (Optional[bool]) – If *True*, an interactive application is launched. Otherwise, an RGB render of the GUI is returned. This could be useful when using a headless machine.
> >
> > - **resolution** (Optional[List[int]]) – Specifies the resolution of the returned render when *interactive* is *False*. If *interactive* is *True*, this argument is ignored.
> >
> > - **filename** (Optional[str]) – If provided, the GUI is rendered to an svg file with this name. If *interactive* is *True*, this argument is ignored.
> >
> > **Return type**
> > > Optional[ndarray]
> >
> > **Returns**
> > > RGB render of the GUI if *interactive* is *False*.

is_valid(*plot=True*)

> Checks the validity of the graph.

- Checks if all selected actions, observations, *inputs*, *actuators*, *targets*, and *feedthroughs* are connected.

- Checks if the graph is directed and acyclic.

Parameters

    plot – Flag to plot the graph. Can be helpful to identify cycles in the graph that break the required acyclic property.

Return type

    bool

Returns

    flag that specifies the validity of the graph.

classmethod load(*file*)

    Loads the graph state.

    The state is loaded in *.yaml* format and contains the state of every added node, object, action, and observation and the connections between them.

    Parameters

        file (str) – A string giving the name (and the file if the file isn't in the current working directory).

reload()

    Reloads (ie imports) all entities in the graph.

remove(*names*, *remove=False*)

    Removes nodes/objects from the graph.

- First, all associated connections are disconnected.

- Then, removes the node/object.

    Parameters

- **names** (Union[str, EntitySpec, List[Union[str, EntitySpec]]]) – Either the name or spec of the node/object that is to be removed.

- **remove** (bool) – Flag to also remove observations/actions if they are left disconnected after the node/object was removed. Actions are only removed if they are completely disconnected.

    Return type

        List

    Returns

        list of disconnected connections.

remove_component(*entry=None*, *action=None*, *observation=None*, *remove=False*)

    Deselects a component entry (e.g. input, output, etc. . .) that was selected.

- First, all associated connections are disconnected.

- Then, deselects the component entry. For feedthroughs, it will also remove the corresponding output entry.

    Parameters

- **entry** (Optional[SpecView]) – Deselects the entry.

- **action** (Optional[str]) – Removes an action entry.

- **observation** (Optional[str]) – Removes an observation entry

- **remove** (bool) – Flag to also remove observations/actions if they are left disconnected after the entry was removed. Actions are only removed if they are completely disconnected.

> **Return type**
> None

rename(*new*, *action=None*, *observation=None*)

> Renames an action/observation.

> **Parameters**
>
> - **new** (str) – New name.
>
> - **action** (Optional[str]) – Old action name.
>
> - **observation** (Optional[str]) – Old observation name.

> **Return type**
> None

render(*source*, *rate*, *processor=None*, *window=None*, *delay=None*, *skip=None*, *render_cls=None*, *process=0*, *encoding='bgr'*, ***kwargs*)

> Visualize rgb images produced by a node/sensor in the graph. The rgb images must be of *dtype=uint8* and *shape=(height, width, 3)*.

> **Parameters**
>
> - **source** (SpecView) – Compatible source types are *outputs* and *sensors*.
>
> - **rate** (float) – The rate (Hz) at which to render the images.
>
> - **processor** (Optional[*ProcessorSpec*]) – Processes the received message before passing it to the target node's *callback()*.
>
> - **window** (Optional[int]) – A non-negative number that specifies the number of messages to pass to the node's *callback()*.
>
>   - *window* = 1: Only the last received input message.
>
>   - *window* = *x* > 1: The trailing last *x* received input messages.
>
>   - *window* = 0: All input messages received since the last call to the node's *callback()*.
>
>   ---
>
>   **Note:** With *window* = 0, the number of input messages may vary and can even be zero.
>
>   ---
>
> - **delay** (Optional[float]) – A non-negative simulated delay (seconds). This delay is ignored if *simulate_delays* = True in the engine's spec().
>
> - **skip** (Optional[bool]) – Skip the dependency on this input during the first call to the node's *callback()*. May be necessary to ensure that the connected graph is directed and acyclic.
>
> - **render_cls** (Optional[Type[*Node*]]) – The *Node* of the render node. By default, it uses the standard *RenderNode*. In Google colab, the *ColabRender* class is used.
>
> - **process** (int) – Process in which the render node is launched. See *process* for all options.
>
> - **encoding** (str) – The encoding (*bgr* or *rgb*) of the render source.
>
> - **kwargs** – Optional arguments required by the render node.

save(*file*)

    Saves the graph state.

