pypownet Documentation

Release 2.1.1

Marvin Lerousseau

Sep 13, 2020

Quick start

pypownet is a simulator for power (electrical) grids able to emulate a power grid (of any size or characteristics) subject to a set of temporal injections (productions and consumptions) for discretized timesteps.

The simulator is able to simulate cascading failures, where successively overflowed lines are switched off and a loadflow is computed on the subsequent grid. It comes with an Reinforcement Learning-focused environment, which implements states (observations), actions (reduced to node-splitting and line status switches) as well as a reward signal. Also, the simulator possess a runner module for easily benchmarking new controlers, and has a renderer module that allows to observe the evolving grid and the actions of a controler.

CHAPTER 1

Introduction

pypownet stands for Python Power Network, which is a simulator for power (electrical) networks.

The simulator is able to emulate a power grid (of any size or characteristics) subject to a set of temporal injections (productions and consumptions) for discretized timesteps. Loadflow computations relies on Matpower and can be run under the AC or DC models. The simulator is able to simulate cascading failures, where successively overflowed lines are switched off and a loadflow is computed on the subsequent grid.

The simulator comes with an Reinforcement Learning-focused environment, which implements states (observations), actions (reduced to node-splitting and line status switches) as well as a reward signal. Finally, a renderer is available, such that the observations of the network can be plotted in real-time (synchronized with the game time).

Fig. 1: Illustration of the renderer with the *default118/* environment; note that the renderer drastically slows the performance of pypownet: the corresponding environment takes ~100 seconds to compute 1000 timesteps (~40s for *default14/* without renderer mode).

1.1 Background

1.1.1 Motivations

Hint: This subsection was extracted from Marvin Lerousseau's master thesis on the development of pypownet (the document was written half way through the project, so some aspects are renewed, but the main ideas are the same) available here.

This project addresses technical aspects related to the transmission of electricity in extra high voltage and high voltage power networks (63kV and above), such as those managed by the company RTE (Réseau de Transport d'Electricité) the French TSO (Transmission System Operator). Numerous improvements of the efficiency of energy infrastructure are anticipated in the next decade from the deployment of smart grid technology in power distribution networks to more accurate consumptions preditive tools. As we progress in deploying renewable energy harnessing wind and solar

power to transform it to electric power, we also expect to see growth in power demand for new applications such as electric vehicles. Electricity is known to be difficult to store at an industrial level. Hence supply and demand on the power grid must be balanced at all times to the extent possible. Failure to achieve this balance may result in network breakdown and subsequent power outages of various levels of gravity. Principally, shutting down and restarting power plants (particularly nuclear power plants) is very difficult and costly since there is no easy way to switching on/off generators. Many consumers, including hospitals and people hospitalized at home as well as factories critically suffer from power outages. Using machine learning (and in particular reinforcement learning) may allow us to optimize better the operation of the grid eventually leading to reduce redundancy in transmission lines, and make better utilization of generators, and lower power prices. We developed pypownet to this end, which allows to train RL models for the task of grid conduct.

1.1.2 Power grid systems

A power grid is an network made of electric hardware and intended to transmit electricity from productions to consumptions. See next figure for a representation of two power grids with 2 productions and 2 consumptions:

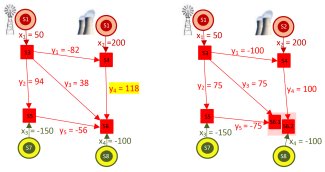


Fig. 2: Example of a power grid with 2 productions (top), 2 consumptions (bottom) and 4 substations (pink) which contain 1 or 2 nodes (red square)

Formally, the structure of a power grid is a graph G = {V, E} with V the set of nodes and E the set of edges. Edges are the power lines, also called branches or transmission lines. In practice, V is the set of substations which are concretely made of electric poles on which the connectible elements can be wired. TSOs technicians can change the pole on which an element is connected by operating switches (in pypownet, there at two poles per substation, i.e. two nodes per susbstation). They can also switch off elements such as branches such that they are temporarily removed from the grid (to repaint power lines for example).

On top of the graph structure, a power grid is submitted to Kirchoff's laws. For instance, at any node, the sum of input power is equal to the sum of output power. Given a set of injections (productions and consumptions values), a network structure and physical laws, a power grid will naturally converge to an equilibrium, also called a steady-state. In pypownet, the time is discretized into timesteps (of any given duration > 1 minute, depending on the environment). At each timestep, there will be new injections to the grid, as well as maintenance and random hazards that will force some lines to be unavailable, then apply the action given by the controler (or player), and then compute the new seatdy-state of the system, giviing a new observation to the controler.

1.1.3 Grid conduct and safety criteria

A branch is said overflowed when its flowing current is above its thermal threshold. The more current in a power line, the more it heats, which causes a dilatation phenomenon of the line. The air between the line and the ground acts as insulation and might then not been sufficient to protect nearby passengers from electric arcs. Apart from the security of nearby passengers, a melted power line needs to be replaced which can take several weeks to replace in the context of very-hight voltage power grids with an associated cost of millions of euros. On top of econominal and security cost, broken lines only reduce the capacity of the grid in term of total power, and can isolate areas which could contain hospitals where the security of thousands of people are at higher risk. Dispatchers perform actions in order to maintain their grid as safe as possible, with limited probabilities of global outage or any failure. In pypownet, controlers control two types of actions:

- switches the status of lines (e.g. disconnect a ON line or connect an OFF line)

- switches the node of which the productions, consumptions and lines are connected within substations: this changes the overall topology of the grid, by bringing "new" nodes

TSOs usually operate at nation-scale. For example, RTE operates the French grid made of more than 6500 nodes, 3000 productions and 10000 branches. Taking into account hazards, maintenance, and the injections distributions, the task of operating the grid in a safe mode is rather complex. Besides, the human factor limits the tools used for predicting the grid subsequent states. For example, temperature estimations and weather predictions could be integrated when managing the grid, but would only complicate the task for dispatchers. In this context, we are interested in building a policy $\Pi : S \rightarrow A$ (S is the State set and A the Action set), that would propose multiple curative solutions given grid states such that operators could take decisions rapidly by selecting an action among the candidate ones.

The role of pypownet is to emulate a grid as closely as possible from real life conditions, such that models can be trained using custom data and much quicker than real life.

1.2 Main features

pypownet is a power grid simulator, that emulates a power grid that is subject to pre-computed injections, planned maintenance as well as random external hazards. Here is a list of pypownet main features:

- emulates a grid of any size and electrical properties in a game discretized in timesteps of any (fixed) size
- computes and apply cascading failure process: at each timestep, overflowed lines with certain conditions are switched off, with a consequent loadflow computation to retrieve the new grid steady-state, and reiterating the process
- has an RL-focused interface, where players or controlers can play actions (node-splitting or line status switches) on the current grid, based on a partial observation of the grid (high dimension), with a customable reward signal (and game over options)
- has a renderer that enables the user to see the grid evolving in real-time, as well as the actions of the controler currently playing and further grid state details (works only for pypownet official grid cases)
- has a runner that enables to use pypownet fully by simply coding an agent (with a method act(observation))
- possess some baselines models (including treesearches) illustrating how to use the furnished environment
- can be launched with CLI with the possibility of managing certain parameters (such as renderer toggling or the agent to be played)
- functions on both DC and AC mode
- has a set of parameters that can be customized (including AC or DC mode, or hard-overflow coefficient), associated with sets of injections, planned maintenance and random hazards of the various chronics
- handles node-splitting (at the moment only max 2 nodes per substation) and lines switches off for topology management

CHAPTER 2

Installation

2.1 Using Docker

Retrieve the Docker image:

```
sudo docker pull marvinler/pypownet:2.1.1
```

This docker image contains all the necessary dependencies built on top of a Linux distribution. The sources of pypownet are available under the **pypownet** folder. See *Command-line usage* for launching the image.

2.2 Without using Docker

2.2.1 Requirements

Python >= 3.6

Octave >= 4.0

Matpower >= 6.0

2.2.2 Instructions

Step 1: Install Octave

To install Octave >= 4.0.0 on Ubuntu >= 14.04:

```
sudo add-apt-repository ppa:octave/stable
sudo apt-get update
sudo apt-get install octave
```

If Octave is already installed on your machine, ensure that its version from `octave --version` is higher than 4.0.0.

Step 2: Install Python3.6

The standard procedure:

```
sudo apt-get update
sudo apt-get install python3.6
```

If you have any trouble with this step, please refer to the official webpage of Python 3.6.6.

Step 3: Get pypownet

In a parent folder, clone the current sources:

```
mkdir parent_folder && cd parent_folder
git clone https://github.com/MarvinLer/pypownet.git
```

This should create a folder **pypownet** with the current sources.

Step 4: Get Matpower

The latest sources of matpower need to be installed for computing loadflows. This can be done using the command that should be run within the parent folder of this file:

```
git clone https://github.com/MATPOWER/matpower.git
```

Attention: In any case, you need to ensure that the path specified in `matpower_path.config` leads to the matpower folder (prefer absolute path; path relative to the configuration file are tolerated if changed before running the next setup script of pypownet).

Step 5: Run the installation script of pypownet

Finally, pypownet relies on some python packages (including e.g. numpy). Run the following command to install the current simulator (including the Python libraries dependencies):

```
python3.6 setup.py install
```

After this, this simulator is available under the name pypownet (e.g. `import pypownet`).

CHAPTER 3

Basic usage

3.1 Command-line usage

Warning: Currently, the sources of pypownet are mandatory to run the command line argument (if using the Docker image, they are already within the image).

If you installed pypownet using the Docker image, you will first need to launch bash within the image using:

```
sudo docker run -it --privileged --net=host --env="DISPLAY" --volume="$HOME/.
↪Xauthority:/root/.Xauthority:rw" marvinler/pypownet sh
```

In any case, pypownet comes with a command-line interface that allows to run simulations with a specific agent and control parameters. The basic usage will run a do-nothing policy with the default parameters:

```
python -m pypownet.main
```

This basic usage will run a do-nothing agent on the **default14/** parameters (emulates a grid with 14 substations, 5 productions, 11 consumptions and 20 lines), it takes ~100 seconds to run 1000 timesteps (old i5).

Hint: You can use `python -m pypownet.main --help` for information about the CLI tool or check the CLI usage section

3.2 As package usage

The installation process should ensure that pypownet in installed along with your corresponding python3.6. As a consequence, pypownet should be importable in your projects just like any package installed using pip: `import pypownet`. Only the modules **pypownet.environment**, **pypownet.agent** and either **pypownet.runner.Runner** or **pypownet.main** should be used, as other packages are not supposed to be used out of the induced context.

Usually, what you would do is first create an instance of **pypownet.environment.RunEnv** with an environment parameters + crate an instance of **pypownet.agent.Agent** (or any subclass), then get an instance of `pypownet.runner.Runner` with appropriate parameters (including previous `RunEnv` and `Agent`), and run its `loop` method to run the simulator. The latter functions output the total reward at the end of the experiment.

CHAPTER 4

Agents specificities

pypownet has an Environment interface, which wraps the backend game instance, and allows a RL-focused mechanism that can be used by the agents (also called models or controlers) which act as grid conduct operators.

At each timestep, upon reception of the current observation of the game, an agent is supposed to produce an action, that will be applied onto the grid, producing the next timestep observation (along with the associated reward) which will be given to the agent to produce a new action and so on. The following figure represents the typical RL loop, which greatly influences pypownet approach:

Fig. 1: Typical feedback loop in Reinforcement Learning which greatly influences the mechanism of the simulator. On the image, the link of the reward is equivalent to the function `feed_reward`, while the agent block is equivalent to the function `act` taking an observation (or state) as input, and outpouting an action.

In more details, an agent should be a python daughter class of the pypownet class **pypownet.agent.Agent** which contains two mandatory methods:

- `act` which should output a **pypownet.game.Action** given a **pypownet.environment.Observation**
- `feed_reward` which is the gateway functions that allows model to process the reward signal, along with the action taken and the consequent observation

Listing 1: agent.py

```
import pypownet.environment


class Agent(object):
    """ The template to be used to create an agent: any controler of the power grid␣
↪is expected to
    be a daughter of this class.
    """

    def __init__(self, environment):
```

(continues on next page)

(continued from previous page)

```
        """Initialize a new agent."""
        assert isinstance(environment, pypownet.environment.RunEnv)
        self.environment = environment

    def act(self, observation):
        """Produces an action given an observation of the environment. Takes as␣
↪argument an observation
        of the current state, and returns the chosen action."""
        # Sanity check: an observation is a structured object defined in the␣
↪environment file.
        assert isinstance(observation, pypownet.environment.Observation)

        # Implement your policy here.
        do_nothing_action = self.environment.action_space.get_do_nothing_action()

        # Sanity check: verify that the action returned is of expected type and␣
↪overall shape
        assert self.environment.action_space.verify_action_shape(action)
        return do_nothing_action

    def feed_reward(self, action, consequent_observation, rewards_aslist):
        pass
```

Hint: Agents without an explicit implementation of **feed_reward** do not learn at any point because they never receive the reward signal.

