Gym-WiPE Documentation

bjoluc

Jun 14, 2019

Documentation

CHAPTER 1

Introduction

Gym-WiPE (Gym Wireless Plant Environment) is an OpenAI Gym environment for the application of reinforcement learning in the simulation of wireless networked feedback control loops. It is written in Python.

1.1 Why Gym-WiPE?

Networked control systems often put high requirements on the underlying networks. Off-the-shelf wireless network solutions, however, may not fulfill their needs without further improvements. Reinforcement learning may help to find appropriate policies for radio resource management in control systems for which optimal static resource management algorithms can not easily be determined. This is where Gym-WiPE comes in: It provides simulation tools for the creation of OpenAI Gym reinforcement learning environments that simulate wireless networked feedback control loops.

1.2 What's included?

Gym-WiPE features an all-Python wireless network simulator based on SimPy. The Open Dynamics Engine (ODE), more specifically its Python wrapper Py3ODE is integrated for plant simulation. Two Gym environments have been implemented for frequency band assignments yet: A simplistic network-only example and a (yet untested) environment for frequency band assignments to a sensor and a controller of an inverted pendulum. The development of further environments may concern frequency band assignments but is not limited to these as the entire simulation model is accessible from within Python and may be used for arbitrary Gym wireless networked control environments. Control algorithm implementations may profit from the python-control project.

1.3 Getting started

1.3.1 Environment Setup

Gym-WiPE uses pipenv. To install it, run

```
pip install pipenv
```

With *pipenv* installed, you may clone the repository like

```
git clone https://github.com/bjoluc/gymwipe.git
cd gymwipe
```

and invoke pipenv to set up a virtual environment and install the dependencies into it:

```
pipenv install
```

Optionally, the development dependencies may be installed via

```
pipenv install --dev
```

If ODE is used for plant Simulation, it has to be downloaded and built. After that, `make ode` will install Py3ODE and pygame for plant visualizations.

1.3.2 Running the tests

The pytest testsuite can be executed via `make test`.

1.4 Further steps

This project lacks tutorials. For now, you can have a look at the API documentation at https://gymwipe.readthedocs.io/en/latest/api/index.html. An example agent implementation for a Gym-WiPE environment is provided in the *agents* directory.

CHAPTER 2

API Reference

2.1 gymwipe.control package

2.1.1 Submodules

gymwipe.control.inverted_pendulum module

Controller implementations for an inverted pendulum (using the `gymwipe.plants.sliding_pendulum` module)

class InvertedPendulumPidController(*name*, *xPos*, *yPos*, *frequencyBand*)

Bases: `gymwipe.networking.devices.SimpleNetworkDevice`

A PID inverted pendulum controller for the SlidingPendulum plant.

Note: After initialization, the `sensorAddr` and `actuatorAddr` attributes have to be set to the network addresses of the sensor and the actuator.

sensorAddr = None

The sensor's network address

Type bytes

actuatorAddr = None

The actuator's network address

Type bytes

onReceive(*packet*)

This method is invoked whenever `receiving` is `True` and a packet has been received.

Note: After `receiving` has been set to `False` it might still be called within `RECEIVE_TIMEOUT` seconds.

Parameters **`packet`** (*`Packet`*) – The packet that has been received

`control``()`

2.1.2 Module contents

Implementations of Control Loop Components

2.2 gymwipe.devices package

2.2.1 Submodules

gymwipe.devices.core module

Core components for modelling physical devices

`class Position``(`*x*, *y*, *owner=None*`)`

Bases: `object`

A simple class for representing 2-dimensional positions, stored as two float values.

Parameters

- **`x`** (`Union[float, int]`) – The distance to a fixed origin in x direction, measured in meters
- **`y`** (`Union[float, int]`) – The distance to a fixed origin in y direction, measured in meters
- **`owner`** (`Optional[Any]`) – The object owning (having) the position.

`nChange = None`

A notifier that is triggered when one or both of *`x`* and *`y`* is changed, providing the triggering position object.

Type *`Notifier`*

`x`

The distance to a fixed origin in x direction, measured in meters

Note: When setting both *`x`* and *`y`*, please use the *`set()`* method to trigger *`nChange`* only once.

Type float

`y`

The distance to a fixed origin in y direction, measured in meters

Note: When setting both *`x`* and *`y`*, please use the *`set()`* method to trigger *`nChange`* only once.

Type float

set (*x*, *y*)
Sets the x and the y value triggering the `nChange` notifier only once.

distanceTo (*p*)
Returns the euclidean distance of this `Position` to *p*, measured in meters.

Parameters **p** (`Position`) – The `Position` object to calculate the distance to

Return type `float`

class Device (*name*, *xPos*, *yPos*)
Bases: `object`

Represents a physical device that has a name and a position.

Parameters

- **name** (`str`) – The device name
- **xPos** (`float`) – The device's physical x position
- **yPos** (`float`) – The device's physical y position

name = None
The device name (for debugging and plotting)

Type str

position
The device's physical position

Type `Position`

2.2.2 Module contents

2.3 gymwipe.envs package

2.3.1 Submodules

gymwipe.envs.core module

class BaseEnv (*frequencyBand*, *deviceCount*)
Bases: `gym.core.Env`

A subclass of the OpenAI gym environment that models the Radio Resource Manager frequency band assignment problem. It sets a frequency band and an action space (depending on the number of devices to be used for frequency band assignment).

The action space is a dict space of two discrete spaces: The device number and the assignment duration.

Parameters

- **band** (*frequency*) – The physical frequency band to be used for the simulation
- **deviceCount** (`int`) – The number of devices to be included in the environment's action space

metadata = {'render.modes': ['human']}

MAX_ASSIGN_DURATION = 20

ASSIGNMENT_DURATION_FACTOR = 1000

seed(*seed=None*)
Sets the seed for this environment's random number generator and returns it in a single-item list.

render(*mode='human'*, *close=False*)
Renders the environment to stdout.

class Interpreter
Bases: `abc.ABC`

An *Interpreter* is an instance that observes the system's behavior by sniffing the packets received by the RRM's physical layer and infers observations and rewards for a frequency band assignment learning agent. Thus, RRM and learning agent can be used in any domain with only swapping the interpreter.

This class serves as an abstract base class for all *Interpreter* implementations.

When implementing an interpreter, the following three methods have to be overridden:

- *onPacketReceived()*
- *getReward()*
- *getObservation()*

The following methods provide default implementations that you might also want to override depending on your use case:

- *reset()*
- *onFrequencyBandAssignment()*
- *getDone()*
- *getInfo()*

onPacketReceived(*senderIndex*, *receiverIndex*, *payload*)
Is invoked whenever the RRM receives a packet that is not addressed to it.

Parameters

- **senderIndex** (`int`) – The device index of the received packet's sender (as in the gym environment's action space)
- **receiverIndex** (`int`) – The device index of the received packet's receiver (as in the gym environment's action space)
- **payload** (*Transmittable*) – The received packet's payload

onFrequencyBandAssignment(*deviceIndex*, *duration*)
Is invoked whenever the RRM assigns the frequency band.

Parameters

- **deviceIndex** (`int`) – The index (as in the gym environment's action space) of the device that the frequency band is assigned to.
- **duration** (`int`) – The duration of the assignment in multiples of *TIME_SLOT_LENGTH*

getReward()
Returns a reward that depends on the last channel assignment.