    The state is saved in *.yaml* format and contains the state of every added node, object, action, and observation and the connections between them.

    Parameters

        file (`str`) – A string giving the name (and the file if the file isn't in the current working directory).

    Return type

        None

set(*mapping*, *entry=None*, *action=None*, *observation=None*, *parameter=None*)

    Sets the parameters of a node/object/action/observation.

    Parameters

- **mapping** (`Any`) – Either a mapping with *key = parameter*, or a single value that corresponds to the optional *parameter* argument.
- **entry** (`Optional[SpecView]`) – The entry whose parameters are mutated.
- **action** (`Optional[str]`) – Action name whose parameters are mutated.
- **observation** (`Optional[str]`) – observation name whose parameters are mutated.
- **parameter** (`Optional[str]`) – If only a single value needs to be set. See documentation for *mapping*.

    Return type

        None

Engine Graph

class eagerx.core.graph_engine.**EngineGraph**(*state*)

    add(*nodes*)

        Add nodes to the graph.

        Parameters

            nodes (`Union[`*NodeSpec*`, List[`*NodeSpec*`]]`) – Nodes/objects to add.

        Return type

            None

    add_component(*entry*)

        Selects an available component entry (e.g. input, output, etc...) that was not already selected.

        Parameters

            entry (`SpecView`) – Selects the entry, so that it can be connected.

        Return type

            None

    connect(*source=None*, *target=None*, *actuator=None*, *sensor=None*, *window=None*, *delay=None*, *skip=None*)

        Connect an actuator/source to a sensor/target.

        Parameters

- **source** (`Optional[SpecView]`) – Compatible source type is *outputs*.

- **target** (Optional[SpecView]) – Compatible target type is *inputs*.

- **actuator** (Optional[str]) – String name of the actuator.

- **sensor** (Optional[str]) – String name of the sensor.

- **window** (Optional[int]) – A non-negative number that specifies the number of messages to pass to the node's *callback()*.

  - *window* = 1: Only the last received input message.

  - *window* = *x* > 1: The trailing last *x* received input messages.

  - *window* = 0: All input messages received since the last call to the node's *callback()*.

  Note: With *window* = 0, the number of input messages may vary and can even be zero.

- **delay** (Optional[float]) – A non-negative simulated delay (seconds). This delay is ignored if *simulate_delays* = True in the engine's spec().

- **skip** (Optional[bool]) – Skip the dependency on this input during the first call to the node's *callback()*. May be necessary to ensure that the connected graph is directed and acyclic.

Return type
    None

disconnect(*source=None*, *target=None*, *actuator=None*, *sensor=None*)

    Disconnect an actuator/source from a sensor/target.

Parameters

- **source** (Optional[SpecView]) – Compatible source type is *outputs*.

- **target** (Optional[SpecView]) – Compatible target type is *inputs*.

- **actuator** (Optional[str]) – String name of the actuator.

- **sensor** (Optional[str]) – String name of the sensor.

Return type
    None

get(*entry=None*, *actuator=None*, *sensor=None*, *parameter=None*)

    Fetches the parameters of a node/actuator/sensor.

Parameters

- **entry** (Union[SpecView, EntitySpec, None]) – The entry whose parameters are fetched.

- **actuator** (Optional[str]) – Actuator name whose parameters are fetched.

- **sensor** (Optional[str]) – Sensor name whose parameters are fetched.

- **parameter** (Optional[str]) – If only a single parameter needs to be fetched.