When an agent is specified as input of pypownet, the simulator will look for its `act` and `feed_reward` methods (or take the ones of the parent class **pypownet.agent.Agent** which is essentially a do-nothing policy), and use them for its step algorithm:

Listing 2: Step algorithm of the simulator. input: an agent A

```
observation = simulator.start_game()
for each timestep:
    action = A.act(observation)

    new_observation, reward_aslist, is_done = simulator.action_step(action)

    if is_done:
         new_observation = simulator.reset()

    A.feed_reward(action, new_observation, rewards_aslist)
    observation = new_observation
```

In short, every daughter class should have at least the following structure (any additional methods can be added; sanity checks should remain as they ensure the data types won't raise errors):

```
import pypownet.environment
import pypownet.agent


class CustomAgent(pypownet.agent.Agent):
    def __init__(self, environment):
        assert isinstance(environment, pypownet.environment.RunEnv)
```

(continues on next page)

(continued from previous page)

```
        super().__init__(environment)

    def act(self, observation):
        assert isinstance(observation, pypownet.environment.Observation)

        # Implement your policy here.
        action = self.environment.action_space.get_do_nothing_action()

        assert self.environment.action_space.verify_action_shape(action)
        return action

    def feed_reward(self, action, consequent_observation, rewards_aslist):
        pass
```

CHAPTER 5

Using Environment information

5.1 Building actions

By construction, every agent has access to the current Environment of the simulator (see **__init__** method of **pypownet.agent.Agent** displayed above). More formally, the `environment` member of the class **Agent** is an instance of **pypownet.environment.RunEnv** which is the interface to the simulator. A **RunEnv** notably contains a `step` method, which is used for computing the next observation, the resulting reward (which might depend on the action), and whether there was a game over. This method should not be explicitely used, since pypownet comes with a **pypownet.runnner.Runner** class (which is used when calling pypownet with CLI) and that automates the simulator looping.

RunEnv also contains the action space of the simulation. Formally, an action space is an instance of **pypownet.environment.ActionSpace**, which implements several functions that are used to build actions, which are instances of **pypownet.game.Action**.

Important: In pypownet, the agents are not supposed to construct actions by manipulating **pypownet.game.Action** but rather by using an **ActionSpace**, which is similar to an **Action** factory.

Given an instance *environment* of **RunEnv**, its action space can be assessed with `environment.action_space`.

5.1.1 Action understanding

Typically on natiowide power grids, dispatchers use two types of physical actions:

- switching ON or OFF some power lines
- switching the nodes on which elements, such as productions, consumptions, or power lines, are connected within their substation

These two types of physical actions constitute an **Action** in pypownet. Sometimes, dispatchers can negociate with producers to change their planned production schemes, but these operations are expensive and longer to process so they are not considered in pypownet.

Within the game, actions are managed as lists of binaries of consistent size within an environment. More precisally, this list is equivalent to the concatenation of two lists: one for the node switching operations, and one for the line status (ON or OFF) switching operations Each binary value of both lists correspond to the activation of a switch (1) or no activation of this switch (0), precisally:

- a value of 1 in the line status switches subaction indicates to activate the switch of the corresponding line status of the line: if the prior line status is 0 (i.e. the line is switched OFF, or OFFLINE), then it will be put to 1 (i.e. the line is switched ON, or ONLINE) and vice versa
- a value of 1 in the node switches subaction indicates to activate the switch of the node on which the corresponding element is *directly* connected: if the prior node on which the element is connected is 0, then this element will be connected the the node 1 (i.e. the second node of the substation of the element) and vice versa

---

Hint: The important thing to remember about actions is that they represent **switches activations**, for the lines status (ON or OFF) or the node on which productions, loads, line origins and line extremities are connected (0 or 1).

In practice, actions of class **Action** are not constructed by hands, but by using the **ActionSpace** (`environment.action_space`) There are essentially two ways of building actions:

(a) either build a binary numpy array of expected length and convert it to an action

(b) or by iteratively constructing an action by building blocks on top of a 0 action (i.e. do-nothing action)

5.1.2 Building actions from (numpy) arrays

The first way of constructing actions is to first build a binary array, such as a numpy array, and then to call the function `environment.action_space.array_to_action`. The expected size of the action (which is essentially a list), can be assessed at `environment.action_space.action_length`. Here is an example of an agent that outputs a random action where switches are randomly activated:

```
import pypownet.environment
import pypownet.agent


class CustomAgent(pypownet.agent.Agent):
    def act(observation):
        action_space = self.environment.action_space
        expected_size = action_space.action_length

        random_action_asarray = np.random.choice([0, 1], expected_size)

        # Convery np array to Action (this function also verifies the good shape and binary-type of input array)
        random_action = action_space.array_to_action(random_action_asarray)
        return random_action
```

It is possible to use some structure of an **Action** while using numpy arrays. Recall that an **Action** is the concatenation of two lists: one for the switches of the topology, and one for the switches of the lines status. Actually, the list of nodes switches is made of the concatenation of 4 sublists which are the switches of the nodes of resp. productions, loads, origins of line and extremities of line. Each of these lists, including the lines status list, should be of size the total number of corresponding elements in the grid. These lengths can be retrieved from resp. `environment.action_space.prods_switches_subaction_length`, `environment.action_space.loads_switches_subaction_length`, `environment.action_space.lines_or_switches_subaction_length`, `environment.action_space.lines_ex_switches_subaction_length`, and `environment.action_space.lines_status_subaction_length` for the line status length. For instance, a grid with 2 productions,

3 consumptions, 4 origins of line and 4 extremities of line (there are in total 4 power line in the grid so 4 lines status), has actions of size 2+3+4+4+4=17 binary values.

For illustration, here are two agents which resp. randomly switches lines status and randomly switches loads nodes:

```
import pypownet.environment
import pypownet.agent


class RandomLineStatusSwitches(pypownet.agent.Agent):
    def act(observation):
        action_space = self.environment.action_space
        expected_size = action_space.action_length

        action_asarray = np.zeros(expected_size)
        action_asarray[-action_space.lines_status_subaction_length:] = \
            np.random.choice([0, 1], action_space.lines_status_subaction_length)

        return action_space.array_to_action(action_asarray)
```

```
import pypownet.environment
import pypownet.agent


class RandomLoadsNodesSwitches(pypownet.agent.Agent):
    def act(observation):
        action_space = self.environment.action_space
        expected_size = action_space.action_length

        # Build 0-subaction where no switch is activated for all elements (incl. lines status) except loads
        prods_switches_subaction = np.zeros(action_space.prods_switches_subaction_length)
        lines_or_switches_subaction = np.zeros(action_space.lines_or_switches_subaction_length)
        lines_ex_switches_subaction = np.zeros(action_space.lines_ex_switches_subaction_length)
        lines_status_switches_subaction = np.zeros(action_space.lines_status_subaction_length)

        # Build action with random activated switches for loads
        loads_switches_subaction = np.random.choice([0, 1], action_space.loads_switches_subaction_length)

        # Build an array on the same principle as an Action; /!\ the order is important here!
        action_asarray = np.concatenate((prods_switches_subaction,
                                         loads_switches_subaction,
                                         lines_or_switches_subaction,
                                         lines_ex_switches_subaction,
                                         lines_status_switches_subaction,))

        return action_space.array_to_action(action_asarray)
```

This first way of building actions (which is essentially building arrays), is quite simple to put in place for neural networks models ans such. However, it hardly exploit the grid structure (elements are decoupled regardin their substations). To perform more dispatchers-like action, the second way of building actions using the action space as a factory is preferred since an **ActionSpace** contains various helpers to retrieve pertinent information with the point of view of

substations.

5.1.3 Building actions with the action space

The other way to construct actions is to consider an action as a container, which will store independent package of actions. The do-nothing action, which consist of an action where no switches are activated so equivalently to a list of only 0, is neutral to the environment: since no switch are activated, the whole topology of the grid stays intact, si it is as if there was no action at all. Based on this principle, we can replace parts of a do-nothing action with some meta-action, such as changing the configuration of a whole substation. For concrete usage of the action space, the the example of agents in *Example of agents*.

Here is the list the building methods of **pypownet.environment.ActionSpace**:

get_do_nothing_action return a do-nothing (neutral) action where no switch are activated.

array_to_action converts a numpy array into a proper **Action**; raises errors if the input array is not of good length, or contains non-binary values.

get_number_elements_of_substation retrieve the number of elements (productions + loads + line origins + line extremities) of a substation from its true ID.

get_switches_configuration_of_substation from a substation id, return the current switch values and type of the corresponding elements; this function returns two lists of size the number of elements of the substation from the input ID: the first one contains the binary values of the switches, while the second one returns the element-wise type of the concerned objects, which can be either `pypownet.ElementType.PRODUCTION`, `pypownet.ElementType.CONSUMPTION`, `pypownet.ElementType.ORIGIN_POWER_LINE` or `pypownet.ElementType.EXTREMITY_POWER_LINE`.

set_switches_configuration_of_substation within the input action, replace the value of the switches configuration related to input substation id with the input new configuration; this is the operation that changes the local topology of a substation with node switches.

set_lines_status_switches_of_substation similarly to the previous one, replace the lines status of the lines of the input substation id with the input new values.

set_lines_status_switch_from_id same as before except that this function changes 1 line status based on the input line id, where lines id range from 0 to the number of lines of the grid - 1.

verify_action_shape verify that the input action or array-like container is of expected shape, and contains only binary values.

5.2 Reading observations

5.2.1 Class Observation

For their `act` method, the agents receive an observation which is extracted from the current state of the grid. Those observations are of type **pypownet.environment.Observation**, which is a class mainly acting as a container for several lists, and also contains several helpers functions juste like **ActionSpace**.

Precisally, an observation is composed of 1 **datetime** object (the current simulator date) and 36 lists of fixed (but different) sizes which are:

Element type	Name	Value type	Description
	substations_ids	>=0 int	ID of the substation on which the productions (generator
production	prods_substations_ids	>=0 int	ID of the substation on which the productions (generator
	active_productions	>=0 float	Real power produced by the generators of the grid (MW
	reactive_productions	float	Reactive power produced by the generators of the grid (
	voltage_productions	>0 float	Voltage magnitude of the generators of the grid (per-unit
	productions_nodes	binary	The node on which each production is connected within
	initial_productions_nodes	binary	The initial (reference) node on which each load is conne
	planned_active_productions	>=0 float	An array-like container of the previsions of the active po
	planned_voltage_productions	>0 float	An array-like container of the previsions of the voltage c
	are_prods_cut	binary	Mask whether the productors are isolated (1) from the re
load	loads_substations_ids	>=0 int	ID of the substation on which the loads (consumers) are
	active_loads	>=0 float	Real power consumed by the demands of the grid (MW)
	reactive_loads	float	Reactive power consumed by the demands of the grid (M
	voltage_loads	>0 float	Voltage magnitude of the demands of the grid (per-unit V
	loads_nodes	binary	The node on which each load is connected within their c
	initial_loads_nodes	binary	The initial (reference) node on which each production is
	planned_active_loads	>=0 float	An array-like container of the previsions of the active po
	planned_reactive_loads	>0 float	An array-like container of the previsions of the voltage c
	are_loads_cut	binary	Mask whether the consumers are isolated (1) from the re
origin of line	lines_or_substations_ids	>=0 int	ID of the substation on which the loads (consumers) are
	active_flows_origin	>=0 float	Real power consumed by the demands of the grid (MW)
	reactive_flows_origin	float	Reactive power consumed by the demands of the grid (M
	voltage_flows_origin	>0 float	Voltage magnitude of the demands of the grid (per-unit V
	lines_or_nodes	binary	The node on which each load is connected within their c
	initial_lines_or_nodes	binary	The initial (reference) node on which each production is
extremity of line	lines_ex_substations_ids	>=0 int	ID of the substation on which the loads (consumers) are
	active_flows_extremity	>=0 float	Real power consumed by the demands of the grid (MW)
	reactive_flows_extremity	float	Reactive power consumed by the demands of the grid (M
	voltage_flows_extremity	>0 float	Voltage magnitude of the demands of the grid (per-unit V
	lines_ex_nodes	binary	The node on which each load is connected within their c
	initial_lines_ex_nodes	binary	The initial (reference) node on which each production is
line	lines_status	binary	Whether the lines are connected/ON (1) or disconnected
	ampere_flows	>=0 float	Total ampere flowing inside lines
	thermal_limits	>0 float	Each line thermal limit in Ampere over which the line ov
	timesteps_before_lines_reconnectable	>=0 int	Number of timesteps before each line can be reconnecte
	timesteps_before_planned_maintenance	>=0 int	Number of timesteps before a line will be disconnected l