Return type `float`

getObservation()
Returns an observation of the system's state.

Return type `Any`

`getDone``()`

Returns whether an episode has ended.

Note: Reinforcement learning problems do not have to be split into episodes. In this case, you do not have to override the default implementation as it always returns `False`.

Return type `bool`

`getInfo``()`

Returns a `dict` providing additional information on the environment's state that may be useful for debugging but is not allowed to be used by a learning agent.

Return type `Dict`[~KT, ~VT]

`getFeedback``()`

You may want to call this at the end of a frequency band assignment to get feedback for your learning agent. The return values are ordered like they need to be returned by the `step()` method of a gym environment.

Return type `Tuple`[`Any`, `float`, `bool`, `Dict`[~KT, ~VT]]

Returns A 4-tuple with the results of *`getObservation()`*, *`getReward()`*, *`getDone()`*, and *`getInfo()`*

`reset``()`

This method is invoked when the environment is reset – override it with your initialization tasks if you feel like it.

gymwipe.envs.counter_traffic module

A simple Gym environment using the *Simple* network devices for demonstration purposes

`class CounterTrafficEnv`

Bases: *`gymwipe.envs.core.BaseEnv`*

An environment for testing reinforcement learning with three devices:

- Two network devices that send a configurable amount of data to each other
- A simple RRM operating an interpreter for that use case

Optimally, a learning agent will fit the length of the assignment intervals to the amount of data sent by the devices.

`COUNTER_INTERVAL = 0.001`

`COUNTER_BYTE_LENGTH = 2`

`COUNTER_BOUND = 65536`

`class SenderDevice``(`*name*, *xPos*, *yPos*, *frequencyBand*, *packetMultiplicity*`)`

Bases: *`gymwipe.networking.devices.SimpleNetworkDevice`*

A device sending packets with increasing COUNTER_BYTE_LENGTH-byte integers. Every *COUNTER_INTERVAL* seconds, a packet with the current integer is sent *packetMultiplicity* times.

`senderProcess``()`

class CounterTrafficInterpreter(*env*)

Bases: *gymwipe.envs.core.Interpreter*

reset()

This method is invoked when the environment is reset – override it with your initialization tasks if you feel like it.

onPacketReceived(*senderIndex*, *receiverIndex*, *payload*)

Is invoked whenever the RRM receives a packet that is not addressed to it.

Parameters

- **senderIndex** (`int`) – The device index of the received packet's sender (as in the gym environment's action space)
- **receiverIndex** (`int`) – The device index of the received packet's receiver (as in the gym environment's action space)
- **payload** (*Transmittable*) – The received packet's payload

onFrequencyBandAssignment(*deviceIndex*, *duration*)

Is invoked whenever the RRM assigns the frequency band.

Parameters

- **deviceIndex** (`int`) – The index (as in the gym environment's action space) of the device that the frequency band is assigned to.
- **duration** (`int`) – The duration of the assignment in multiples of *TIME_SLOT_LENGTH*

getReward()

Reward depends on the change of the difference between the values received from both devices: If the difference became smaller, it is the positive reward difference, limited by 10. Otherwise, it is the negative reward difference, limited by -10. This is a result of trial and error and most likely far away from being perfect.

getObservation()

Returns an observation of the system's state.

getDone()

Returns whether an episode has ended.

Note: Reinforcement learning problems do not have to be split into episodes. In this case, you do not have to override the default implementation as it always returns `False`.

getInfo()

Returns a `dict` providing additional information on the environment's state that may be useful for debugging but is not allowed to be used by a learning agent.

reset()

Resets the state of the environment and returns an initial observation.

step(*action*)

Run one timestep of the environment's dynamics. When end of episode is reached, you are responsible for calling *reset()* to reset this environment's state.

Accepts an action and returns a tuple (observation, reward, done, info).

Parameters action (*object*) – an action provided by the agent

Returns agent's observation of the current environment reward (float) : amount of reward returned after previous action done (bool): whether the episode has ended, in which case further step() calls will return undefined results info (dict): contains auxiliary diagnostic information (helpful for debugging, and sometimes learning)

Return type observation (object)

`render`(*mode='human'*, *close=False*)
Renders the environment to stdout.

gymwipe.envs.inverted_pendulum module

A Gym environment for frequency band assignments to a sensor and a controller in the wireless networked control of an inverted pendulum

`class InvertedPendulumInterpreter`(*env*)
Bases: `gymwipe.envs.core.Interpreter`

`onPacketReceived`(*senderIndex*, *receiverIndex*, *payload*)
No actions for received packets, as we read sensor angles directly from the plant object.

`onFrequencyBandAssignment`(*deviceIndex*, *duration*)
Is invoked whenever the RRM assigns the frequency band.

Parameters

- **`deviceIndex`** (`int`) – The index (as in the gym environment's action space) of the device that the frequency band is assigned to.
- **`duration`** (`int`) – The duration of the assignment in multiples of *`TIME_SLOT_LENGTH`*

`getReward`()
Reward is $|180 - \alpha|$ with α being the pendulum angle.

`getObservation`()
Returns an observation of the system's state.

`getDone`()
Returns whether an episode has ended.

Note: Reinforcement learning problems do not have to be split into episodes. In this case, you do not have to override the default implementation as it always returns `False`.

`getInfo`()
Returns a `dict` providing additional information on the environment's state that may be useful for debugging but is not allowed to be used by a learning agent.

`class InvertedPendulumEnv`
Bases: `gymwipe.envs.core.BaseEnv`

An environment that allows an agent to assign a frequency band to a sliding pendulum's *`AngleSensor`* and an *`InvertedPendulumPidController`*

Note: This environment is yet untested!

`reset`()
Resets the state of the environment and returns an initial observation.

`step`(*action*)
Run one timestep of the environment's dynamics. When end of episode is reached, you are responsible for calling *reset()* to reset this environment's state.

Accepts an action and returns a tuple (observation, reward, done, info).

Parameters **action** (*object*) – an action provided by the agent

Returns agent's observation of the current environment reward (float) : amount of reward returned after previous action done (bool): whether the episode has ended, in which case further step() calls will return undefined results info (dict): contains auxiliary diagnostic information (helpful for debugging, and sometimes learning)

Return type observation (object)

render (*mode='human'*, *close=False*)
Renders the environment to stdout.

2.3.2 Module contents

2.4 gymwipe.networking package

2.4.1 Submodules

gymwipe.networking.attenuation_models module

A collection of *AttenuationModel* implementations. Currently contains:

FsplAttenuation(frequencyBandSpec, deviceA, ...)	A free-space path loss (FSPL) `AttenuationModel` implementation

class FsplAttenuation (*frequencyBandSpec*, *deviceA*, *deviceB*)
Bases: *gymwipe.networking.physical.PositionalAttenuationModel*

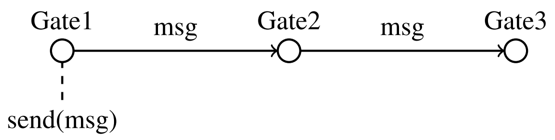
A free-space path loss (FSPL) `AttenuationModel` implementation

gymwipe.networking.construction module

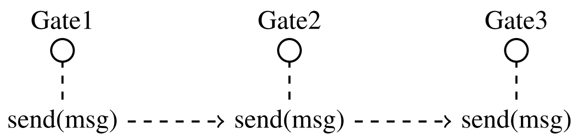
The construction module provides classes for building network stack representations. The concept of modules, compound modules, and gates is borrowed from the OMNeT++ model structure, which is described in [VH08].

class Gate (*name*, *owner=None*)
Bases: `object`

Gates provide features for the transfer of arbitrary objects. They can be connected to each other and offer a *send()* method that passes an object to all connected gates, as shown in the figure below, where connections are depicted as arrows.