Return type
    Any

Returns
    Parameters

get_spec(*name*)

> Get Spec from the graph
>
> > **Parameters**
> >   **name** (str) – Name
> >
> > **Return type**
> >   [*NodeSpec*](#)
> >
> > **Returns**
> >   The specification of the entity.

gui(*interactive=True*, *resolution=None*, *filename=None*)

> Opens a graphical user interface of the graph.
>
> ---
>
> **Note:** Requires *eagerx-gui*:
>
> ```
> pip3 install eagerx-gui
> ```
>
> ---
>
> > **Parameters**
> >
> > - **interactive** (Optional[bool]) – If *True*, an interactive application is launched. Otherwise, an RGB render of the GUI is returned. This could be useful when using a headless machine.
> >
> > - **resolution** (Optional[List[int]]) – Specifies the resolution of the returned render when *interactive* is *False*. If *interactive* is *True*, this argument is ignored.
> >
> > - **filename** (Optional[str]) – If provided, the GUI is rendered to an svg file with this name. If *interactive* is *True*, this argument is ignored.
> >
> > **Return type**
> >   Optional[ndarray]
> >
> > **Returns**
> >   RGB render of the GUI if *interactive* is *False*.

is_valid(*plot=True*)

> Checks the validity of the graph.
>
> - Checks if all selected [`inputs`](#) are connected.
> - Checks if the graph is directed and acyclic.
>
> > **Parameters**
> >   **plot** – Flag to plot the graph. Can be helpful to identify cycles in the graph that break the required acyclic property.
> >
> > **Return type**
> >   bool
> >
> > **Returns**
> >   flag that specifies the validity of the graph.

register()

> Returns the nodes that make up this subgraph, and their relation to the registered actuators and sensors.

remove(*names*)

Removes a node from the graph.

- First, all associated connections are disconnected.

- Then, removes the nodes/objects.

> **Parameters**
>> **names** (Union[str, EntitySpec, List[Union[str, EntitySpec]]]) – Either the name or spec of the node/object that is to be removed.
>
> **Return type**
>> None

remove_component(*entry*)

Deselects a component entry (e.g. input, output, etc. . .) that was selected.

- First, all associated connections are disconnected.

- Then, deselects the component entry.

> **Parameters**
>> **entry** (SpecView) – Deselects the entry.
>
> **Return type**
>> None

set(*mapping*, *entry*, *parameter=None*)

Sets the parameters of a node.

> **Parameters**
>
>> - **mapping** (Any) – Either a mapping with *key = parameter*, or a single value that corresponds to the optional *parameter* argument.
>>
>> - **entry** (Optional[SpecView]) – The entry whose parameters are mutated.
>>
>> - **parameter** (Optional[str]) – If only a single value needs to be set. See documentation for *mapping*.
>
> **Return type**
>> None

2.3.9 Environment

class eagerx.core.env.**BaseEnv**(*name*, *rate*, *graph*, *engine*, *backend=None*, *force_start=True*, *render_mode=None*)

The base class for all EAGERx environments that follows the OpenAI gym's Env API.

- Be sure to call super().__init__() inside the subclass' constructor with the required arguments (name, graph, etc. . .).

A subclass should implement the following methods:

- *step()*: Be sure to call *_step()* inside this method to perform the step.

- *reset()*: Be sure to call *_reset()* inside this method to perform the reset.

A subclass can optionally overwrite the following properties:

- *observation_space*: Per default, the observations, registered in the graph, are taken.

- *action_space*: Per default, the actions, registered in the graph, are taken.

__init__(*name*, *rate*, *graph*, *engine*, *backend=None*, *force_start=True*, *render_mode=None*)

Initializes an environment with EAGERx dynamics.

> **Parameters**
>
> - **name** (str) – The name of the environment. Everything related to this environment (parameters, topics, nodes, etc...) will be registered under namespace: "*/name*".
>
> - **rate** (float) – The rate (Hz) at which the environment will run.
>
> - **graph** (*Graph*) – The graph consisting of nodes and objects that describe the environment's dynamics.
>
> - **engine** (*EngineSpec*) – The physics engine that will govern the environment's dynamics. For every *Object* in the graph, the corresponding engine implementations is chosen.
>
> - **backend** (Optional[*BackendSpec*]) – The backend that will govern the communication for this environment. Per default, the SingleProcess backend is used.
>
> - **force_start** (bool) – If there already exists an environment with the same name, the existing environment is first shutdown by calling the *BaseEnv()* method before initializing this environment.
>
> - **render_mode** (Optional[str]) – The render mode that will be used for rendering the environment.