All of these lists containers (also including the *datetime* field) can be retrieved from an **Observation** `observation` with `observation.name` where name is one of the previous names (+ `observation.datetime`) e.g.:

```
import pypownet.environment
import pypownet.agent


class CustomAgent(pypownet.agent.Agent):
    """ At each timestep, this agent selects all the even substations id, and
    switch the node of their production if any.
    """
    def act(observation):
        action_space = self.environment.action_space
        # Build 0 action
        action = action_space.get_do_nothing_action()

```

(continues on next page)

(continued from previous page)

```
        # Retrieve the substations ids
        substations_ids = observation.substations_ids

        # Loops on all substations
        for substation_id in substations_ids:
            # Selects even substations ids
            if substation_id % 2 == 0:
                # Build new configuration: will activate switches of productions only
                _, elements_type = action_space.get_switches_configuration_of_
↪substation(action, substation_id)
                new_configuration = np.zeros(len(elements_type))

                # Activate all switches of productions
                new_configuration[elements_type == pypownet.environment.ElementType.
↪PRODUCTION] = 1

                # Set new configuration for this substation within the buffer action
                action_space.set_switches_configuration_of_substation(action,␣
↪substation_id, new_configuration)

        return action
```

Hint: At any moment, you can retrieve the name of the lists and their associated description available in an **Observation** with the global python dictionary `pypownet.environment.OBSERVATION_MEANING`: `print(pypownet.environment.OBSERVATION_MEANING)`

The **pypownet.environment.Observation** class has several dispatcher-like functions as well as helper for manipulating them. Here is the list of functions of **Observation**:

as_dict returns the observation as a dictionary, with the field name as keys (str) and the lists as values (np arrays) + the `datetime` value as a **datetime** Python object (note that the dictionary has 37 elements for all environments).

as_array returns the observation as a numpy array: each list is concatenated (order is consistent for all environments) and returned as 1 numpy array of 1 dimension (essentially a list); note that `datetime` is not included in this array, and can be accesses with `observation.datetime` (as any other field member).

get_lines_capacity_usage helper function that computes the nominal lines capacity usage of the self observation: it performs elementwise division of the ampere flows in lines by their nominal thermal limit. A line capacity usage greater than 1 indicates an overflowed line.

get_nodes_of_substation returns the list of value of the nodes on which each element of the substation with input id. This function also computes the type of element associated to each node value of the returned nodes-value list.

__keys__ returns a list of all the field name of the class as strings (list of size 37).

__str__ returns a prettify version of the observation (this is what can be seen when using `-v -vv` CLI arguments) as condenses matrices. Note: the substations ids are not included in this string.

as_ac_minimalist see after

as_minimalist see after

The `observation.get_nodes_of_substation` is essentially similar to the `action_space.get_switches_configuration_of_substation`, because both their arguments returns (consitently)

the **pypownet.environment.ElementType** of the elements of the substation with input substation id, and they both return the associated values of the elements. The difference is that the one in **Observation** returns the true node on which elements are connected within the self observation, whereas the one in **ActionSpace** returns the values of the *switches* in the input action.

5.2.2 Reducing the observation space

For an environment associated to the common grid case known as case14 (available in environment **default14/**), they are 438 values in the array return by any `observation.as_array()`. Some of these values are integer, other are float, and some are binaries. In any case, by construction some fields of an **Observation** are fixed throughout a given environment (such as **default14/**), including for instance `observation.substations_ids`. Some model approach including neural network have no precise way to naturally leverage some structure information, so reducing the observation by the fixed lists can help optimizing alorithms by shrinking *uninformative* values. No matter the final usage, there are two version of **Observation** which contains a subset of its fields:

- **pypownet.environment.MinimalistACObservation** contains 26 lists + `datetime` and represents an **Observation** without the list fixed members (mainly the IDs list fields)
- **pypownet.environment.MinimalistObservation** contains 14 lists + `datetime` and represent a **MinimalistACObservation** without the list members which are fixed in DC mode but not in AC mode (including voltages since one of the hypothesis of DC is all voltage to 1, also reactive power for similar reasons etc)

You can convert an **Observation** into a **MinimalistACObservation** with `ac_minimalist_observation = observation.as_ac_minimalist()` and a **MinimalistACObservation** into a **MinimalistObservation** with `minimalist_observation = ac_minimalist_observation.as_minimalist()`. Transitively, an **Observation** can be converted to a **MinimalistObservation** with `minimalist_observation = observation.as_minimalist()`.

Both implement the following methods, which acts the same as the one of **Observation** with the same name: `as_array`, `__keys__` and `as_dict`. Note that the function `get_nodes_of_substation` is not implemented in neither **MinimalistACObservation** or **MinimalistObservation** because those classes do not have the ids of the elements, so they are no way to find the various elements of a given substation for both these classes.

CHAPTER 6

Example of agents

This page contains a collection of *simple* (non-learning) policies, with some of them leveraging the information contained in observations and the environment. These agents are all available using the command line agent parameters.

6.1 Do-nothing agent

This agent returns a do-nothing action at each timestep, i.e. an action that has no consequence on the grid. The implementation of this agent is actually the default Agent class of pypownet:

Listing 1: agent.py

```
import pypownet.environment
import pypownet.agent


class CustomAgent(pypownet.agent.Agent):
    """ The template to be used to create an agent: any controler of the power grid is expected to be a daughter of this
    class.
    """
    def __init__(self, environment):
        assert isinstance(environment, pypownet.environment.RunEnv)
        super().__init__(environment)

    def act(self, observation):
        """Produces an action given an observation of the environment. Takes as argument an observation of the current state, and returns the chosen action."""
        action = self.environment.action_space.get_do_nothing_action()

        # Alternative equivalent code:
        # action_space = self.environment.action_space  # Retrieve action builder, an instance of pypownet.environment.ActionSpace
        # action_asarray = np.zeros(action_space.action_length)
```

(continues on next page)

(continued from previous page)

```
        # action = action_space.array_to_action(action_asarray)

        return action
```

6.2 Random switch agent

This agent will simply return an action where only one value is 1, which is equivalent of saying that only 1 switch is activated (whether it is a node-splitting switch or a line status switch).

This method performs poorly, but is here to illustrate the gateway between (numpy) arrays and actions.

```
import pypownet.environment
import pypownet.agent
import numpy as np


class RandomSwitch(pypownet.agent.Agent):
    def __init__(self, environment):
        super().__init__(environment)

    def act(self, observation):
        # Sanity check: an observation is a structured object defined in the environment file.
        assert isinstance(observation, pypownet.environment.Observation)
        action_space = self.environment.action_space  # Retrieve action builder, an instance of pypownet.environment.ActionSpace

        # Build array of action length with all 0 except one random 1
        action_asarray = np.zeros(action_space.action_length)
        action_asarray(np.random.randint(action_length)) = 1  # Activate one random switch

        # Convert previous array into an instance of pypownet.game.Action
        action = action_space.array_to_action(action_asarray)
        return action
```

6.3 Random 1-line switch agent

The actions of this agent are completely restricted to line status switches with at most 1 switch. At each timestep, the agent will pick a random line id, and switch its status: that is, its action vector is all 0 except one 1 value within the line status subaction.

At the beginning (so after each reset), all of the power lines of the grid should be switched ON, which implies that this model mainly switches OFF power lines, which tend to create isolated loads or productions or even global outage. As a consequence, this model does not perform well (by switching line status of ON lines, the model only lower the capacity of the grid). It illustrates how to leverage the inner structure of an action, which is roughly made of two subarrays, one for node-splitting related switches, and one for lines status related switches.

Listing 2: agent.py

```
import pypownet.environment
import pypownet.agent
```

(continues on next page)

(continued from previous page)

```
import numpy as np


class RandomLineSwitch(pypownet.agent.Agent):
    """ An example of a baseline controler that randomly switches the status of one random power line per timestep
    (if the random line is previously online, switch it off, otherwise switch it on).
    """
    def __init__(self, environment):
        super().__init__(environment)

    def act(self, observation):
        # Sanity check: an observation is a structured object defined in the environment file.
        assert isinstance(observation, pypownet.environment.Observation)
        action_space = self.environment.action_space

        # Create template of action with no switch activated (do-nothing action)
        action = action_space.get_do_nothing_action()
        # Select random line id
        random_line_id = np.random.randint(action_space.lines_status_subaction_length)

        # Given the template 0 action, and the line id, set new line status SWITCH to 1 (i.e. activate line status switch)
        action_space.set_lines_status_switch_from_id(action=action,
                                                     line_id=random_line_id,
                                                     new_switch_value=1)

        return action
```

6.4 Random 1-substation node-splitting switch agent

To the image of the previous agent, the second part of an action is related to node-splitting switches: if a switch is activated, then the corresponding element will move from its current node within its substation to the other one (there are 2 nodes per substation). This agent leverages some helpers in the action space that allows the model to fully exploit the topological structure of the grid.

Indeed, the model first retrieve the true IDs of the substations of the grid, which are contained within any observation given by the Environment in their sublist **substations_ids**. It then picks one random substation ID, and uses the function **get_number_elements_of_substation** of action space to retrieve the number of elements in this substation (this function is a helper, which returns the total number of productions, loads, lines origins and lines extremities that are part of the associated substation). The agent then construct a random binary configuration of size the previous number, and insert this new configuration into a 0 action using the function **set_switches_configuration_of_substation** of the action space, that *build* the new configuration on top of the action (i.e. it does not modify the other values of the input action).

This agent mainly illustrates how to leverage some of the grid system topological structure, with the use of the action space to construct meaningful actions.

Listing 3: agent.py

```
import pypownet.environment
import pypownet.agent
```

(continues on next page)

(continued from previous page)

```
import numpy as np


class RandomNodeSplitting(Agent):
    """ Implements a "random node-splitting" agent: at each timestep, this controler will select a random substation
    (id), then select a random switch configuration such that switched elements of the selected substations change the
    node within the substation on which they are directly wired.
    """
    def __init__(self, environment):
        super().__init__(environment)

    def act(self, observation):
        # Sanity check: an observation is a structured object defined in the environment file.
        assert isinstance(observation, pypownet.environment.Observation)
        action_space = self.environment.action_space

        # Create template of action with no switch activated (do-nothing action)
        action = action_space.get_do_nothing_action()

        # Select a random substation ID on which to perform node-splitting, and retrieve its total number of elements (i.e. size of its subaction list)
        substations_ids = observation.substations_ids  # Retrieve the true substations ID in the observation (they are fixed per environment)
        target_substation_id = np.random.choice(substations_ids)

        # Computes the number of elements of substation (= size of values-related subaction) and choses a new random
        # switch configuration (binary array)
        expected_target_configuration_size = action_space.get_number_elements_of_substation(target_substation_id)
        target_configuration = np.random.choice([0, 1], size=(expected_target_configuration_size,))

        # Incorporate this new subaction into the action (which is initially a 0 action)
        action_space.set_switches_configuration_of_substation(action=action,
                                                              substation_id=target_substation_id,
                                                              new_configuration=target_configuration)

    return action
```

6.5 Exhaustive 1-line switch search agent

The actions of this agent are completely restricted to line status switches with at most 1 switch. At each timestep, the agent will *simulate* every 1-line switch action, independently from one another, as well as the do-nothing action. The *simulate* method will return one reward per action simulated. After this, the model will return the action that maximizes the expected reward for the next timestep restricted to 1-line switches.

Note: The *simulate* method is an approximation of the *step* method: the mode is automatically DC (should be AC in

step), and the hazards are not computed for future timesteps becaseu they should not be predicted by the agent.

This agent should perform relatively well compared to the other ones above. Even if the *simulate* is approximative, the model can easily discard actions that would lead to global outage (if outage in DC, then probably also in AC). Also, this agent is way longer than the other ones, because it simulates multiple actions per timestep: one per line (+ 1 do-nothing) and the simulate method takes some time on its own.