Gates emulate the transmission of objects via connections by calling *send()* on their connected gates as illustrated below.



name
The Gate's name

Type str

nReceives
A notifier that is triggered when `send()` is called, providing the value passed to `send()`

Type *gymwipe.simtools.Notifier*

nConnectsTo
A notifier that is triggered when `connectTo()` is called, providing the gate passed to `connectTo()`

Type `Notifier`

connectTo(*gate*)
Connects this `Gate` to the provided `Gate`. Thus, if `send()` is called on this `Gate`, it will also be called on the provided `Gate`.

Parameters **gate** (`Gate`) – The `Gate` for the connection to be established to

send(*object*)
Triggers `nReceives` with the provided object and sends it to all connected gates.

class Port(*name*, *owner=None*)
Bases: `object`

A `Port` simplifies the setup of bidirectional connections by wrapping an input and an output `Gate` and offering two connection methods: `biConnectWith()` and `biConnectProxy()`.

Parameters

- **name** (`str`) – The `Port`'s name
- **owner** (`Any`) – The object that the `Port` belongs to (e.g. a `Module`)

name
The port's name, as provided to the constructor

Type `str`

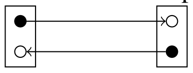
input
The port's input `Gate`

output
The port's output `Gate`

biConnectWith(*port*)
Shorthand for

```
self.output.connectTo(port.input)
port.output.connectTo(self.input)
```

Creates a bidirectional connection between this port an the passed port. If ○ indicates input gates and ● indicates output gates, the resulting connection between two ports can be visualized like this:

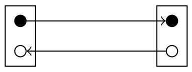


Parameters **port** (`Port`) – The `Port` to establish the bidirectional connection to

biConnectProxy(*port*)
Shorthand for

```
self.output.connectTo(port.output)
port.input.connectTo(self.input)
```

If ○ indicates input gates and ● indicates output gates, the resulting connection between two ports can be visualized like this:



Parameters **port** (*Port*) – The *Port* to establish the bidirectional proxy connection to

nReceives

The input *Gate*'s *nReceives Notifier*, which is triggered when an object is received by the input *Gate*.

Type *Notifier*

class GateListener(*gateName*, *validTypes=None*, *blocking=True*, *queued=False*)

Bases: `object`

A factory for decorators that allow to call a module's method (or process a SimPy generator method) whenever a specified gate of a *Module* receives an object. The received object is provided to the decorated method as a parameter.

Note: In order to make this work for a class' methods, you have to decorate that class' constructor with *@PortListener.setup*.

Examples

A *Module*'s method using this decorator could look like this:

```
@GateListener("myPortIn")
def myPortInListener(self, obj):
    # This  method is processed whenever self.gates["myPortIn"]
    # receives an object and all previously created instances
    # have been processed.
    yield SimMan.timeout(1)
```

Parameters

- **gateName** (`str`) – The index of the module's *Gate* to listen on
- **validTypes** (`Union[type, Tuple[type], None]`) – If this argument is provided, a `TypeError` will be raised when an object received via the specified *Gate* is not of the `type` / one of the types specified.
- **blocking** (`bool`) – Set this to `False` if you decorate a SimPy generator method and want it to be processed for each received object, regardless of whether an instance of the generator is still being processed or not. By default, only one instance of the decorated generator method is run at a time (blocking is `True`).
- **queued** (`bool`) – If you decorate a SimPy generator method, *blocking* is `True`, and you set *queued* to `True`, an object received while an instance of the generator is being processed will be queued. Sequentially, a new generator instance will then be processed for every queued object as soon as a previous generator instance has been processed. Using

queued, you can, for example, react to multiple objects that are received at the same simulated time, while only having one instance of a subscribed generator method processed at a time. Queued defaults to `False`.

static setup(*function*)
A decorator to be used for the constructors of *`Module`* subclasses that apply the *`GateListener`* decorator.

class Module(*name*, *owner=None*)
Bases: `object`

Modules are used to model components that interact with each other, as for example network stack layers. A module has a number of ports and gates that can be used to exchange data with it and connect it to other modules. Modules provide the methods *`_addPort()`* and *`_addGate()`* that allow to add ports and gates, which can be accessed via the *`ports`* and the *`gates`* dictionaries.

Note: Modules may have both ports (for bidirectional connections) and individual gates (for unidirectional connections). When a port is added by *`_addPort()`*, its two gates are also added to the *`gates`* dictionary.

---

name
The Module's name

Type str

ports
The Module's outer Ports

Type Dict[str, *Port*]

gates
The Module's outer Gates

Type Dict[str, *Gate*]

_addPort(*name*)
Adds a new *`Port`* to the *`ports`* dictionary, indexed by the name passed. Since a *`Port`* holds two *`Gate`* objects, a call of this method also adds two entries to the *`gates`* dictionary, namely "<name>In" and "<name>Out".

Parameters name (`str`) – The name for the *`Port`* to be indexed with

_addGate(*name*)
Adds a new *`Gate`* to the *`gates`* dictionary, indexed by the name passed.

Note: Plain *`Gate`* objects are only needed for unidirectional connections. Bidirectional connections can profit from *`Port`* objects.

Parameters name (`str`) – The name for the *`Gate`* to be indexed with

class CompoundModule(*name*, *owner=None*)
Bases: *`gymwipe.networking.construction.Module`*

A *`CompoundModule`* is a *`Module`* that contains an arbitrary number of submodules (*`Module`* objects) which can be connected with each other and their parent module's gates and ports. Submodules are added using *`_addSubmodule()`* and can be accessed via the *`submodules`* dictionary.

Note: When subclassing `CompoundModule`, do not directly implement functionalities in your subclass, but wrap them in submodules to ensure modularity. Also, do not connect a CompoundModule's submodules to anything else than other submodules or the CompoundModule itself for the same reason.

`submodules`
The `CompoundModule`'s nested `Module` objects

Type Dict[str, *Module*]

`_addSubmodule`(*name*, *module*)
Adds a new `Module` to the `submodules` dictionary, indexed by the name passed.

Parameters

- **name** (`str`) – The name for the submodule to be indexed with
- **module** (`Module`) – The `Module` object to be added as a submodule

gymwipe.networking.devices module

`Device` implementations for network devices

`class NetworkDevice`(*name*, *xPos*, *yPos*, *frequencyBand*)
Bases: `gymwipe.devices.core.Device`

A subclass of `Device` that extends the constructor's parameter list by a *frequencyBand* argument. The provided `FrequencyBand` object will be stored in the `frequencyBand` attribute.