_reset(*states*)

A private method that should be called within *reset()*.

> **Parameters**
> **states** (Dict) – The desired states to be set before the start an episode. May also be an (empty) subset of registered states if not all states require a reset.
>
> **Return type**
> Dict
>
> **Returns**
> The initial observation.

_step(*action*)

A private method that should be called within *step()*.

> **Parameters**
> **action** (Dict) – The actions to be applied in the next timestep. Should include all registered actions.
>
> **Return type**
> Dict
>
> **Returns**
> The observation of the current timestep that comply with the graph's observation space.

close()

A method to stop rendering (i.e. close the render window).

A bool message to topic address "**name** */env/render/toggle*", which toggles the rendering on/off.

Note: Depending on the source node that is producing the images that are rendered, images may still be produced, even when the render window is not visible. This may add computational overhead and influence the run speed.

Optionally, users may subscribe to topic address "name */env/render/toggle*" in the node that is producing the images to stop the production and output empty images instead.

---

gui(*interactive=True*, *resolution=None*, *filename=None*)

Opens a graphical user interface of the graph.

Note: Requires *eagerx-gui*:

```
pip3 install eagerx-gui
```

Parameters

- **interactive** (Optional[bool]) – If *True*, an interactive application is launched. Otherwise, an RGB render of the GUI is returned. This could be useful when using a headless machine.

- **resolution** (Optional[List[int]]) – Specifies the resolution of the returned render when *interactive* is *False*. If *interactive* is *True*, this argument is ignored.

- **filename** (Optional[str]) – If provided, the GUI is rendered to an svg file with this name. If *interactive* is *True*, this argument is ignored.

Return type

Optional[ndarray]

Returns

RGB render of the GUI if *interactive* is *False*.

classmethod load(*name*, *file*, *backend=None*, *force_start=True*)

Loads an environment corresponding to the graph state.

Parameters

- **name** (str) – The name of the environment. Everything related to this environment (parameters, topics, nodes, etc…) will be registered under namespace: "*/name*".

- **file** (str) – A string giving the name (and the file if the file isn't in the current working directory).

- **backend** (Optional[*BackendSpec*]) – The backend that will govern the communication for this environment. Per default, the `SingleProcess` backend is used.

- **force_start** (bool) – If there already exists an environment with the same name, the existing environment is first shutdown by calling the *BaseEnv()* method before initializing this environment.

render()

A method to start rendering (i.e. open the render window).

A bool message to topic address "name */env/render/toggle*", which toggles the rendering on/off. :rtype: Optional[ndarray] :returns: Optionally, a rgb_array if env.mode=rgb_array.

abstract **reset**(*seed=None*, *options=None*)

An abstract method that resets the environment to an initial state and returns an initial observation.

Note:

To reset the graph, the private method _reset() must be called with the
desired initial states. The spaces of all states (of Objects and Nodes in the graph) are stored in

state_space().

> **Return type**
> Tuple[Union[Dict, ndarray], Dict]
>
> **Returns**
> The initial observation that is complies with the *observation_space()*.

save(*file*)

Saves the (engine-specific) graph state, that includes the engine & environment nodes.

The state is saved in *.yaml* format and contains the state of every added node, action, and observation and the connections between them.

> **Parameters**
> **file** (str) – A string giving the name (and the file if the file isn't in the current working directory).
>
> **Return type**
> None

shutdown()

A method to shutdown the environment.

- Clear the parameters on the ROS parameter under the namespace /name.

- Close nodes (i.e. release resources and perform close procedure).

- Unregister topics that supplied the I/O communication between nodes.

abstract **step**(*action*)

An abstract method that runs one timestep of the environment's dynamics.

Note: To run one timestep of the graph dynamics (that essentially define the environment dynamics), this method must call the private method _step() with the actions that comply with _action_space.

When the end of an episode is reached, the user is responsible for calling reset() to reset this environment's state.

> **Params action**
> Actions provided by the agent. Should comply with the *action_space()*.
>
> **Return type**
> Tuple[Union[Dict, ndarray], float, bool, bool, Dict]
>
> **Returns**
> A tuple (observation, reward, terminated, truncated, info).
>
> - **observation: Observations of the current timestep that comply with**
>   the *observation_space()*.