Listing 4: agent.py

```
import pypownet.environment
import pypownet.agent
import numpy as np


class SearchLineServiceStatus(Agent):
""" Exhaustive tree search of depth 1 limited to no action + 1 line switch activation
"""
    def __init__(self, environment):
        super().__init__(environment)

    def act(self, observation):
        # Sanity check: an observation is a structured object defined in the␣
↪environment file.
        assert isinstance(observation, pypownet.environment.Observation)
        action_space = self.environment.action_space

        number_of_lines = action_space.lines_status_subaction_length
        # Simulate the line status switch of every line, independently, and save␣
↪rewards for each simulation (also store
        # the actions for best-picking strat)
        simulated_rewards = []
        simulated_actions = []
        for l in range(number_of_lines):
            print('    Simulating switch activation line %d' % l, end='')

            # Construct the action where only line status of line l is switched
            action = action_space.get_do_nothing_action()
            action_space.set_lines_status_switch_from_id(action=action, line_id=l,␣
↪new_switch_value=1)

            # Call the simulate method with the action to be simulated
            simulated_reward = self.environment.simulate(action=action)

            # Store ROI values
            simulated_rewards.append(simulated_reward)
            simulated_actions.append(action)
            print('; expected reward %.5f' % simulated_reward)

        # Also simulate the do nothing action
        print('    Simulating switch activation line %d' % l, end='')
        donothing_action = self.environment.action_space.get_do_nothing_action()
        donothing_simulated_reward = self.environment.simulate(action=donothing_
↪action)
        simulated_rewards.append(donothing_simulated_reward)
        simulated_actions.append(donothing_action)

        # Seek for the action that maximizes the reward
```

(continues on next page)

(continued from previous page)

```
45          best_simulated_reward = np.max(simulated_rewards)
46          best_action = simulated_actions[simulated_rewards.index(best_simulated_
    ↪reward)]
47
48          print('  Best simulated action: disconnect line %d; expected reward: %.5f' %␣
    ↪(
49              simulated_rewards.index(best_simulated_reward), best_simulated_reward))
50
51          return best_action
```

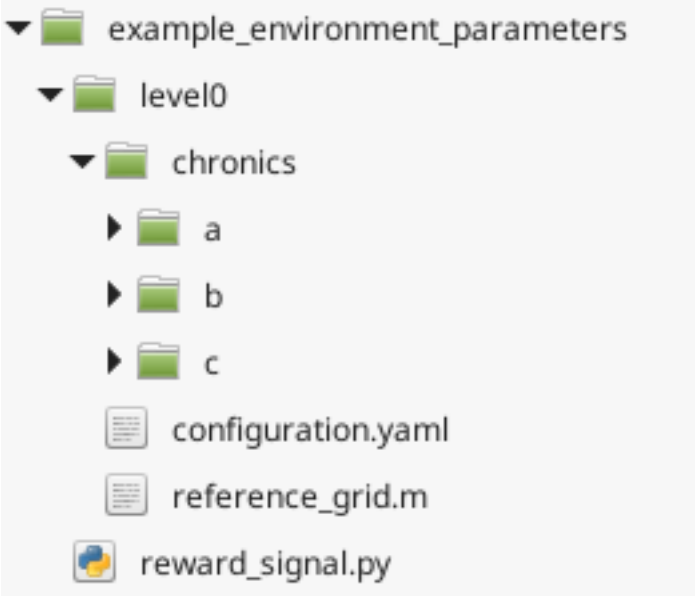
CHAPTER 7

Parameters management

Some mechanisms and inputs of the game are parameterizable to ensure that the users of the package have some control with respect to the simulations to be run. New simulations environments can be created with any virtual grid (provinding an initial grid case) and an associated set of grid timestep entries to be injected.

Hint: The simulator has primarly been designed for RL research; consequently, the overall parameters environment organization is influenced for RL integration.

Parameters are organized into a generic folder structure:



An environment parameters is made of the following elements:

- one reference grid (defines static parameters of the simulated power grid)

- sets of chronics (defines the temporal data driving the inputs of the grid environment)
- a configuration file (contains parameters for some mechanisms of the simulator)
- (*optional*) a reward signal python file (implements the formula to compute a reward based on a observation and an action)

Before rewieving the details about each of these elements, pypownet comes with a helper that creates a template parameters environment. The script can be launched using:

```
python -m parameters.build_new_parameters_environment
```

The terminal will ask you several questions relative to the latter elements, and then build the overal architecture of the environment. After the execution of this script, you will need to:

- (*mandatory*) fill the data for the chronics
- (*optional*) modify the default values of the auto-generated configuration file
- (*optional*) modify the reward signal of the file reward_signal.py.

CHAPTER 8

Default environments

pypownet comes with 4 sets of paremeters of environment which are named according to their top folder:

- default14/: https://github.com/MarvinLer/pypownet/tree/master/parameters/default14
- default30/: https://github.com/MarvinLer/pypownet/tree/master/parameters/default30
- default118/: https://github.com/MarvinLer/pypownet/tree/master/parameters/default118
- custom14/: https://github.com/MarvinLer/pypownet/tree/master/parameters/custom14

Both environments suse the official IEEE case format associated with the int in their name (e.g. default14/ uses the official case14.m reference grid). Each has one **level0/** level folder, and has 12 sets of chronics (one per month, starting by january). All of the chronics are discretized into 1 hour timestep: this can be seen within the **datetimes.csv** of each chronic folder. The values of the parameters of the **configuration.yaml** file are printing at each start of pypownet: they slightly differ among these environments, notably the maximum number of prods and loads isolated.

The reward signal of the custom14/ is rather simple: if the timestep lead to a game over, then return -1, otherwise return 1. This reward signal is not very representative of the factors to optimize for real conditions grid conduct, but illustrates that the reward signal can be designed very simply.

For the *defaultXX* environments above, the reward signal has the same mechanism (in fact, they are the same except for hyperparameters which scale with the size of the grid). More precisally, their reward signal is made of 5 subrewards (the output is then a list of 5 values; the input is still the new observation form the application of the action, as well as the action and a flag indicating game overs, see *Reward signal file*):

subreward proportional to the number of (topologically) isolated productions Negatively proportional to the number of fully isolated productions. In real life settings, production plants are not controlled by dispatchers, so it is best to take actions such that no production is isolated, as it would perturbate the scheme of producers.

subreward proportional to the number of (topologically) isolated loads Negatively proportional to the number of fully isolated consumtpions (i.e. demands or loads). In real life settings, no consumer should be deprive of electricity which notably happens when the load are isolated (since no line reaches them). Typically it is more problematic to have an isolated load compared to an isolated production: if the geographic zone without electricity has hospitals, the life of many patients

are at risk (even if some have personal generators). Consequently, the amplitude of the multiplicative factor to the sum of isolated loads is chosen twice the one for isolated productions (2 isolated productions *equivalent* to 1 isolated load in terms of reward).

subreward of the cost of the action Recall that all the values of actions are switches: this is roughly what happens in real life, where switches are activated to isolate lines and potentially wire them on the other node (or nodes). In practice, a team of line operators need to go the the particular switch to activate, which takes some time and thus cost money to the company. Consequently, each switch will account for some negative reward, which scale to the number of activated switches. For these reward signals, the cost of a node switch activation halves the one of a line status switch activation, because we would like agents to perform more node-splitting actions (the *a priori* here is that generally, switching the line status of a line will put it OFFLINE, which will reduce the overall capacity of the grid making it more sensible to failures or global outages).

subreward of the distance to the reference grid The *distance* to the reference grid is precisally the number of nodes of every element in the current grid that differs from the nodes of every element of the initiale grid (note that this in independent from the lines status). As such, the distance to the reference grid can be viewed as the minimal number of node switches to activate to convert the current grid topology to the reference one. We introduce this distance scale by a negative factor to *force* agents to keep the grid topology around the reference one. This is motivated by the fact that human dispatchers are *used to* work with a grid close to a reference grid topology (i.e. the *normal* topology); the dispatchers know the macro patterns of the reference grid, so ensure that their action renders the grid topology close to their confidence zone. Since the models would eventually be used to *assist* dispatchers in real life, they should ensure that the grid topology does not drift. An important consequence of the design of this reward to be kept in mind is that the reward will sum at each step, such that there might be some delay between the intentions of models (e.g. return to reference grid) and the improvement of the associated reward.

subreward proportional to the sum of squared lines capacity usage Finally, we use the lines capacity usage to make the models avoid overflows. One approach could have been for this subreward to be equal to the number of overflows, but the counting function is not smooth (not even continuous), which is usually not wanted for learning models. Instead, we use the square of the nominal lines capacity usage: for each line, its ampere flow is divided by its thermal limit, and the result is squared. Since an overflowed line is one with a capacity usage >= 1, the lines currently overflows will be amplified (>=1 squared), while the non-overflowed lines will be deamplified (<1 squared). Another positive point of this subreward is that it discriminizes well two grid situations with the same number of overflows, since it takes into account the capacity usage information of all lines. For instance, one grid with 1 overflow and all other lines at 99% capacity usage should have a worse reward than one with 1 overflow and 10% usage for other lines.

The values of the scaling factors for each of the environment can be seen in their `reward_signal.CustomRewardSignal.__init__`.

Besides, the **CustomRewardSignal** of the *defaultXX* default environments manage the flag of game over returned by th step function `pypownet.environment.RunEnv.step(action)`; see *Reward signal file* for more information on this flag. For all the kinds of pypownet's exceptions contained within the flag, the model will output a specific (constant) reward whose values can be viewed in the `__init__` file. The `compute_reward` function of their custom reward will output rewards with the values at specific places within the output list: for instance, if too many loads are cut (see *Configuration file*), then the first value of the ouptut list (relative to loads cut subreward by design) will be of value the generic value for game over caused by too many cut loads, and 0 for the other values; the sum of this list equals the effective nominal value for too many loads isolated.

Overall, the design of the *defaultXX* default environments reward signal involves security (grid and individuals), economical and convenience aspects, which should approximate the ultimate score function for the goal of creating controlers supporting dispatchers for the everyday task of conducting power grid of nationwide scale.

See the next section *Creating new environment parameters* for creating new environments.

CHAPTER 9

Creating new environment parameters

9.1 Reference grid

A reference grid is a case file in matlab format defining a grid (or more precisally, a *photo* of a grid, with instant injections). This file will be read by the simulator to load the electrical parameters of the grid, including for instance reactances and susceptances of lines, or the substations of which the grid productions are wired.

Currently, the simulator expects a IEEE-format case file for the reference_grid. .. Hint:: Such files can be found on the Matpower official repository. You can also find some details about the value of the matrices and columns of the IEEE format here.

The simulator cannot work with such a grid without some modifications, including the addition of artificial buses that will emulate the sister nodes of the original substations of the grid (two sister nodes per substation), and renaming and sorting the buses original ids. The `build_new_parameters_environment` script already performs this operation from the prompted filepath of the reference grid.

In a given environment, you might want to modify only the reference grid (for example, specifiying a different topological configuration) without modifying the other resources (reward_signal.py, chronics etc). In that case, you can run the following script, which will produce a valid reference grid for pypownet:

```
python -m parameters.make_reference_grid CASE_FILE_FILEPATH
```

where CASE_FILE_FILEPATH is the path to a grid case file (e.g. case30.m).

9.2 Chronics

In pratice, given injections and a grid topology, the flows within the grid will converge to a steady-steate, where electricity is carried from producers (e.g. nuclear central) toward consumers (e.g. a city). In real condition, the amount of demand cannot be controled: in other words, the injections relative to the loads are external to the operation of grid conduct. Besides, for several countries, the company responsible for natiowide grid conduct is different than the one responsible for the nationwide electricity production: in other words, the values of the productions are not in control of tehe grid conduct operators (they are by some other company, which also ensures that there is enough

production to satisfy all demand). Those two macro aspects underlines that injections are effectively an input of the grid system in the context of grid conduct of natiowide scale.

For reproducibility purposes, productions and loads injections are thus an entry of the system (and not generated on the fly by the software), which can be controlled by a meta-user creating new chronics sets. An advantage to this approach is that the meta-user can control the timestep of the simulation: chronics entirely define the behavior of the flows within a grid. If the values of injections of a chronic have been generated with a timestep of 2 minutes, then the software will naturally be discretized into 2 minutes timesteps.