Parameters

- **name** (`str`) – The device name
- **xPos** (`float`) – The device's physical x position
- **yPos** (`float`) – The device's physical y position
- **band** (*frequency*) – The `FrequencyBand` instance that will be used for transmissions

`frequencyBand = None`
The `FrequencyBand` instance that is used for transmissions

Type `FrequencyBand`

`class SimpleNetworkDevice`(*name*, *xPos*, *yPos*, *frequencyBand*)
Bases: `gymwipe.networking.devices.NetworkDevice`

A `NetworkDevice` implementation running a network stack that consists of a SimplePHY and a SimpleMAC. It offers a method for sending a packet using the MAC layer, as well as a callback method that will be invoked when a packet is received. Also, receiving can be turned on or of by setting `receiving` either to `True` or to `False`.

Parameters

- **name** (`str`) – The device name
- **xPos** (`float`) – The device's physical x position
- **yPos** (`float`) – The device's physical y position
- **band** (*frequency*) – The `FrequencyBand` instance that will be used for transmissions

`macAddr = None`
The address that is used by the MAC layer to identify this device

Type bytes

`RECEIVE_TIMEOUT = 100`
The timeout in seconds for the simulated blocking MAC layer receive call

Type int

`receiving`

Return type `bool`

`send`(*data*, *destinationMacAddr*)

`onReceive`(*packet*)
This method is invoked whenever *`receiving`* is `True` and a packet has been received.

Note: After *`receiving`* has been set to `False` it might still be called within *`RECEIVE_TIMEOUT`* seconds.

Parameters **`packet`** (*`Packet`*) – The packet that has been received

`class SimpleRrmDevice`(*name*, *xPos*, *yPos*, *frequencyBand*, *deviceIndexToMacDict*, *interpreter*)
Bases: *`gymwipe.networking.devices.NetworkDevice`*

A Radio Resource Management *`NetworkDevice`* implementation. It runs a network stack consisting of a SimplePHY and a SimpleRrmMAC. It offers a method for frequency band assignment and operates an *`Interpreter`* instance that provides observations and rewards for a learning agent.

Parameters

- **`name`** (`str`) – The device name
- **`xPos`** (`float`) – The device's physical x position
- **`yPos`** (`float`) – The device's physical y position
- **`band`** (*`frequency`*) – The *`FrequencyBand`* instance that will be used for transmissions
- **`deviceIndexToMacDict`** (`Dict[int, bytes]`) – A dictionary mapping integer indexes to device MAC addresses. This allows to pass the device index used by a learning agent instead of a MAC address to *`assignFrequencyBand()`*.
- **`interpreter`** (*`Interpreter`*) – The *`Interpreter`* instance to be used for observation and reward calculations

`interpreter = None`
The *`Interpreter`* instance that provides domain-specific feedback on the consequences of *`assignFrequencyBand()`* calls

Type *`Interpreter`*

`deviceIndexToMacDict = None`
A dictionary mapping integer indexes to device MAC addresses. This allows to pass the device index used by a learning agent instead of a MAC address to *`assignFrequencyBand()`*.

`macToDeviceIndexDict = None`
The counterpart to *`deviceIndexToMacDict`*

`macAddr`
The RRM's MAC address

Type bytes

Return type `bytes`

`assignFrequencyBand`(*deviceIndex*, *duration*)
Makes the RRM assign the frequency band to a certain device for a certain time.

Parameters

- **`deviceIndex`** (`bytes`) – The integer id that maps to the MAC address of the device to assign the frequency band to (see *`deviceIndexToMacDict`*)
- **`duration`** (`int`) – The number of time units for the frequency band to be assigned to the device

Return type `Tuple[Any, float]`

Returns The `Signal` object that was used to make the RRM MAC layer assign the frequency band. When the frequency band assignment is over, the signal's `eProcessed` event will succeed.

gymwipe.networking.messages module

The messages module provides classes for network packet representations and inter-module communication.

The following classes are used for transmission simulation:

`Transmittable`(value[, byteSize])	The *`Transmittable`* class provides a `byteSize` attribute allowing the simulated sending of *`Transmittable`* objects via a frequency band.
`FakeTransmittable`(byteSize)	A *`Transmittable`* implementation that sets its value to None.
`Packet`(header, payload[, trailer])	The Packet class represents packets.
`SimpleMacHeader`(sourceMAC, destMAC, flag)	A class for representing MAC packet headers
`SimpleNetworkHeader`(sourceMAC, destMAC)	Since no network protocol is implemented in Gym-WiPE, there is a need for some interim way to specify source and destination addresses in packets that are passed to the `SimpleMAC` layer.

The following classes are used for inter-module communication:

`Message`(type[, args])	A class used for the exchange of arbitrary messages between components.
`StackMessageTypes`	An enumeration of control message types to be used for the exchange of *Message* objects between network stack layers.

class `Transmittable`(*value*, *byteSize=None*)
Bases: `object`

The *`Transmittable`* class provides a *`byteSize`* attribute allowing the simulated sending of *`Transmittable`* objects via a frequency band.

`value`
The object that has been passed to the constructor as *value*

Type Any

`byteSize`

The transmittable's byteSize as it was passed to the constructor

Parameters

- **`value`** (`Any`) – The object of which the string representation will be used
- **`byteSize`** – The number of bytes that are simulated to be transmitted when the data represented by this *`Transmittable`* is sent via a frequency band. Defaults to the length of the UTF-8 encoding of *str(value)*.

`bitSize`
`byteSize` $\times 8$

Return type `int`

`transmissionTime`(*bitrate*)
Returns the time in seconds needed to transmit the data represented by the *`Transmittable`* at the specified bit rate.

Parameters **`bitrate`** (`float`) – The bitrate in bps

Return type `float`

`class FakeTransmittable`(*byteSize*)
Bases: *`gymwipe.networking.messages.Transmittable`*

A *`Transmittable`* implementation that sets its value to None. It can be helpful for test applications when the data itself is irrelevant and only its size has to be considered.

Parameters **`byteSize`** (`int`) – The number of bytes that the *`FakeTransmittable`* represents

`class Packet`(*header*, *payload*, *trailer=None*)
Bases: *`gymwipe.networking.messages.Transmittable`*

The Packet class represents packets. A Packet consists of a header, a payload and an optional trailer. Packets can be nested by providing them as payloads to the packet constructor.

`header`
The object representing the Packet's header

Type *Transmittable*

`payload`
The object representing the Packet's payload. Might be another *`Packet`*.

Type *Transmittable*

`trailer`
The object representing the Packet's trailer (defaults to `None`)

Type *Transmittable*

`__str__`()
Return str(self).