- reward: amount of reward returned after previous action

- **terminated: whether the episode has ended due to a terminal state, in which case further step() calls will**
    return undefined results

- **truncated: whether the episode has ended due to a time limit, in which case further step() calls will**
    return undefined results

- info: contains auxiliary diagnostic information (helpful for debugging, and sometimes learning)

property _action_space: Dict

Infers the action space from the space of every action.

This space defines the format of valid actions.

> **Returns**
> > A dictionary with *key = action* and *value =* Space.

property _observation_space: Dict

Infers the observation space from the space of every observation.

This space defines the format of valid observations.

Note: Observations with window = 0 are excluded from the observation space. For observations with window > 1, the observation space is duplicated window times.

> **Returns**
> > A dictionary with *key = observation* and *value =* Space.

property action_space: Space

The Space object corresponding to valid actions

Per default, the action space of all registered actions in the graph is used.

property np_random: Generator

Returns the environment's internal _np_random that if not set will initialise with a random seed.

> **Returns:**
> > Instances of *np.random.Generator*

property observation_space: Space

The Space object corresponding to valid observations.

Per default, the observation space of all registered observations in the graph is used.

property state_space: Dict

Infers the state space from the space of every state.

This space defines the format of valid states that can be set before the start of an episode.

> **Returns**
> > A dictionary with *key = state* and *value =* Space.

2.3.10 Utilities

Space

class eagerx.core.space.**Space**(*low=None*, *high=None*, *shape=None*, *dtype=<class 'numpy.float32'>*, *seed=None*)

A (possibly unbounded) space in R^n. Specifically, a Space represents the Cartesian product of n closed intervals. Each interval has the form of one of [a, b], (-oo, b], [a, oo), or (-oo, oo).

There are two common use cases:

- **Identical bound for each dimension::**

    ```
    >>> Space(low=-1.0, high=2.0, shape=(3, 4), dtype="float32")
    Space(3, 4)
    ```

- **Independent bound for each dimension::**

    ```
    >>> Space(low=np.array([-1.0, -2.0]), high=np.array([2.0, 4.0]), dtype=
    →"float32")
    Space(2,)
    ```

contains(*x*)

Return boolean specifying if x is a valid member of this space :type x: :param x: array to check.

> **Return type**
> > bool

contains_space(*space*)

Return boolean specifying if space is contained in this space. Low and high of the space must exactly match (instead of lying within the bounds) to return True :type space: Union[*Space*, Dict] :param space: Space that is to be checked.

> **Return type**
> > bool

classmethod from_dict(*d*)

Create a space from a dict.

> **Parameters**
> > **d** (Dict) – Dict containing the arguments to initialize the space
>
> **Return type**
> > *Space*
>
> **Returns**
> > The space.

from_jsonable(*sample_n*)

Convert a JSONable data type to a batch of samples from this space.

sample()

Randomly sample an element of this space. Can be uniform or non-uniform sampling based on boundedness of space.

> **Return type**
> > ndarray

seed(*seed=None*)

>   Seed the PRNG of this space and possibly the PRNGs of subspaces.

>>   **Return type**
>>   list[int]

to_dict()

>   Convert the space to a dict representation

>>   **Return type**
>>   Dict

>>   **Returns**
>>   Dict representation of the space.

to_jsonable(*sample_n*)

>   Convert a batch of samples from this space to a JSONable data type.

property is_fully_defined: bool

>   Check if space is fully defined (i.e. low, high, shape and dtype are all provided). :return: flag

property is_np_flattenable

>   Checks whether this space can be flattened.

property np_random: Generator

>   Lazily seed the PRNG since this is expensive and only needed if sampling from this space.

>   As *seed()* is not guaranteed to set the *_np_random* for particular seeds. We add a check after *seed()* to set a new random number generator.

property shape: tuple[int, ...] | None

>   Return the shape of the space as an immutable property.

Process

class eagerx.core.constants.**process**

>   **ENGINE: int = 2**

>>   Spawn a node in the process of the engine. If an *EngineNode* requires direct access to the *simulator*, config, and engine_config, it must be spawned in the same process as the engine.