Chronics define the precise values of the entries of the Environment in which the grid will be subjected to through time. A *game level* folder contains one **chronics** folder, which contains one or several folders, which are the chronic folders (in the previous image, those chronic folders are named **a**, **b**, and **c**). More precisally, a chronic folder is made of 13 CSV files containing the temporal data for all the entries of the simulated grid system which are grouped into categories:

(i) grid injections (productions and consumptions temporal nominal values)

(ii) grid previsions of injections which are given to the agents

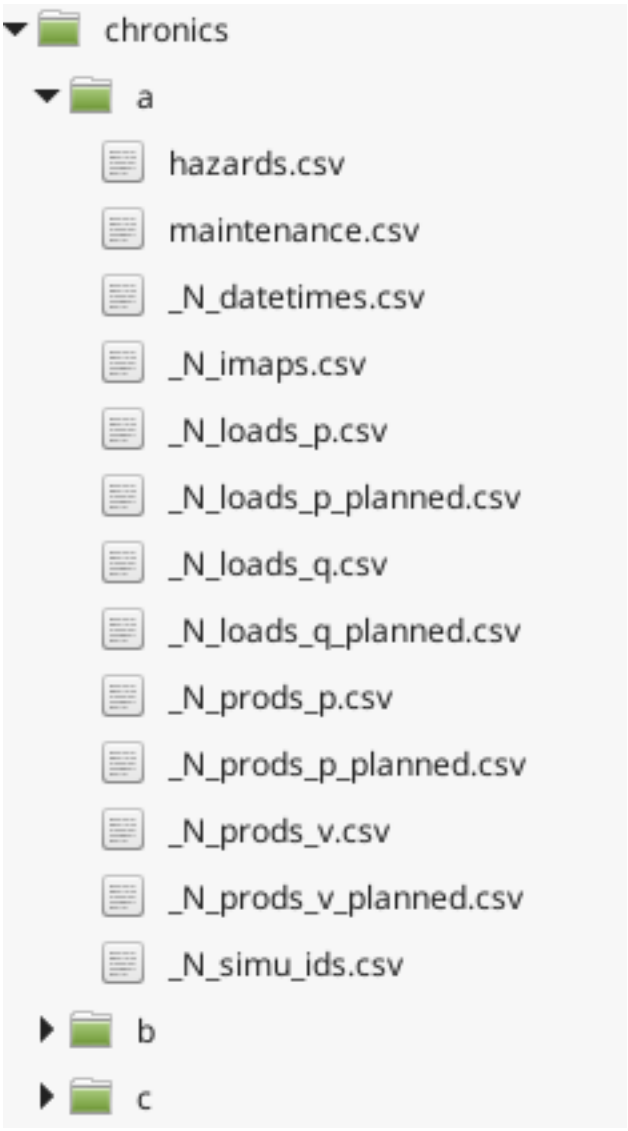
(iii) maintenance planned operations and grid external line breaking events (e.g. thunder breaking a line)

(iv) simulation datetime and absolute IDs

(v) power lines nominal thermal limit for the whole chronic

Important: The delimiter in CSV files is always ';'

For visual purposes, here is a list of the files names in a chronic:



Important: The software will seek files with the exact filenames indicating in the above figure; your chronics should eventually contain 13 CSV files with the same name as listed above.

9.2.1 1. Grid injections

The grid injections (also called *realized injections*, since the values will effectively be an unmodified input of the grid) refer to four values:

- the active power (P) of productions
- the voltage magnitude (V) of productions
- the active power (P) of consumptions
- the reactive power (Q) of consumptions

Hint: In short, injections are the P and V values of productions, and P and Q values of loads, hence the respective

names PV buses and PQ buses

The respective names of the associated chronic files are:

- _N_prods_p.csv
- _N_prods_v.csv
- _N_loads_p.csv
- _N_loads_q.csv

Each of these CSV files should have a header (which is not used in practice but mandatory) line of the desired number of file columns, followed by lines of ';'-separated values. Each line will correspond to one timestep, such that consecutive lines represent the injections of consecutive timesteps. The columns define the nominal values for each elements. For instance, if the grid is made of 5 productions and 8 loads, then both **_N_prods_p.csv** and **_N_prods_v.csv** should be made of 5 columns (so 4 ';' per line), and both **_N_loads_p.csv** and **_N_loads_q.csv** should be made of 8 columns.

In practice, all of the active power values of productions are non-negative, because productions do produce active power. Sometimes, productions undergo some maintenance process (e.g. cleaning or repairing). This aspect can be controlled within the voltage magnitudes of productions (file **_N_prods_v.csv**), by setting the associated active production value to 0 (a production producing 0 effectively does not produce any electricity), or by setting the nominal value of the production to <= 0. Usually, productions voltage magnitudes are close to 1 (ranging from 0.94 to 1.06) in per-unit (understand: in the chronic file of production voltages). Any excessive value will almost automatically lead to a game over situation caused by a non-converging loadflow.

For the loads injections, the active power (**_N_loads_p.csv**) need to be non-negative (they represent the amount of *demand* of active power). The reactive power injections of the loads (**_N_loads_q.csv**) have no restrictions, but they usually are of lower magnitudes than the active values overall.

At initialization, the software will read the 4 realized files of the chronic. The first header row is discarded for each file, then the content is split into *n* lines, where *n* is the number of timesteps. At each timestep, the software will read the same line number in each of the 4 files, and insert the values into the grid. That is, the productions P and V values are replaces by the ones in the file, same for the loads P and Q values.

Note: If there are not enough active power production to satisfy all the active power demand, the slack bus will augment its output consequently, thus producing border effects on its adjacent lines. A good reflex is to ensure that the produced chronics has enough active power production to satisfy the active power demand at each timestep.

For illustration, suppose a grid is made of 2 productions and 2 consumptions, with the following realized injections which correspond to 3 timesteps (because there are 3 lines of data):

Listing 1: _N_prods_p.csv

```
prod0;prod1
10;5
11;6
12;6.4
```

Listing 2: _N_prods_v.csv

```
prod0;prod1
1;1
1;1
1;1
```

Listing 3: _N_loads_p.csv

```
1 load0;load1
2 7;8
3 9;8.4
4 11;7
```

Listing 4: _N_loads_q.csv

```
1 load0;load1
2 -2;3
3 -2;4
4 0;-1
```

For the first timestep, the software will read the highlighted line of each files (line 2 here, because this is the first timestep) and change the corresponding P, Q, V values of productions and loads.

9.2.2 2. Grid previsions of injections

Throughout the year, nationwide grid operators have constructed tools to estimate the future demands at various scales. This can be done because the consumptions pattern are very cyclical at many scales: day-to-day, week-to-week, year-to-year etc. For instance in France, on weekdays there is a peak of consumption at 7PM (probably when people get home and start cooking), while demand is relatively low during the night. Also, there is less demand during weekends, since a lot of companies work on weekdays (industries and companies are major electricity consumers). In that context, the simulator can give to the agents some predictions about the next timesteps injections (next loads PQ values come from demand estimation, and next prods PV values come from the schedules plans of producers). At each timestep, the agent will have access to both the current timestep injections, and the previsions (which are pre-simulation computed) for the next timestep.

The value of the previsions of injections (also called *planned injections*) are nominal for each production and each consumption (i.e. there are previsions for each injection gate). Consequently, the overall structure of the planned injections files are the same than the grid injections files. At each timestep, the software will read the next line for all the 4 realized injections file, as well as the same line for all 4 planned injections files, which should be named similarly to the realized files:

- _N_prods_p_planned.csv
- _N_prods_v_planned.csv
- _N_loads_p_planned.csv
- _N_loads_q_planned.csv

For illustration, given the following pair of realized/planned active power of productions, for the second timestep, the software will read the 3rd line in both files, replace the current productions P output by the read values, and carry the previsions of P values in an Observation:

Listing 5: _N_prods_p.csv

```
1 prod0;prod1
2 10;5
3 11;6
4 12;6.4
```

Listing 6: _N_prods_p_planned.csv

```
prod0;prod1
10.9;5.8
12.9;6.3
```

In this example, the predictions, given at the first timestep, of the next timestep active power of productions are 10.9MW and 5.8MW for resp. the first production and the second production (seen on line 2 of **_N_prods_p_planned.csv**). In reality, at the next (second) timestep, the active power of productions inserted into the grid system are resp. 11MW and 5MW (seen on line 3 of **_N_prods_p.csv**).

9.2.3 3. Maintenance and external hazards

In real conditions, the power lines need to be maintained to ensure they are secure and work as intended. Such operations, called maintenance, involve switching power lines off for several hours, which make them unusable to ensure the safe functioning of the grid. The cause of maintenance are diverse (e.g. line repainting), but they are all known in advance (because they are planned by the grid manager). For the same reproducibility purposes as before, the maintenance are pre-computed prior to the simulation.

The file **maintenance.csv** provide all the maintenance that will happen during the chronic. Similarly to the previous files, the maintenance file has a header (not effectively use), followed by ';'-separated data e.g.:

Listing 7: maintenance.csv

```
lines0;line1;line2;line3
1;0;0;0
0;0;0;0
0;2;0;3
0;0;0;0
```

The number of column of **maintenance.csv** should be equal to the number of power lines in the grid (= the number of lines in the 'branch' matrix of the reference grid). Its number of lines should be the same as the files before, i.e. the number of timesteps of the chronic.

For a given timestep and a given power line (i.e. resp. a given line and a given column), a value d equal to 0 indicates that there are no maintenance starting at the corresponding timestep. A value d>0 indicates that a maintenance starts at this timestep, and that the power line will be unavailable (to be switched ON) for d timesteps starting from the current timestep.

Regarding maintenance, since in real life condition they are typically known, an Observation will also contain the previsions of the maintenance: given an *horizon* parameter (see later), the vecteur will contain one integer value for each power line, with a 0 value indicated no planned maintenance within the next *horizon* timestep, and a non-0 value indicating the number of timesteps before the next seen maintenance.

On top of maintenance operations, power grids are naturally subjected to external events that break lines from time to time. Such events could be related to nature (thunder hitting a power line, tree falling on some power line, etc), or could come from hardware malfunctioning. Such hazards are an entry of the system, and should be within the **hazards.csv** file which works exactly like the maintenance file, except that hazards are unpreditable in real life so no information is given to agents regarding forthcoming hazards.

9.2.4 4. Datetimes and IDs

The datetime file, **_N_datetimes.csv** contains the date associated with each timestep. As such, there is one date per line. The date should have the following format: 'yyyy-mmm-dd;h:mm' with 'yyyy' the 4 digits of the year, 'mmm'

the 3 first letters in lowercase of the month, 'dd' the 1 or 2 digits of the day in the month, 'h' for the 1 or 2 digits of hour (from 0 to 23) and 'mm' for the 2 digits of minutes. Example of datetimes file:

Listing 8: _N_datetimes.csv

```
date;time
2018-jan-31;8:00
2018-jan-31;9:00
2018-jan-31;10:00
2018-jan-31;11:00
```

The datetimes entirely controls the timestep used for the simulation (this is due because the game mechanism is independent of time, so essentially the chronics dictatet the speed of temporal dimension). In the latter example, the duration between two timesteps is 1 hour, so an agent can only perform one action per hour. Because of regex limitations, the system cannot be discretized into seconds timesteps; you can create an issue on the official repository if you need such a feature.

The file **_N_simu_ids.csv** allows to bring consistency with the indexing of timesteps. This simple csv file has one column, one header line and one int or float value per timestep e.g.:

Listing 9: _N_simu_ids.csv

```
id
0
1
2
```

With both examples, the timestep of id 2 happens at precisely 31st January of 2018 at 11AM.

9.2.5 5. Thermal limits

Finally, the last file of a chronic is the file **_N_imaps.csv** containing the nominal thermal limits of the power line: one thermal limit per line. The file consists in two lines: one is the header, not used (but should respect the correct number of columns), the other contain a list of ';'-separated float or int, indicating the thermal limits of each line e.g.:

Listing 10: _N_imaps.csv

```
line0;line1;line2;line3
30;90;100;50
```

Note: There is one thermal limits per chronic, and not per game level, because chronics could be splitted by month, and thermal limits are technically lower during summer (higher heat), which could be emulated with lower thermal limits for the summer chronics.

9.3 Configuration file

The configuration file contains parameters that control the inner game mechanism in several ways. More precisally, the configuration file should be named **configuration.yaml** and should be placed at the top level of the considered level folder. As its name indicates, its format should be YAML, which is preferred here over JSON because of its possibility of comments and efficiency.

Hint: The template-building script **build_new_parameters_environment.py** automatically constructs such a file, with all the mandatory parameters, with default values.