`class SimpleMacHeader`(*sourceMAC*, *destMAC*, *flag*)
Bases: *`gymwipe.networking.messages.Transmittable`*

A class for representing MAC packet headers

`sourceMAC`
The 6-byte-long source MAC address

Type bytes

destMAC
The 6-byte-long destination MAC address

Type bytes

flag
A single byte flag (stored as an integer in range(256))

Type int

class SimpleNetworkHeader(*sourceMAC*, *destMAC*)
Bases: *gymwipe.networking.messages.Transmittable*

Since no network protocol is implemented in Gym-WiPE, there is a need for some interim way to specify source and destination addresses in packets that are passed to the `SimpleMAC` layer. Therefore, a *SimpleNetworkHeader* holds a source and a destination MAC address. The destination address is used by the `SimpleMAC` layer.

sourceMAC
The 6-byte-long source MAC address

Type bytes

destMAC
The 6-byte-long destination MAC address

Type bytes

class Message(*type*, *args=None*)
Bases: `object`

A class used for the exchange of arbitrary messages between components. A *Message* can be used to simulate both asynchronous and synchronous function calls.

type
An enumeration object that defines the message type

Type Enum

args
A dictionary containing the message's arguments

Type Dict[str, Any]

eProcessed
A SimPy event that is triggered when *setProcessed()* is called. This is useful for simulating synchronous function calls and also allows for return values (an example is provided in *setProcessed()*).

Type Event

setProcessed(*returnValue=None*)
Makes the *eProcessed* event succeed.

Parameters **returnValue** (`Optional`[`Any`]) – If specified, will be used as the *value* of the *eProcessed* event.

Examples

If *returnValue* is specified, SimPy processes can use Signals for simulating synchronous function calls with return values like this:

```
signal = Signal(myType, {"key", value})
gate.output.send(signal)
value = yield signal.eProcessed
# value now contains the returnValue that setProcessed() was called with
```

class StackMessageTypes

Bases: `enum.Enum`

An enumeration of control message types to be used for the exchange of *Message* objects between network stack layers.

RECEIVE = 0

SEND = 1

ASSIGN = 2

gymwipe.networking.physical module

Physical-layer-related components

calculateEbToN0Ratio(*signalPower*, *noisePower*, *bitRate*, *returnDb=False*)

Computes $E_b/N_0 = \frac{S}{N_0 R}$ (the "ratio of signal energy per bit to noise power density per Hertz" [Sta05]) given the signal power S_{dBm}, the noise power $N_{0_{dBm}}$, and the bit rate R, according to p. 95 of [Sta05].

Parameters

- **signalPower** (`float`) – The signal power S in dBm
- **noisePower** (`float`) – The noise power N_0 in dBm
- **bitRate** (`float`) – The bit rate R in bps
- **returnDb** (`bool`) – If set to `True`, the ratio will be returned in dB.

Return type `float`

approxQFunction(*x*)

Approximates the gaussian Q-Function for $x \geq 0$ by using the following formula, derived from [KL07]:

$$Q(x) \approx \frac{\left(1-e^{-1.4x}\right)e^{-\frac{x^2}{2}}}{1.135\sqrt{2\pi}x}$$

Parameters x (`float`) – The x value to approximate $Q(x)$ for

Return type `float`

temperatureToNoisePowerDensity(*temperature*)

Calculates the noise power density N_0 in W/Hz for a given temperature T in degrees Celsius according to [Sta05] by using the following formula:

$N_0 = k(T + 273.15)$ with k being Boltzmann's constant

Parameters temperature (`float`) – The temperature T in degrees Celsius

Return type `float`

wattsToDbm(*watts*)

Converts a watt value to a dBm value.

Parameters watts (`float`) – The watt value to be converted

milliwattsToDbm(*milliwatts*)

Converts a milliwatt value to a dBm value.

Parameters **`watts`** – The milliwatt value to be converted

`dbmToMilliwatts`(*milliwatts*)
Converts a dBm value to a milliwatt value.

Parameters **`watts`** – The dBm value to be converted

`class Mcs`(*frequencyBandSpec*, *codeRate*)
Bases: `abc.ABC`

The *`Mcs`* class represents a Modulation and Coding Scheme. As the MCS (beside frequency band characteristics) determines the relation between Signal-to-Noise Ratio (SNR) and the resulting Bit Error Rate (BER), it offers a `getBitErrorRateBySnr()` method that is used by receiving PHY layer instances. *`Mcs`* objects also provide a *`bitRate`* and a *`dataRate`* attribute, which specify the physical bit rate and the effective data rate of transmissions with the corresponding *`Mcs`*.

Currently, only BPSK modulation is implemented (see *`BpskMcs`* for details). Subclass *`Mcs`* if you need something more advanced.

`frequencyBandSpec = None`
The frequency band specification that determines the bandwidth for which the MCS is operated

`codeRate = None`
The relative amount of transmitted bits that are not used for forward error correction

Type Fraction

`calculateBitErrorRate`(*signalPower*, *noisePower*, *bitRate*)
Computes the bit error rate for the passed parameters if this modulation and coding scheme is used.

Parameters

- **`signalPower`** (`float`) – The signal power S in dBm
- **`noisePower`** (`float`) – The noise power N_0 in dBm
- **`bitRate`** (`float`) – The bit rate R in bps

Returns: The estimated resulting bit error rate (a float in [0,1])

Return type `float`

`bitRate`
The physical bit rate in bps that results from the use of this MCS

Type float

Return type `float`

`dataRate`
The effective data rate in bps that results from the use of this MCS (considers coding overhead)

Type float

Return type `float`

`maxCorrectableBer`()
Returns the maximum bit error rate that can be handled when using the MCS. It depends on the codeRate and is calculated via the Varshamov-Gilbert bound.

Return type `float`

`class BpskMcs`(*frequencyBandSpec*, *codeRate=Fraction(3, 4)*)
Bases: *`gymwipe.networking.physical.Mcs`*

A Binary Phase-Shift-Keying MCS

bitRate
The physical bit rate in bps that results from the use of this MCS

Type float

Return type `float`

dataRate
The effective data rate in bps that results from the use of this MCS (considers coding overhead)

Type float

Return type `float`

calculateBitErrorRate(*signalPower*, *noisePower*)
Computes the bit error rate for the passed parameters if this modulation and coding scheme is used.

Parameters

- **signalPower** (`float`) – The signal power S in dBm
- **noisePower** (`float`) – The noise power N_0 in dBm
- **bitRate** – The bit rate R in bps

Returns: The estimated resulting bit error rate (a float in [0,1])

Return type `float`

class Transmission(*sender*, *power*, *packet*, *mcsHeader*, *mcsPayload*, *startTime*)
Bases: `object`

A *`Transmission`* models the process of a device sending a specific packet via a communication frequency band.

Note: The proper way to instantiate *`Transmission`* objects is via *`FrequencyBand.transmit()`*.

sender = None
The device that initiated the transmission

Type *Device*

power = None
The tramsmission power in dBm

Type float

mcsHeader = None
The modulation and coding scheme used for the transmitted packet's header

Type *Mcs*

mcsPayload = None
The modulation and coding scheme used for the transmitted packet's payload

Type *Mcs*

packet = None
The packet sent in the transmission

Type *Packet*

startTime = None
The simulated time at which the transmission started

Type float

headerDuration = None
The time in seconds taken by the transmission of the packet's header

Type float

payloadDuration = None
The time in seconds taken by the transmission of the packet's payload

Type float

duration = None
The time in seconds taken by the transmission

Type float

stopTime = None
The moment in simulated time right after the transmission has completed

Type float

headerBits = None
Transmitted bits for the packet's header (including coding overhead)

payloadBits = None
Transmitted bits for the packet's payload (including coding overhead)

eHeaderCompletes = None
A SimPy event that succeeds at the moment in simulated time right after the packet's header has been transmitted. The transmission object is provided as the value to the `succeed()` call.