>   **ENVIRONMENT: int = 1**

>>   Spawn the node/engine in the process of the environment.

>   **EXTERNAL: int = 3**

>>   Spawn the node/engine in a separate process. This process is not spawned by the environment. Instead, the user is responsible for running the executable script with the appropriate arguments. This allows nodes to run distributed.

>   **NEW_PROCESS: int = 0**

>>   Spawn the node/engine in a separate process. Allows parallelization, but increases communication overhead due to the (de)serialization of messages.

Register

class eagerx.core.register.**inputs**(***inputs*)

A decorator to register the inputs to a *callback()*.

The *callback()* method should be decorated.

> **Parameters**
> **inputs** (Any) – The input's msg_type class.

> **Return type**
> Callable

class eagerx.core.register.**outputs**(***outputs*)

A decorator to register the outputs of a *callback()*.

The *callback()* method should be decorated.

> **Parameters**
> **outputs** – The output's msg_type class.

> **Return type**
> Callable

class eagerx.core.register.**states**(***states*)

A decorator to register the states for a *reset()*.

The *reset()* method should be decorated.

> **Parameters**
> **outputs** – The state's msg_type class.

> **Return type**
> Callable

class eagerx.core.register.**targets**(***targets*)

A decorator to register the targets of a *callback()*.

The *callback()* method should be decorated.

> **Parameters**
> **targets** – The target's msg_type class.

> **Return type**
> Callable

class eagerx.core.register.**sensors**(***sensors*)

A decorator to register the sensors of an *Object*.

The agnostic() method should be decorated.

> **Parameters**
> **sensors** – The sensor's msg_type class.

> **Return type**
> Callable

class eagerx.core.register.**actuators**(***actuators*)

A decorator to register the actuators of an *Object*.

The agnostic() method should be decorated.

> **Parameters**
> **actuators** – The actuator's msg_type class.

> **Return type**
>> Callable

class eagerx.core.register.**engine_states**(***engine_states*)

> A decorator to register the engine states of an *Object*.
>
> The agnostic() method should be decorated.
>
>> **Parameters**
>>> **engine_states** – The engine state's msg_type class.
>>
>> **Return type**
>>> Callable

class eagerx.core.register.**engine**(*engine_cls*, *entity=None*)

> A decorator to register an engine implementation of an *Object*.
>
> ---
>
> **Note:** In our running example, the *example_engine()* method would be decorated.
>
> ---
>
>> **Parameters**
>>
>> - **engine_cls** (*Engine*) – The Engine's subclass (not the baseclass *Engine*).
>>
>> - **entity** – The entity that corresponds to the engine implementation. If left unspecified, the engine is registered to the class that owns the method.
>>
>> **Return type**
>>> Callable

Message

class eagerx.utils.utils.**Msg**(*info:* Info, *msgs: List[Any]*)

> A dataclass representing a (windowed) input that is passed to *callback()*.
>
> **property info**
>> Info on the received messages in *msgs*.
>
> **property msgs**
>> The received messages with indexing *msgs[-1]* being the most recent message and *msgs[0]* the oldest.

class eagerx.utils.utils.**Info**(*name: Optional[str] = None*, *node_tick: Optional[int] = None*, *rate_in: Optional[float] = None*, *t_node: Optional[List[*Stamp*]] = None*, *t_in: Optional[List[*Stamp*]] = None*, *done: Optional[bool] = None*)

> A dataclass containing info about the received messages in *msgs*.
>
> **property name**
>> Name of the registered input.
>
> **property node_tick**
>> Number of times *callback()* has been called since the last reset.
>
> **property rate_in**
>> Rate (Hz) of the input.
>
> **property t_in**
>> Simulated timestamp that states at what time the message was received according to *rate_in* and *seq*.

> **property t_node**
>
> Simulated timestamp that states during which cycle the message was received since the last reset according to *rate* and *node_tick*.

class eagerx.utils.utils.**Stamp**(*seq: Optional[int] = None, sc: Optional[float] = None, wc: Optional[float] = None*)

> A dataclass for timestamping received messages.
>
> **property sc**
>
> Timestamp according to the simulated clock (seconds). This time is scaled by the real-time factor if > 0.
>
> **property seq**
>
> Sequence number of received message.
>
> **property wc**
>
> Timestamp according to the wall clock (seconds).