Here is the list of (mandatory) parameters:

loadflow_backend backend used by the simulator to compute loadflows; can be "pypower" or "matpower"

loadflow_mode model of loadflow used by the backend to compute loadflow; can be "AC" (alternative current) or "DC" (direct current)

max_seconds_per_timestep *not supported yet*; maximum number of seconds allowed for the agent to produce an action at each timestep, before timeout

hard_overflow_coefficient percentage of thermal limit above which its current ampere value will make a line in hard-overflow (hard-overflowed lines break instantly)

n_timesteps_hard_overflow_is_broken duration in timesteps a hard-overflowed line is broken: the line needs repairs and cannot be switched ON for this number of timesteps

n_timesteps_consecutive_soft_overflow_breaks number of consecutive timesteps at the end of which an overflowed (but not hard-overflowed) line is breaks (heat build-up)

n_timesteps_soft_overflow_is_broken duration in timesteps a soft-overflowed line is broken: the line needs repairs and cannot be switched ON for this number of timesteps

n_timesteps_horizon_maintenance number of future timesteps for which previsions of maintenance are provided in an Observation

max_number_prods_game_over maximum (inclusive) number of isolated productions tolerated before a game over signal is raised

max_number_loads_game_over maximum (inclusive) number of isolated consumptions tolerated before a game over signal is raised

n_timesteps_actionned_line_reactionable cooldown in timesteps on the activations of lines: number of timesteps to wait before a controler-activated line (switched ON or OFF) can be activated again by the controler

n_timesteps_actionned_node_reactionable cooldown in timesteps on the activations of substations: number of timesteps to wait before a controler-activated substation (any node-splitting operation) can be activated again by the controler

n_timesteps_pending_line_reactionable_when_overflowed *not supported yet*

n_timesteps_pending_node_reactionable_when_overflowed *not supported yet*

max_number_actionned_substations per timestep maximum (inclusive) number of separated controler-activated substations (ie with at least one node-splitting operation): an action with strictly more activated substations than this value is replaced by a do-nothing action

max_number_actionned_lines per timestep maximum (inclusive) number of separated controler-activated lines (ie switched ON or OFF): an action with strictly more activated lines than this value is replaced by a do-nothing action

max_number_actionned_total per timestep maximum (inclusive) number of separated controler-activated lines+substations: an action with strictly more activated lines+substations than this value is replaced by a do-nothing action

Here is the default **configuration.yaml** (produced by the template-creater script):

Listing 11: configuration.yaml

```yaml
loadflow_backend: pypower
#loadflow_backend: matpower

loadflow_mode: AC  # alternative current: more precise model but longer to process
#loadflow_mode: DC  # direct current: more simplist and faster model

max_seconds_per_timestep: 1.0  # time in seconds before player is timedout

hard_overflow_coefficient: 1.5  # % of line capacity usage above which a line will break bc of hard overflow
n_timesteps_hard_overflow_is_broken: 10  # number of timesteps a hard overflow broken line is broken

n_timesteps_consecutive_soft_overflow_breaks: 3  # number of consecutive timesteps for a line to be overflowed b4 break
n_timesteps_soft_overflow_is_broken: 5  # number of timesteps a soft overflow broken line is broken

n_timesteps_horizon_maintenance: 20  # number of immediate future timesteps for planned maintenance prevision

max_number_prods_game_over: 10  # number of tolerated isolated productions before game over
max_number_loads_game_over: 10  # number of tolerated isolated loads before game over

n_timesteps_actionned_line_reactionable: 3  # number of consecutive timesteps before a switched line can be switched again
n_timesteps_actionned_node_reactionable: 3  # number of consecutive timesteps before a topology-changed node can be changed again
n_timesteps_pending_line_reactionable_when_overflowed: 1 # number of cons. timesteps before a line waiting to be reactionable is reactionable if it is overflowed
n_timesteps_pending_node_reactionable_when_overflowed: 1 # number of cons. timesteps before a none waiting to be reactionable is reactionable if it has an overflowed line

max_number_actionned_substations: 7  # max number of changes tolerated in number of substations per timestep; actions with more than max_number_actionned_substations have at least one 1 value are replaced by do-nothing action
max_number_actionned_lines: 10  # max number of changes tolerated in number of lines per timestep; actions with more than max_number_actionned_lines are switched are replaced by do-nothing action
max_number_actionned_total: 15  # combination of 2 previous parameters; actions with more than max_number_total_actionned elements (substation or line) have a switch are replaced by do-nothing action
```

9.4 Reward signal file

The reward signal is the function that computes the reward which will be fed to the models at each timestep, after they perform an action given an observation. This is the typical reward function that feeds reinforcement learning models. pypownet is able to handle custom reward signals, as there is not yet particular reward functions that seem to drive the optimisation of useful dispatchers-like controlers. For a given environment, if not explicit reward signal is given, the simulator will use the default reward signal which always outputs 0: this implies no learning for models.

Formally, the reward signal should be a class `CustomRewardSignal` daughter class of `RewardSignal` (default reward signal), placed within each environment folder (e.g. in **default14/**). The python file containing this class should be named **reward_signal.py**, otherwise it won't be taken into account by the simulator. Here is the default reward signal:

Listing 12: pypownet/pypownet/reward_signal.py

```
class RewardSignal(object):
""" This is the basic template for the reward signal class that should at least implement a compute_reward
method with an observation of pypownet.environment.Observation, an action of pypownet.environment.Action and a flag
which is an exception of the environment package.
"""
def __init__(self):
    pass

def compute_reward(self, observation, action, flag):
    """ Effectively computes a reward given the current observation, the action taken (some actions are penalized)
    as well as the flag reward, which contains information regarding the latter game step, including the game
    over exceptions raised or illegal line reconnections.

    :param observation: an instance of pypownet.environment.Observation
    :param action: an instance of pypownet.game.Action
    :param flag: an exception either of pypownet.environment.DivergingLoadflowException,
    pypownet.environment.IllegalActionException, pypownet.environment.TooManyProductionsCut or
    pypownet.environment.TooManyConsumptionsCut
    :return: a list of subrewards as floats or int (potentially a list with only one value)
    """
    return [0.]
```

`CustomRewardSignal` should at least implement a function `CustomRewardSignal.compute_reward` which takes as input:

(i) the current observation of the simulated grid system

(ii) the last action played by the player (which lead to the above observation)

(iii) the simulator flag, which is an instance of a customized Exception of pypownet indicating game over triggers if any (i.e. if the last action lead to a game over)

The current observation is an instance of **pypownet.environment.Observation**, see *Reading observations* for further information about observations. The last action is an instance of **pypownet.game.Action**. The simulator flag is either None if the last step did not lead to a game over. However, if the last step lead to a game over, the input flag will be of either type, representing various types of pypownet exceptions. *flag* will be an instance of either exceptions:

(a) **pypownet.environment.DivergingLoadflowException**: game over provocked by a non-converging grid; might happend when the grid is not connexe, or in too poor shape such that flows diverge

(b) **pypownet.environment.TooManyProductionsCut**: the number of isolated productions has exceeded the maximum number of tolerated isolated productions; see *Configuration file*

(c) **pypownet.environment.TooManyConsumptionsCut**: the number of isolated consumptions has exceeded the maximum number of tolerated isolated consumptions; see *Configuration file*

(d) **pypownet.environment.IllegalActionException**: at least one illegal action (such as reconnecting unavailable broken lines) has been performed

Among those exceptions, **pypownet.environment.IllegalActionException** is special: this is the only one which does not mean that there wxas a game over. Actually, if some lines status are attempted to be switched while the associated lines are broken, the simulator will simply change the action such that the switch is deactivated, without any cost; for practical justifications, we could imagine an automatous mechanism that checks whether a line is available before switching its status.

Here is a concrete example of a custom reward signal used in the environment **default14/** (for more insight about this class, see *Default environments*):

Listing 13: pypownet/parameters/default14/reward_signal.py

```
import pypownet.environment
import pypownet.reward_signal
import numpy as np


class CustomRewardSignal(pypownet.reward_signal.RewardSignal):
    def __init__(self):
        super().__init__()

        constant = 14

        # Hyper-parameters for the subrewards
        # Mult factor for line capacity usage subreward
        self.multiplicative_factor_line_usage_reward = -1.
        # Multiplicative factor for total number of differed nodes in the grid and reference grid
        self.multiplicative_factor_distance_initial_grid = -.02
        # Multiplicative factor total number of isolated prods and loads in the grid
        self.multiplicative_factor_number_loads_cut = -constant / 5.
        self.multiplicative_factor_number_prods_cut = -constant / 10.

        # Reward when the grid is not connexe (at least two islands)
        self.connexity_exception_reward = -constant
        # Reward in case of loadflow software error (e.g. 0 line ON)
        self.loadflow_exception_reward = -constant

        # Multiplicative factor for the total number of illegal lines reconnections
        self.multiplicative_factor_number_illegal_lines_reconnection = -constant / 100.

        # Reward when the maximum number of isolated loads or prods are exceeded
        self.too_many_productions_cut = -constant
        self.too_many_consumptions_cut = -constant

        # Action cost reward hyperparameters
        self.multiplicative_factor_number_line_switches = -.2  # equivalent to - cost of line switch
        self.multiplicative_factor_number_node_switches = -.1  # equivalent to - cost of node switch

    def compute_reward(self, observation, action, flag):
        # First, check for flag raised during step, as they indicate errors from grid computations (usually game over)
        if flag is not None:
```

(continues on next page)

(continued from previous page)

```
            if isinstance(flag, pypownet.environment.DivergingLoadflowException):
                reward_aslist = [0., 0., -self.__get_action_cost(action), self.
↪loadflow_exception_reward, 0.]
            elif isinstance(flag, pypownet.environment.IllegalActionException):
                # If some broken lines are attempted to be switched on, put the␣
↪switches to 0, and add penalty to
                # the reward consequent to the newly submitted action
                reward_aslist = self.compute_reward(observation, action, flag=None)
                n_illegal_reconnections = np.sum(flag.illegal_lines_reconnections)
                illegal_reconnections_subreward = self.multiplicative_factor_number_
↪illegal_lines_reconnection * \
                                                  n_illegal_reconnections
                reward_aslist[2] += illegal_reconnections_subreward
            elif isinstance(flag, pypownet.environment.TooManyProductionsCut):
                reward_aslist = [0., self.too_many_productions_cut, 0., 0., 0.]
            elif isinstance(flag, pypownet.environment.TooManyConsumptionsCut):
                reward_aslist = [self.too_many_consumptions_cut, 0., 0., 0., 0.]
            else:  # Should not happen
                raise flag
        else:
            # Load cut reward
            number_cut_loads = sum(observation.are_loads_cut)
            load_cut_reward = self.multiplicative_factor_number_loads_cut * number_
↪cut_loads

            # Prod cut reward
            number_cut_prods = sum(observation.are_productions_cut)
            prod_cut_reward = self.multiplicative_factor_number_prods_cut * number_
↪cut_prods

            # Reference grid distance reward
            reference_grid_distance = self.__get_distance_reference_grid(observation)
            reference_grid_distance_reward = self.multiplicative_factor_distance_
↪initial_grid * reference_grid_distance

            # Action cost reward: compute the number of line switches, node switches,␣
↪and return the associated reward
            action_cost_reward = -self.__get_action_cost(action)

            # The line usage subreward is the sum of the square of the lines capacity␣
↪usage
            lines_capacity_usage = self.__get_lines_capacity_usage(observation)
            line_usage_reward = self.multiplicative_factor_line_usage_reward * np.
↪sum(np.square(lines_capacity_usage))

            # Format reward
            reward_aslist = [load_cut_reward, prod_cut_reward, action_cost_reward,␣
↪reference_grid_distance_reward,
                             line_usage_reward]

        return reward_aslist

    def __get_action_cost(self, action):
        # Action cost reward: compute the number of line switches, node switches, and␣
↪return the associated reward
        """ Compute the >=0 cost of an action. We define the cost of an action as the␣
↪sum of the cost of node-splitting
```

(continues on next page)