Type `Event`

eCompletes = None
A SimPy event that succeeds at *`stopTime`*, providing the transmission object as the value.

Type `Event`

completed
Returns `True` if the transmission has completed (i.e. the current simulation time >= stopTime)

class FrequencyBandSpec (*frequency=2400000000.0*, *bandwidth=22000000.0*)
Bases: `object`

A frequency band specification stores a *`FrequencyBand`*'s frequency and its bandwidth.

Parameters

- **frequency** (`float`) – The frequency band's frequency in Hz. Defaults to 2.4 GHz.
- **bandwidth** (`float`) – The frequency band's bandwidth in Hz. Defaults to 22 MHz (as in IEEE 802.11)

class AttenuationModel (*frequencyBandSpec*, *deviceA*, *deviceB*)
Bases: `object`

An *`AttenuationModel`* calculates the attenuation (measured in db) of any signal sent from one network device to another. It runs a SimPy process and subscribes to the positionChanged events of the `NetworkDevice` instances it belongs to. When the attenuation value changes, the `attenuationChanged` event succeeds.

Parameters

- **frequencyBandSpec** (*`FrequencyBandSpec`*) – The frequency band specification of the corresponding *`FrequencyBand`*

- **deviceA** (*Device*) – Network device a
- **deviceB** (*Device*) – Network device b

Raises `ValueError` – If *deviceA* is *deviceB*

attenuation = None
The attenuation of any signal sent from `NetworkDevice` *deviceA* to `NetworkDevice` *deviceB* (or vice versa) at the currently simulated time, measured in db.

Type float

nAttenuationChanges = None
A notifier that is triggered when the attenuation value changes, providing the new attenuation value.

Type *gymwipe.simtools.Notifier*

class PositionalAttenuationModel (*frequencyBandSpec*, *deviceA*, *deviceB*)
Bases: *gymwipe.networking.physical.AttenuationModel*, `abc.ABC`

An *AttenuationModel* subclass that executes `_positionChanged()` whenever one of its two devices changes its position and the distance between the devices does not exceed *STANDBY_THRESHOLD*.

STANDBY_THRESHOLD = 3000
The minimum distance in metres, that allows the *AttenuationModel* not to react on position changes of its devices

Type float

class JoinedAttenuationModel (*frequencyBandSpec*, *deviceA*, *deviceB*, *models*)
Bases: *gymwipe.networking.physical.AttenuationModel*

An *AttenuationModel* that adds the attenuation values of two or more given *AttenuationModel* instances. If the position of one of both devices is changed, it will gather the update notifications of its *AttenuationModel* instances, sum them up and trigger the `nAttenuationChanges` notifier only once after the updates (this is implemented using callback priorities). When an *AttenuationModel* instance changes its attenuation without reacting to a position update, the `nAttenuationChanges` notifier of the *JoinedAttenuationModel* will be triggered as a direct consequence.

Parameters

- **frequencyBandSpec** (*FrequencyBandSpec*) – The frequency band specification of the corresponding *FrequencyBand*
- **deviceA** (*Device*) – Network device a
- **deviceB** (*Device*) – Network device b
- **models** (`List`[`Type`[~AttenuationModel]]) – A non-empty list of the *AttenuationModel* subclasses to create a *JoinedAttenuationModel* instance of

class AttenuationModelFactory (*frequencyBandSpec*, *models*)
Bases: `object`

A factory for *AttenuationModel* instances.

Parameters

- **frequencyBandSpec** (*FrequencyBandSpec*) – The frequency band specification of the corresponding *FrequencyBand*
- **models** (`List`[~AttenuationModel]) – A non-empty list of *AttenuationModel* subclasses that will be used for instantiating attenuation models.

setCustomModels(*deviceA*, *deviceB*, *models*)

Sets the `AttenuationModel` subclasses for signals sent from *deviceA* to *deviceB* and vice versa.

Note: In order for this method to work, it has to be invoked before an `AttenuationModel` instance is requested for the first time for the specified pair of devices.

Parameters

- **deviceA** (`Device`) – One device
- **deviceB** (`Device`) – Another device
- **models** (`List`[~AttenuationModel]) – A non-empty list of `AttenuationModel` subclasses that will be used for instantiating attenuation models for signals sent from *deviceA* to *deviceB* and vice versa

getInstance(*deviceA*, *deviceB*)

Returns the `AttenuationModel` for signals sent from *deviceA* to *deviceB* and vice versa. If not yet existent, a new `AttenuationModel` instance will be created. If the factory was initialized with multiple `AttenuationModel` subclasses, a `JoinedAttenuationModel` will be handed out.

Return type `AttenuationModel`

class FrequencyBand(*modelClasses*, *frequency=2400000000.0*, *bandwidth=22000000.0*)

Bases: `object`

The `FrequencyBand` class serves as a manager for transmission objects and represents a wireless frequency band. It also offers a `getAttenuationModel()` method that returns a frequency-band-specific AttenuationModel for any pair of devices.

Parameters

- **modelClasses** (`List`[~AttenuationModel]) – A non-empty list `AttenuationModel` subclasses that will be used for attenuation calculations regarding this frequency band.
- **frequency** (`float`) – The frequency band's frequency in Hz. Defaults to 2.4 GHz.
- **bandwidth** (`float`) – The frequency band's bandwidth in Hz. Defaults to 22 MHz (as in IEEE 802.11)

spec = None

The frequency band's specification object

Type `FrequencyBandSpec`

nNewTransmission = None

A notifier that is triggered when `transmit()` is executed, providing the `Transmission` object that represents the transmission.

Type `Notifier`

getAttenuationModel(*deviceA*, *deviceB*)

Returns the AttenuationModel instance that provides attenuation values for transmissions between *deviceA* and *deviceB*.

Return type `AttenuationModel`

transmit(*sender*, *power*, *packet*, *mcsHeader*, *mcsPayload*)

Simulates the transmission of *packet* with the given properties. This is achieved by creating a `Transmission` object with the values passed and triggering the `transmissionStarted` event of the `FrequencyBand`.

Parameters

- **sender** (*Device*) – The device that transmits
- **mcs** – The modulation and coding scheme to be used (represented by an instance of an Mcs subclass)
- **power** (float) – Transmission power in dBm
- **brHeader** – Header bitrate
- **brPayload** – Payload bitrate
- **packet** (*Packet*) – *Packet* object representing the packet being transmitted

Return type *Transmission*

Returns The *Transmission* object representing the transmission

getActiveTransmissions()

Returns a list of transmissions that are currently active.

Return type List[*Transmission*]

getActiveTransmissionsInReach(*receiver*, *radius*)

Returns a list of transmissions that are currently active and whose sender is positioned within the radius specified by *radius* around the receiver.

Parameters

- **receiver** (*Device*) – The NetworkDevice, around which the radius is considered
- **radius** (float) – The radius around the receiver (in metres)

Return type List[*Transmission*]

nNewTransmissionInReach(*receiver*, *radius*)

Returns a notifier that is triggered iff a new *Transmission* starts whose sender is positioned within the radius specified by *radius* around the *receiver*.