2.4 Code Examples

Below you can find a code example of environment creation and training using Stable-Baselines3. To run this code, you should install eagerx_tutorials, which can be done by running:

```
pip3 install eagerx_tutorials
```

Detailed explanation of the code can be found in this Colab tutorial.

```python
import eagerx
from eagerx.backends.single_process import SingleProcess
from eagerx.wrappers import Flatten
from eagerx_tutorials.pendulum.objects import Pendulum
from eagerx_ode.engine import OdeEngine

import stable_baselines3 as sb3
import numpy as np
from typing import Dict


class PendulumEnv(eagerx.BaseEnv):
    def __init__(self, name: str, rate: float, graph: eagerx.Graph, engine: eagerx.specs.
→EngineSpec,
                 backend: eagerx.specs.BackendSpec):
        self.max_steps = 100
        self.steps = None
        super().__init__(name, rate, graph, engine, backend, force_start=True)

    def step(self, action: Dict):
        observation = self._step(action)
        self.steps += 1

        # Calculate reward and check if the episode is terminated
        th = observation["angle"][0]
        thdot = observation["angular_velocity"][0]
        u = float(action["voltage"])
```

(continues on next page)

```python
        th -= 2 * np.pi * np.floor((th + np.pi) / (2 * np.pi))
        cost = th ** 2 + 0.1 * thdot ** 2 + 0.01 * u ** 2
        truncated = self.steps > self.max_steps
        terminated = False

        # Render
        if self.render_mode == "human":
            self.render()
        return observation, -cost, terminated, truncated, {}

    def reset(self, seed=None, options=None) -> Dict:
        states = self.state_space.sample()
        observation = self._reset(states)
        self.steps = 0
        # Render
        if self.render_mode == "human":
            self.render()
        return observation, {}

if __name__ == "__main__":
    rate = 30.0

    pendulum = Pendulum.make("pendulum", actuators=["u"], sensors=["theta", "theta_dot"],
→ states=["model_state"])

    graph = eagerx.Graph.create()
    graph.add(pendulum)
    graph.connect(action="voltage", target=pendulum.actuators.u)
    graph.connect(source=pendulum.sensors.theta, observation="angle")
    graph.connect(source=pendulum.sensors.theta_dot, observation="angular_velocity")

    engine = OdeEngine.make(rate=rate)
    backend = SingleProcess.make()

    env = PendulumEnv(name="PendulumEnv", rate=rate, graph=graph, engine=engine,
→backend=backend)
    env = Flatten(env)

    model = sb3.SAC("MlpPolicy", env, verbose=1)
    model.learn(total_timesteps=int(150 * rate))

    env.shutdown()
```

2.5 Troubleshooting

Here we list commonly encountered problems and effective methods for debugging.

- When developing, users are advised to select the `SingleProcess` *Backend*. Other backends, such as the `Ros1` *Backend* can make debugging unnecessarily hard due to their distributed capabilities. Switch to multi-processing and distributed computing once you have a stable implementation.

- If you must debug using the `Ros1` *Backend*, then you are advised to launch all nodes in the ENVIRONMENT process. See *process* for more info.

- Live-plotting is currently only supported when the `Ros1` *Backend* is selected.

- To run your code using the `Ros1` *Backend* from within PyCharm, make sure to modify your launcher file as described here. This will also allow you to attach a debugger and set breakpoints. Instructions for several other IDEs are also covered in the provided link.

- Using eagerx with anaconda can produce warnings (see below) when rendering or when using the GUI. This is a known issue that is caused by the interaction of pyqtgraph (used in the GUI) and opencv (used for rendering) with Qt libraries. Code seems not to break, so as a temporary fix, you are advised to suppress this error. Please file a bug report if eagerx/opencv/gui functionality actually breaks.

```
QObject::moveToThread: Current thread (0x7fb6c4009eb0) is not the object's thread
↪(0x7fb6c407cf40). Cannot move to
target thread (0x7fb6c4009eb0).
```

2.6 Contributing to EAGERx

2.6.1 Creating a Package

In this section we will describe how to create an EAGERx package, in this case the eagerx_ode package. This package will contain the OdeEngine for simulating systems based on Ordinary Differential Equations (ODEs). Since the OdeEngine will be a generic engine that can be useful for others, we will create a public repository for the OdeEngine.