(continued from previous page)

```
        and the cost of lines status switches. In short, the function sums the number of 1 in the action vector, since
        they represent activation of switches. The two parameters self.cost_node_switch and self.cost_line_switch
        control resp the cost of 1 node switch activation and 1 line status switch activation.

        :param action: an instance of Action or a binary numpy array of length self.action_space.n
        :return: a >=0 float of the cost of the action
        """
        # Computes the number of activated switches of the action
        number_line_switches = np.sum(action.get_lines_status_subaction())

        number_prod_nodes_switches = np.sum(action.get_prods_switches_subaction())
        number_load_nodes_switches = np.sum(action.get_loads_switches_subaction())
        number_line_or_nodes_switches = np.sum(action.get_lines_or_switches_subaction())
        number_line_ex_nodes_switches = np.sum(action.get_lines_ex_switches_subaction())
        number_node_switches = number_prod_nodes_switches + number_load_nodes_switches + \
                               number_line_or_nodes_switches + number_line_ex_nodes_switches

        action_cost = self.multiplicative_factor_number_node_switches * number_node_switches + \
                      self.multiplicative_factor_number_line_switches * number_line_switches
        return action_cost

    @staticmethod
    def __get_lines_capacity_usage(observation):
        ampere_flows = observation.ampere_flows
        thermal_limits = observation.thermal_limits
        lines_capacity_usage = np.divide(ampere_flows, thermal_limits)
        return lines_capacity_usage

    @staticmethod
    def __get_distance_reference_grid(observation):
        # Reference grid distance reward
        """ Computes the distance of the current observation with the reference grid (i.e. initial grid of the game).
        The distance is computed as the number of different nodes on which two identical elements are wired. For
        instance, if the production of first current substation is wired on the node 1, and the one of the first initial
        substation is wired on the node 0, then their is a distance of 1 (there are different) between the current and
        reference grid (for this production). The total distance is the sum of those values (0 or 1) for all the
        elements of the grid (productions, loads, origin of lines, extremity of lines).

        :return: the number of different nodes between the current topology and the initial one
        """
```

(continues on next page)

(continued from previous page)

```
125         #initial_topology = np.asarray(self.game.get_initial_topology())
126         initial_topology = np.concatenate((observation.initial_productions_nodes,␣
    ↪observation.initial_loads_nodes,
127                                            observation.initial_lines_or_nodes,␣
    ↪observation.initial_lines_ex_nodes))
128         current_topology = np.concatenate((observation.productions_nodes, observation.
    ↪loads_nodes,
129                                            observation.lines_or_nodes, observation.
    ↪lines_ex_nodes))
130 
131         return np.sum((initial_topology != current_topology))  # Sum of nodes that␣
    ↪are different
```

CHAPTER 10

Changelog

10.1 vx.y.z

Date todate

This version blabla

10.1.1 New Features

10.1.2 Fixes

10.1.3 Other Changes

Note: bla

CHAPTER 11

API reference

11.1 Module agent

class pypownet.agent.**ActIOnManager**(*destination_path='saved_actions.csv'*, *delete=True*)

dump(*action*)

static load(*filepath*)

class pypownet.agent.**ActionsFileReaderControler**(*environment*)

act(*observation*)

Produces an action given an observation of the environment.

Takes as argument an observation of the current state, and returns the chosen action of class Action or np array.

class pypownet.agent.**Agent**(*environment*)

The template to be used to create an agent: any controller of the power grid is expected to be a daughter of this class.

act(*observation*)

Produces an action given an observation of the environment.

Takes as argument an observation of the current state, and returns the chosen action of class Action or np array.

feed_reward(*action*, *consequent_observation*, *rewards_aslist*)

class pypownet.agent.**DoNothing**(*environment*)

act(*observation*)

Produces an action given an observation of the environment.

Takes as argument an observation of the current state, and returns the chosen action of class Action or np array.

class pypownet.agent.**FlowsSaver**(*environment*)

act(*observation*)
Produces an action given an observation of the environment.

Takes as argument an observation of the current state, and returns the chosen action of class Action or np array.

class pypownet.agent.**GreedySearch**(*environment*)
This agent is a tree-search model of depth 1, that is constrained to modifiying at most 1 substation configuration or at most 1 line status. This controller used the simulate method of the environment, by testing every 1-line status switch action, every new configuration for substations with at least 4 elements, as well as the do-nothing action. Then, it will seek for the best reward and return the associated action, expecting the maximum reward for the action pool it can reach. Note that the simulate method is only an approximation of the step method of the environment, and in three ways: * simulate uses the DC mode, while step is in AC * simulate uses only the predictions given to the player to simulate the next timestep injections * simulate can not compute the hazards that are supposed to come at the next timestep

act(*observation*)
Produces an action given an observation of the environment.

Takes as argument an observation of the current state, and returns the chosen action of class Action or np array.

class pypownet.agent.**RandomAction**(*environment*)
An example of a baseline controller that produce random actions (ie random line switches and random node switches.

act(*observation*)
Produces an action given an observation of the environment.

Takes as argument an observation of the current state, and returns the chosen action of class Action or np array.

class pypownet.agent.**RandomLineSwitch**(*environment*)
An example of a baseline controller that randomly switches the status of one random power line per timestep (if the random line is previously online, switch it off, otherwise switch it on).

act(*observation*)
Produces an action given an observation of the environment.

Takes as argument an observation of the current state, and returns the chosen action of class Action or np array.

class pypownet.agent.**RandomNodeSplitting**(*environment*)
Implements a "random node-splitting" agent: at each timestep, this controller will select a random substation (id), then select a random switch configuration such that switched elements of the selected substations change the node within the substation on which they are directly wired.

act(*observation*)
Produces an action given an observation of the environment.

Takes as argument an observation of the current state, and returns the chosen action of class Action or np array.

class pypownet.agent.**RandomPointAction**(*environment*)
An example of a baseline controller that produce 1 random activation (ie an array with all 0 but one 1).

act(*observation*)
Produces an action given an observation of the environment.

Takes as argument an observation of the current state, and returns the chosen action of class Action or np array.

class pypownet.agent.**TreeSearchLineServiceStatus**(*environment*)
Exhaustive tree search of depth 1 limited to no action + 1 line switch activation

act(*observation*)
Produces an action given an observation of the environment.

Takes as argument an observation of the current state, and returns the chosen action of class Action or np array.

11.2 Module grid

exception pypownet.grid.**DivergingLoadflowException**(*last_observation*, **args*)

class pypownet.grid.**Grid**(*loadflow_backend*, *src_filename*, *dc_loadflow*, *new_imaps*)

apply_topology(*new_topology*)
Applies a new topology to self. topology should be an instance of class Topology, with computed values to be replaced in self.

Parameters **new_topology** – an instance of Topology, with destination values for the nodes values/lines service status

compute_loadflow(*fname_end*)
Given the current state of the grid (topology + injections), compute the new loadflow of the grid. This function subtreats the Octave pipeline to self.__vanilla_matpower_callback.

Returns 0 for failed computation, 1 for success

Raises ***DivergingLoadflowException*** – if the loadflow did not converge, raise diverging exception (could be because of grid not connexe, or voltages issues, or angle issues etc).

compute_topological_mapping_permutation()
Computes a permutation that shuffles the construction order of a topology (prods->loads->lines or->lines ex) into a representation where all elements of a substation are consecutive values (same order, but locally). By construction, the topological vector is the concatenation of the subvectors: productions nodes (for each value, on which node, 0 or 1, the prod is wired), loads nodes, lines origin nodes, lines extremity nodes and the lines service status.

This function should only be called once, at the instantiation of the grid, for it computes the fixed mapping function for the remaining of the game (also fixed along games).

discard_flows()

export_lines_capacity_usage(*safe_mode=False*)
Computes and returns the lines capacity usage, i.e. the elementwise division of the flows in Ampere by the lines nominal thermal limit.

Returns a list of size the number of lines of positive values

export_to_observation()
Exports the current grid state into an observation.

extract_flows_a(*safe_mode=False*)

get_lines_status()

get_number_elements()

get_thermal_limits()

get_topology()

load_timestep_injections(*timestep_injections*, *prods_p=None*, *prods_v=None*, *loads_p=None*, *loads_q=None*)

Loads a scenario from class Scenario: contains P and V values for prods, and P and Q values for loads. Other timestep entries are loaded using other modules (including pypownet.game). If one of input except TimestepInjections are not None, they are all used for next injections (used in simulate with planned injections).

Parameters **timestep_injections** – an instance of class Scenario

Returns if do_trigger_lf_computation then the result of self.compute_loadflow else nothing

normalize_prods_voltages(*voltages*)

set_flows_to_0()

set_lines_status(*new_lines_status*)

set_voltage_angles(*new_voltage_angles*)

set_voltage_magnitudes(*new_voltage_magnitudes*)

exception pypownet.grid.**GridNotConnexeException**(*last_observation*, **args*)

class pypownet.grid.**Topology**(*prods_nodes*, *loads_nodes*, *lines_or_nodes*, *lines_ex_nodes*, *mapping_array*)

This class is a container for the topology lists defining the current topological state of a grid. Topology should be manipulated using this class, as it maintains the adopted convention consistently.

get_length()

get_unzipped()

get_zipped()

static unzip(*topology*, *n_prods*, *n_loads*, *n_lines*, *invert_mapping_function*)

pypownet.grid.**compute_flows_a**(*active*, *reactive*, *voltage*, *are_lines_on*)

11.3 Module env

class pypownet.environment.**ActionSpace**(*number_generators*, *number_consumers*, *number_power_lines*, *number_substations*, *substations_ids*, *prods_subs_ids*, *loads_subs_ids*, *lines_or_subs_id*, *lines_ex_subs_id*)

_verify_action_shape(*action*)

array_to_action(*array*)

Converts and returns an pypownet.game.Action from a array-object (e.g. list, numpy arrays).

Parameters **array** – array-style object

Returns an instance of pypownet.game.Action equivalent to input action

Raises **ValueError** – the input array is not of the same length than the expected action (self.action_length)

get_do_nothing_action(*as_class_Action=False*)
Creates and returns an action equivalent to a do-nothing: all of the activable switches are 0 i.e. not activated.

Returns an instance of pypownet.game.Action that is equivalent to an action doing nothing

static get_lines_status_switch_from_id(*action*, *line_id*)

get_lines_status_switches_of_substation(*action*, *substation_id*)

get_number_elements_of_substation(*substation_id*)

get_substation_switches_in_action(*action*, *substation_id*, *concatenated_output=True*)
From the input action, retrieves the list of value of the switch (0 or 1) of the switches on which each element of the substation with input id. This function also computes the type of element associated to each switch value of the returned switches-value list.

Parameters

- **action** – input action whether a numpy array or an element of class pypownet.game.Action.
- **substation_id** – an integer of the id of the substation to retrieve the switches of its elements in the input action.
- **concatenated_output** – False to return an array per elementype, True to return a single concatenated array

Returns a switch-values list (binary list) in the order: production (<=1), loads (<=1), lines origins, lines extremities; also returns a ElementType list of same size, where each value indicates the type of element associated to each first-returned list values.

static set_lines_status_switch_from_id(*action*, *line_id*, *new_switch_value*)

set_lines_status_switches_of_substation(*action*, *substation_id*, *new_configuration*)

set_substation_switches_in_action(*action*, *substation_id*, *new_values*)
Replaces the switches (binary) values of the input substation in the input action with the new specified values. Note that the mapping between the new values and the elements of the considered substation are the same as the one retrieved by the opposite function self.get_substation_switches. Consequently, the length of the array new_values is len(self.get_substation_switches(action, substation_id)[1]).

Parameters

- **action** – input action whether a numpy array or an element of class pypownet.game.Action.
- **substation_id** – an integer of the id of the substation to retrieve the switches of its elements in the input action

Returns the modified action; WARNING: the input action is not modified in place if of array type: ensure that you catch the returned action as the modified action.

exception pypownet.environment.**DivergingLoadflowException**(*last_observation*, **args*)

class pypownet.environment.**ElementType**
An enumeration.