Parameters

- **receiver** (*Device*) – The NetworkDevice, around which the radius is considered
- **radius** (float) – The radius around the receiver (in metres)

Return type *Notifier*

gymwipe.networking.simple_stack module

The simple_stack package contains basic network stack layer implementations. Layers are modelled by *gymwipe.networking.construction.Module* objects.

TIME_SLOT_LENGTH = 1e-06

The length of one time slot in seconds (used for simulating slotted time)

Type float

class SimplePhy(*name*, *device*, *frequencyBand*)

Bases: *gymwipe.networking.construction.Module*

A physical layer implementation that does not take propagation delays into account. It provides a port called *mac* to be connected to a mac layer. Slotted time is used, with the length of a time slot being defined by *TIME_SLOT_LENGTH*.

During simulation the frequency band is sensed and every successfully received packet is sent via the *macOut* gate.

The *macIn* gate accepts `Message` objects with the following `StackMessageTypes`:

- `SEND`

  Send a specified packet on the frequency band.

  `Message` args:

  packet The `Packet` object representing the packet to be sent

  power The transmission power in dBm

  mcs The `Mcs` object representing the MCS for the transmission

`NOISE_POWER_DENSITY = 4.0454699999999995e-21`
The receiver's noise power density in Watts/Hertz

Type float

`macInHandler``()`
A SimPy process method which is decorated with the `GateListener` decorator. It is processed when the module's *macIn* `Gate` receives an object.

class `SimpleMac`(*name*, *device*, *frequencyBandSpec*, *addr*)
Bases: `gymwipe.networking.construction.Module`

A MAC layer implementation of the contention-free protocol described as follows:

- Every SimpleMac has a unique 6-byte-long MAC address.
- The MAC layer with address `0` is considered to belong to the RRM.
- Time slots are grouped into frames.
- Every second frame is reserved for the RRM and has a fixed length (number of time slots).
- The RRM uses those frames to send a short *announcement* containing a destination MAC address and the frame length (number of time slots **n**) of the following frame. By doing so it allows the specified device to use the frequency band for the next frame. *Announcements* are packets with a `SimpleMacHeader` having the following attributes:

  `sourceMAC`: The RRM MAC address

  `destMAC`: The MAC address of the device that may transmit next

  `flag`: `1` (flag for allowing a device to transmit)

  The packet's `payload` is the number **n** mentioned above (wrapped inside a `Transmittable`)
- Every other packet sent has a `SimpleMacHeader` with `flag` `0`.

The *networkIn* gate accepts objects of the following types:

- `Message`

  Types:

  – `RECEIVE`

    Listen for packets sent to this device.

    `Message` args:

    duration The time in seconds to listen for

When a packet destinated to this device is received, the `ePROCESSED` event of the `Message` will be triggered providing the packet as the value. If the time given by *duration* has passed and no packet was received, it will be triggered with `None`.

- `Packet`

  Send a given packet (with a `SimpleNetworkHeader`) to the MAC address defined in the header.

The *phyIn* gate accepts objects of the following types:

- `Packet`

  A packet received by the physical layer

Parameters

- **name** (`str`) – The layer's name
- **device** (`Device`) – The device that operates the SimpleMac layer
- **addr** (`bytes`) – The 6-byte-long MAC address to be assigned to this MAC layer

rrmAddr = b'\x00\x00\x00\x00\x00\x00'

The 6 bytes long RRM MAC address

Type bytes

classmethod newMacAddress()

A method for generating unique 6-byte-long MAC addresses (currently counting upwards starting at 1)

Return type `bytes`

phyInHandler()

A SimPy process method which is decorated with the `GateListener` decorator. It is processed when the module's *phyIn* `Gate` receives an object.

networkInHandler()

A method which is decorated with the `GateListener` decorator. It is invoked when the module's *networkIn* `Gate` receives an object.

class SimpleRrmMac(*name*, *device*, *frequencyBandSpec*)

Bases: `gymwipe.networking.construction.Module`

The RRM implementation of the protocol described in `SimpleMac`

The *networkIn* gate accepts objects of the following types:

- `Message`

`StackMessageTypes`:

- `ASSIGN`

  Send a frequency band assignment announcement that permits a device to transmit for a certain time.

  `Message` args:

  dest The 6-byte-long MAC address of the device to be allowed to transmit

  duration The number of time steps to assign the frequency band for the specified device

The payloads of packets from other devices are outputted via the *networkOut* gate, regardless of their destination address. This enables an interpreter to extract observations and rewards for a frequency band assignment learning agent.

addr = None
The RRM's MAC address

Type bytes

phyInHandler()
A method which is decorated with the `GateListener` decorator. It is invoked when the module's *phyIn* `Gate` receives an object.

networkInHandler()
A method which is decorated with the `GateListener` decorator. It is invoked when the module's *networkIn* `Gate` receives an object.

2.4.2 Module contents

The networking package provides classes for network stack modeling and wireless data transmission simulations, as well as network stack implementations and `Device` implementations that run network stacks.

2.5 gymwipe.plants package

2.5.1 Submodules

gymwipe.plants.core module

Core components for plant implementations.

class Plant
Bases: `object`

Plants are supposed to hold the state of a simulated plant and make it accessible to simulated sensors and modifyable by simulated actuators. The `Plant` class itself does not provide any features.

class OdePlant(*world=None*)
Bases: `gymwipe.plants.core.Plant`

A `Plant` implementation that interacts with an ODE world object: It offers an `updateState()` method that makes the ODE world simulate physics for the SimPy simulation time that has passed since the most recent `updateState()` call.

Parameters **world** (`Optional[World]`) – A py3ode `World` object. If not provided, a new one will be created with gravity `(0,-9.81,0)`.

updateState()
Performs an ODE time step to update the plant's state according to the current simulation time.

gymwipe.plants.sliding_pendulum module

A plant, sensor, and actuator implementation for an inverted pendulum.

class SlidingPendulum(*world=None*, *visualized=False*)
Bases: `gymwipe.plants.core.OdePlant`

Simulates a pendulum, mounted on a motorized slider.

getAngle()

Return type `float`

getAngleRate()

getWagonPos()

Return type `float`

getWagonVelocity()

Return type `float`

setMotorVelocity(*velocity*)

class AngleSensor(*name*, *frequencyBand*, *plant*, *controllerAddr*, *sampleInterval*)

Bases: *gymwipe.networking.devices.SimpleNetworkDevice*

A networked angle sensor implementation for the *SlidingPendulum* plant

class WagonActuator(*name*, *frequencyBand*, *plant*)

Bases: *gymwipe.networking.devices.SimpleNetworkDevice*

A networked actuator implementation for moving the *SlidingPendulum* plant's wagon

onReceive(*packet*)

This method is invoked whenever `receiving` is `True` and a packet has been received.

Note: After `receiving` has been set to `False` it might still be called within `RECEIVE_TIMEOUT` seconds.

Parameters packet (*Packet*) – The packet that has been received

2.5.2 Module contents

2.6 gymwipe.simtools module

Module for simulation tools

class SimulationManager

Bases: `object`

The *SimulationManager* offers methods and properties for managing and accessing a SimPy simulation.