Template

We will start by creating a new repository for this Python package, using the template that is available here.

As you can see, this template repository already contains some folders and files. The main benefit of using this template, is that it facilitates to perform continuous integration and provides a clear code structure. Since the package is just a Python package in the end, any other Python package structure could be used.

In our case, we create a new repository called eagerx_ode using this template. Since we want to create a package named *eagerx_ode* and not *eagerx_template*, we do the following:

- Rename the folder *eagerx_template* to *eagerx_ode*.

- Update the PACKAGE_NAME variable in Makefile to be *eagerx_ode* instead of *eagerx_template*.

Fig. 9: Screenshot of the EAGERx template package on Github.

Poetry

Next we will create a Python package using Poetry. If you are not familiar with Poetry, we recommend to check out this article. It is a very convenient tool for package management. In the remainder of this section it is assumed that Poetry is installed.

Next, we modify the pyproject.toml file to specify dependencies, add a short description, state the authors of the package etc. . Here we specify scipy as dependencies, since we will be using scipy to perform the integration of the ODEs. This results following pyproject.toml.

Now we are ready to start coding! Note that you can always add or update dependencies later using Poetry.

After adding the source code, installing the package is simple (from the root of the repository):

```
poetry install
```

Note: This will install the package and its dependencies in a virtual environment, see https://python-poetry.org/docs/basic-usage/#using-your-virtual-environment.

Black

In the *eagerx_template*, we also make use of black. According to their docs:

"By using Black, you agree to cede control over minutiae of hand-formatting. In return, Black gives you speed, determinism, and freedom from pycodestyle nagging about formatting. You will save time and mental energy for more important matters."

It allows to automatically format your code such that it satisfies the *Black* code style requirements and allows to check these. In the *eagerx_template* this can be done as follows. First, we install the package using *Poetry*:

```
poetry install
```

Next, we activate the poetry environment that is created during installation:

```
poetry shell
```

Now we can format the code using black:

```
make codestyle
```

Also, we can check the code style:

```
make check-codestyle
```

Note: A number of Github workflows are present within the *eagerx_template*. One of them checks for code style using *Black*. Therefore, when using this template for a public Github repository, don't forget to run: *make codestyle* before pushing your code.

pytest

Also, the *eagerx_template* allows to easily add tests using pytest. You can add your own tests to the tests folder. Only a dummy test is currently present here. You can run the test as follows (from the root of the repository):

First, we install the package using *Poetry* (if you haven't done so yet):

```
poetry install
```

Next, we activate the poetry environment that is created during installation:

```
poetry shell
```

Now we run the tests:

```
make pytest
```

Note: A number of Github workflows are present within the *eagerx_template*. One of them checks if the tests are passing. So before pushing your code, you can check whether the tests are passing locally by running *make pytest*.

Note: Be aware that in order to use a *Node*, *EngineNode* or any other enitity from eagerx.core.entities you have created, that they should be imported before making them using make() with the corresponding ID. Therefore, we advice to import these in the *__init__.py* as is done for example here.

CITE EAGERX

If you are using EAGERx for your scientific publications, please cite:

```
@article{eagerx,
    author  = {van der Heijden, Bas and Luijkx, Jelle, and Ferranti, Laura and Kober,
↪Jens and Babuska, Robert},
    title = {EAGERx: Engine Agnostic Graph Environments for Robotics},
    year = {2022},
    publisher = {GitHub},
    journal = {GitHub repository},
    howpublished = {\url{https://github.com/eager-dev/eagerx}}
}
```

FOUR

MAINTAINERS

EAGERx is currently maintained by Bas van der Heijden ([@bheijden](#)) and Jelle Luijkx ([@jelledouwe](#)).

HOW TO CONTACT US

For any question, send an e-mail to eagerx.dev@gmail.com.

ACKNOWLEDGEMENTS