Note: Do not create instances on your own. Reference the existing instance by *SimMan* instead.

env

The SimPy `Environment` object belonging to the current simulation

nextTimeSlot(*timeSlotLength*)

Returns a SimPy timeout event that is scheduled for the beginning of the next time slot. A time slot starts whenever *now % timeSlotLength* is `0`.

Parameters timeSlotLength (`float`) – The time slot length in seconds

Return type `Event`

now
The current simulation time step

Type int

process (*generator*)
Registers a SimPy process generator (a generator yielding SimPy events) at the SimPy environment and returns it.

Parameters process – The generator to be registered as a process

Return type Process

event ()
Creates and returns a new Event object belonging to the current environment.

runSimulation (*until*)
Runs the simulation (or continues running it) until the amount of simulated time specified by *until* has passed (with *until* being a float) or *until* is triggered (with *until* being an Event).

init ()
Creates a new Environment.

timeout (*duration*, *value=None*)
Shorthand for env.timeout(duration, value)

Return type Event

timeoutUntil (*triggerTime*, *value=None*)
Returns a SimPy EventEvent that succeeds at the simulated time specified by *triggerTime*.

Parameters

- **triggerTime** (float) – When to trigger the Event
- **value** (Optional[Any]) – The value to call succeed() with

Return type Event

triggerAfterTimeout (*event*, *timeout*, *value=None*)
Calls succeed() on the *event* after the simulated time specified in *timeout* has passed. If the event has already been triggered by then, no action is taken.

SimMan = <gymwipe.simtools.SimulationManager object>
A globally accessible *SimulationManager* instance to be used whenever the SimPy simulation is involved

class SourcePrepender (*logger*)
Bases: logging.LoggerAdapter

A LoggerAdapter that prepends the string representation of a provided object to log messages.

Examples

The following command sets up a Logger that prepends message sources:

```
logger = SourcePrepender(logging.getLogger(__name__))
```

Assuming *self* is the object that logs a message, it could prepend *str(self)* to a message like this:

```
logger.info("Something happened", sender=self)
```

If *str(self)* is myObject, this example would result in the log message “myObject: Something happened”.

Parameters **logger** (Logger) – The Logger instance to be wrapped by the SourcePrepender LoggerAdapter

process (*msg*, *kwargs*)

If a *sender* keyword argument is provided, prepends "*obj*: " to *msg*, with *obj* being the string representation of the *sender* keyword argument.

class SimTimePrepender (*logger*)

Bases: *gymwipe.simtools.SourcePrepender*

A LoggerAdapter that prepends the current simulation time (fetched by requesting `SimMan.now`) to any log message sent. Additionally, the *sender* keyword argument can be used for logging calls, which also prepends a sender to messages like explained in *SourcePrepender*.

Examples

The following command sets up a Logger that prepends simulation time:

```
logger = SimTimePrepender(logging.getLogger(__name__))
```

Parameters **logger** (Logger) – The Logger instance to be wrapped by the SimTimePrepender LoggerAdapter

process (*msg*, *kwargs*)

Prepends "[Time: *x*]" to *msg*, with *x* being the current simulation time. Additionally, if a *sender* argument is provided, str(*sender*) is also prepended to the simulation time.

ensureType (*input*, *validTypes*, *caller*)

Checks whether *input* is an instance of the type / one of the types provided as *validTypes*. If not, raises a TypeError with a message containing the string representation of *caller*.

Parameters

- **input** (Any) – The object for which to check the type
- **validTypes** (Union[type, Tuple[type]]) – The type / tuple of types to be allowed
- **caller** (Any) – The object that (on type mismatch) will be mentioned in the error message.

Raises TypeError – If the type of *input* mismatches the type(s) specified in *validClasses*

class Notifier (*name=''*, *owner=None*)

Bases: object

A class that helps implementing the observer pattern. A *Notifier* can be triggered providing a value. Both callback functions and SimPy generators can be subscribed. Every time the *Notifier* is triggered, it will run its callback methods and trigger the execution of the subscribed SimPy generators. Aditionally, SimPy generators can wait for a *Notifier* to be triggered by yielding its *event*.

Parameters

- **name** (str) – A string to identify the *Notifier* instance (e.g. among all other *Notifier* instances of the owner object)
- **owner** (Optional[Any]) – The object that provides the *Notifier* instance

subscribeCallback (*callback*, *priority=0*, *additionalArgs=None*)

Adds the passed callable to the set of callback functions. Thus, when the *Notifier* gets triggered, the callable will be invoked passing the value that the *Notifier* was triggered with.

Note: A callable can only be added once, regardless of its priority.

Parameters

- **`callback`** (`Callable`[[`Any`], `None`]) – The callable to be subscribed
- **`priority`** (`int`) – If set, the callable is guaranteed to be invoked only after every callback with a higher priority value has been executed. Callbacks added without a priority value are assumed to have priority *0*.
- **`additionalArgs`** (`Optional`[`List`[`Any`]]) – A list of arguments that are passed as further arguments when the callback function is invoked

`unsubscribeCallback`(*callback*)
Removes the passed callable from the set of callback functions. Afterwards, it is not triggered anymore by this *`Notifier`*.

Parameters **`callback`** (`Callable`[[`Any`], `None`]) – The callable to be removed

`subscribeProcess`(*process*, *blocking=True*, *queued=False*)
Makes the SimPy environment process the passed generator function when *`trigger()`* is called. The value passed to *`trigger()`* will also be passed to the generator function.

Parameters

- **`blocking`** – If set to `False`, only one instance of the generator will be processed at a time. Thus, if *`trigger()`* is called while the SimPy process started by an earlier *`trigger()`* call has not terminated, no action is taken.
- **`queued`** – Only relevant if blocking is `True`. If *queued* is set to false `False`, a *`trigger()`* call while an instance of the generator is still active will not result in an additional generator execution. If queued is set to `True` instead, the values of *`trigger()`* calls that happen while the subscribed generator is being processed will be queued and as long as the queue is not empty, a new generator instance with a queued value will be processed every time a previous instance has terminated.

`trigger`(*value=None*)
Triggers the *`Notifier`*. This runs the callbacks, makes the *`event`* succeed, and triggers the processing of subscribed SimPy generators.

`event`
A SimPy event that succeeds when the *`Notifier`* is triggered

Type `Event`

`name`
The *`Notifier`*'s name as it has been passed to the constructor

Type str

2.7 gymwipe.utility module

Domain-independent utility functions

`ownerPrefix`(*ownerObject*)
Calls `__repr__()` on the *ownerObject* (if it is not `None`) and returns the result concatenated with '.'. If the object is `None`, an empty string will be returned.

Return type `str`

`strAndRepr`(*obj*)

Returns “str (repr)” where str and repr are the result of *str(obj)* and *repr(obj)*.

Return type `str`

CHAPTER 3

Bibliography

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Bibliography

[KL07] George K Karagiannidis and Athanasios S Lioumpas. An improved approximation for the gaussian q-function. *IEEE Communications Letters*, 2007.

[Sta05] William Stallings. Data and computer communications. *Prentice Hall*, 2005.

[VH08] András Varga and Rudolf Hornig. An overview of the omnet++ simulation environment. In *Proceedings of the 1st international conference on Simulation tools and techniques for communications, networks and systems & workshops*, 60. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2008.

Python Module Index

Index