CONSUMPTION = 'consumption'

EXTREMITY_POWER_LINE = 'extremity of power line'

ORIGIN_POWER_LINE = 'origin of power line'

PRODUCTION = 'production'

exception pypownet.environment.**IllegalActionException**(*text*, *illegal_lines_reconnections*, *illegal_unavailable_lines_switches*, *illegal_oncoolown_substations_switches*, **args*)

class pypownet.environment.**MinimalistACObservation**(*active_loads*, *reactive_loads*, *voltage_loads*, *active_productions*, *reactive_productions*, *voltage_productions*, *active_flows_origin*, *reactive_flows_origin*, *voltage_flows_origin*, *active_flows_extremity*, *reactive_flows_extremity*, *voltage_flows_extremity*, *ampere_flows*, *lines_status*, *are_loads_cut*, *are_productions_cut*, *timesteps_before_lines_reconnectable*, *timesteps_before_lines_reactionable*, *timesteps_before_nodes_reactionable*, *timesteps_before_planned_maintenance*, *planned_active_loads*, *planned_reactive_loads*, *planned_active_productions*, *planned_voltage_productions*, *date_year*, *date_month*, *date_day*, *date_hour*, *date_minute*, *date_second*, *productions_nodes*, *loads_nodes*, *lines_or_nodes*, *lines_ex_nodes*)

as_array()

as_dict()

as_minimalist()

class pypownet.environment.**MinimalistObservation**(*active_loads*, *active_productions*, *ampere_flows*, *lines_status*, *are_loads_cut*, *are_productions_cut*, *timesteps_before_lines_reconnectable*, *timesteps_before_lines_reactionable*, *timesteps_before_nodes_reactionable*, *timesteps_before_planned_maintenance*, *planned_active_loads*, *planned_active_productions*, *date_year*, *date_month*, *date_day*, *date_hour*, *date_minute*, *date_second*, *productions_nodes*, *loads_nodes*, *lines_or_nodes*, *lines_ex_nodes*)

as_array()

as_dict()

class pypownet.environment.**Observation**(*substations_ids*, *active_loads*, *reactive_loads*, *voltage_loads*, *active_productions*, *reactive_productions*, *voltage_productions*, *active_flows_origin*, *reactive_flows_origin*, *voltage_flows_origin*, *active_flows_extremity*, *reactive_flows_extremity*, *voltage_flows_extremity*, *ampere_flows*, *thermal_limits*, *lines_status*, *are_loads_cut*, *are_productions_cut*, *loads_substations_ids*, *productions_substations_ids*, *lines_or_substations_ids*, *lines_ex_substations_ids*, *timesteps_before_lines_reconnectable*, *timesteps_before_lines_reactionable*, *timesteps_before_nodes_reactionable*, *timesteps_before_planned_maintenance*, *planned_active_loads*, *planned_reactive_loads*, *planned_active_productions*, *planned_voltage_productions*, *date_year*, *date_month*, *date_day*, *date_hour*, *date_minute*, *date_second*, *productions_nodes*, *loads_nodes*, *lines_or_nodes*, *lines_ex_nodes*, *initial_productions_nodes*, *initial_loads_nodes*, *initial_lines_or_nodes*, *initial_lines_ex_nodes*)

The class State is a container for all the values representing the state of a given grid at a given time. It contains the following values: * The active and reactive power values of the loads * The active power values and the voltage setpoints of the productions * The values of the power through the lines: the active and reactive values at the origin/extremity of the lines as well as the lines capacity usage * The exhaustive topology of the grid, as a stacked vector of one-hot vectors

as_ac_minimalist()

as_array()

as_dict()

as_minimalist()

get_lines_capacity_usage()

get_lines_status_of_substation(*substation_id*)

From the current observation, retrieves the list of lines status (binary) from lines connected to the input substations. This function also computes and retrieve the list of ifs of ids at the other end of each corresponding lines.

Parameters **substation_id** – an integer of the id of the substation to retrieve the nodes on which its elements are wired

Returns (consistently fixed-order list of binary (0 or 1) values, list of ids of other end substations)

get_nodes_of_substation(*substation_id*)

From the current observation, retrieves the list of value of the nodes on which each element of the substation with input id. This function also computes the type of element associated to each node value of the returned nodes-value list.

Parameters **substation_id** – an integer of the id of the substation to retrieve the nodes on which its elements are wired

Returns a nodes-values list in the order: production (<=1), loads (<=1), lines origins, lines extremities; also returns a ElementType list of same size, where each value indicates the type of element associated to each first-returned list values.

class pypownet.environment.**ObservationSpace**(*number_generators*, *number_consumers*, *number_power_lines*, *number_substations*, *n_timesteps_horizon_maintenance*)

array_to_observation(*array*)
Converts and returns an pypownet.game.Observation from a array-object (e.g. list, numpy arrays).

Parameters **array** – array-style object

Returns an instance of pypownet.game.Action equivalent to input action

Raises **ValueError** – the input array is not of the same length than the expected action (self.action_length)

class pypownet.environment.**RunEnv**(*parameters_folder*, *game_level*, *chronic_looping_mode='natural'*, *start_id=0*, *game_over_mode='soft'*, *renderer_latency=None*, *without_overflow_cutoff=False*, *seed=None*)

static _RunEnv__wrap_exception(*flag*)

_get_obs()

get_current_chronic_name()

get_current_datetime()

get_observation(*as_array=True*)

is_action_valid(*action*)

process_game_over()

render(*game_over=False*)

reset()
Instantiate the game Environment based on the specified parameters.

simulate(*action*, *do_sum=True*)
Computes the reward of the simulation of action to the current grid.

step(*action*, *do_sum=True*)
Performs a game step given an action. The as list pattern is: load_cut_reward, prod_cut_reward, action_cost_reward, reference_grid_distance_reward, line_usage_reward

exception pypownet.environment.**TooManyConsumptionsCut**(**args*)

exception pypownet.environment.**TooManyProductionsCut**(**args*)

11.4 Module game

class pypownet.game.**Action**(*prods_switches_subaction*, *loads_switches_subaction*, *lines_or_switches_subaction*, *lines_ex_switches_subaction*, *lines_status_subaction*, *substations_ids*, *prods_subs_ids*, *loads_subs_ids*, *lines_or_subs_id*, *lines_ex_subs_id*, *elementtype*)

as_array()

get_lines_ex_switches_subaction()

get_lines_or_switches_subaction()

get_lines_status_subaction()

get_loads_switches_subaction()

get_node_splitting_subaction()

get_prods_switches_subaction()

get_substation_switches(*substation_id*, *concatenated_output=True*)

set_as_do_nothing()

set_node_splitting_subaction(*new_node_splitting_subaction*)

set_substation_switches(*substation_id*, *new_values*)

exception pypownet.game.**DivergingLoadflowException**(*last_observation*, **args*)

class pypownet.game.**Game**(*parameters_folder*, *game_level*, *chronic_looping_mode*, *chronic_starting_id*, *game_over_mode*, *renderer_frame_latency*, *without_overflow_cutoff*)

apply_action(*action*)
Applies an action on the current grid (topology). The action is first into lists of same objects (e.g. nodes on which productions are connected), then the destination values are computed, such that the grid will replace its current topology with the latter. Since actions come from pypownet.env.RunEnv.Action, they are switches. Here, given the last values of the grid and the switches, this function computes the actual destination values (e.g. switch line status of line 10: if line 10 is on, put its status to off i.e. 0, otherwise put to on i.e. 1)

Parameters **action** – an instance of pypownet.env.RunEnv.Action

export_observation()
Retrieves an observation of the current state of the grid.

Returns an instance of class pypownet.env.Observation

get_changed_substations(*action*)
Computes the boolean array of changed substations from an Action.

get_current_chronic_name()

get_current_datetime()

get_current_timestep_id()
Retrieves the current index of scenario; this index might differs from a natural counter (some id may be missing within the chronic).

Returns an integer of the id of the current scenario loaded

get_initial_topology()
Retrieves the initial topology of the grid (when it was initially loaded). This is notably used to reinitialize the grid after a game over.

Returns an instance of pypownet.grid.Topology or a list of integers

get_max_seconds_per_timestep()

get_next_chronic()

get_number_elements()

get_reward_signal_class()

get_substations_ids()

get_substations_ids_lines_ex()

get_substations_ids_lines_or()

get_substations_ids_loads()

get_substations_ids_prods()

is_action_valid(*action*)

load_entries_from_next_timestep(*is_simulation=False*)
Loads the next timestep injections (set of injections, maintenance, hazard etc for the next timestep id).

Returns

raise ValueError raised in the case where they are no more scenarios available

load_entries_from_timestep_id(*timestep_id*, *is_simulation=False*, *silence=False*)

parameters_environment_tostring()

process_game_over()
Handles game over behavior of the game: put the grid topology to the initial one + if restart is True, then the game will load the first set of injections (i)_{t0}, otherwise the next set of injections of the chronics (i)_{t+1}.

render(*rewards*, *game_over=False*, *cascading_frame_id=None*, *date=None*, *timestep_id=None*)
Initializes the renderer if not already done, then compute the necessary values to be carried to the renderer class (e.g. sum of consumptions).

Parameters

- **rewards** – list of subrewards of the last timestep (used to plot reward per timestep)
- **game_over** – True to plot a "Game over!" over the screen if game is over

Returns

raise ImportError pygame not found raises an error (it is mandatory for the renderer)

reset_grid()
Reinitialized the grid by applying the initial topology to the current state (topology).

simulate(*action*)

step(*action*, *_is_simulation=False*)

exception pypownet.game.**IllegalActionException**(*text*, *has_too_much_activations*, *illegal_lines_reconnections*, *illegal_unavailable_lines_switches*, *illegal_oncoolown_substations_switches*, **args*)

get_has_too_much_activations()

get_illegal_broken_lines_reconnections()

get_illegal_oncoolown_lines_switches()

get_illegal_oncoolown_substations_switches()

is_empty

exception pypownet.game.**ListExceptions**(*exceptions*)

exception pypownet.game.**NoMoreScenarios**

exception pypownet.game.**TooManyConsumptionsCut**(**args*)

exception pypownet.game.**TooManyProductionsCut**(**args*)

11.5 Module main

pypownet.main.**main**()

11.6 Module renderer

class pypownet.renderer.**Renderer**(*grid_case*, *or_ids*, *ex_ids*, *are_prods*, *are_loads*, *timestep_duration_seconds*)

create_plot_loads_curve(*n_timesteps*, *left_xlabel*)

static draw_plot_game_over()

static draw_plot_pause()

draw_surface_diagnosis(*number_loads_cut*, *number_prods_cut*, *number_nodes_splitting*, *number_lines_switches*, *distance_initial_grid*, *line_capacity_usage*, *n_offlines_lines*, *number_unavailable_lines*, *number_unavailable_nodes*, *max_number_isolated_loads*, *max_number_isolated_prods*)

draw_surface_grid(*relative_thermal_limits*, *lines_por*, *lines_service_status*, *prods*, *loads*, *are_substations_changed*, *number_nodes_per_substation*)

draw_surface_legend()

draw_surface_loads_curves(*n_hours_to_display_top_loadplot*, *n_hours_to_display_bottom_loadplot*)

draw_surface_n_overflows(*n_timesteps*, *left_xlabel=' 7 days ago '*)

draw_surface_nodes_headers(*scenario_id*, *date*, *cascading_result_frame*)

draw_surface_relative_thermal_limits(*n_timesteps*, *left_xlabel='24 hours ago'*)

render(*lines_capacity_usage*, *lines_por*, *lines_service_status*, *epoch*, *timestep*, *scenario_id*, *prods*, *loads*, *date*, *are_substations_changed*, *number_nodes_per_substation*, *number_loads_cut*, *number_prods_cut*, *number_nodes_splitting*, *number_lines_switches*, *distance_initial_grid*, *number_off_lines*, *number_unavailable_lines*, *number_unactionable_nodes*, *max_number_isolated_loads*, *max_number_isolated_prods*, *game_over=False*, *cascading_frame_id=None*)

pypownet.renderer.**recenter**()

pypownet.renderer.**scale**(*u*, *z*, *t*)

11.7 Module runner

class pypownet.runner.**Runner**(*environment*, *agent*, *render=False*, *verbose=False*, *vverbose=False*, *parameters=None*, *level=None*, *max_iter=None*, *log_filepath='runner.log'*, *machinelog_filepath='machine_logs.csv'*)

dump_machinelogs(*timestep_id*, *done*, *reward*, *reward_aslist*, *cumul_rew*, *datetime*)

loop(*iterations*, *epochs=1*)
Runs the simulator for the given number of iterations time the number of episodes. :param iterations: int of number of iterations per episode :param epochs: int of number of episodes, each resetting the environment at the beginning :return:

step(*observation*)
Performs a full RL step: the agent acts given an observation, receives and process the reward, and the env is resetted if done was returned as True; this also logs the variables of the system including actions, observations. :param observation: input observation to be given to the agent :return: (new observation, action taken, reward received)

exception pypownet.runner.**TimestepTimeout**

11.8 Module scenarios_chronic

class pypownet.chronic.**Chronic**(*source_folder*, *with_previsions=True*)

construct_timesteps_injections()
Loop over all the pertinent data row by row creating scenarios that are stored within the self.scenarios container.

static get_csv_content(*csv_absolute_fpath*)

get_imaps()

get_planned_maintenance(*timestep_id*, *horizon*)

get_timestep_duration()

get_timestep_entries(*timestep_id*)

get_timestep_ids()

import_data(*data*)

retrieve_data()

retrieve_input_files()

class pypownet.chronic.**ChronicLooper**(*chronics_folder*, *game_level*, *start_id*, *looping_mode*)

get_current_chronic_name()

get_next_chronic_folder()

class pypownet.chronic.**TimestepEntries**(*timestep_id*, *loads_p*, *loads_q*, *prods_p*, *prods_v*, *maintenance*, *hazards*, *date*, *planned_loads_p=None*, *planned_loads_q=None*, *planned_prods_p=None*, *planned_prods_v=None*)

get_datetime()

get_hazards()

get_id()

get_loads_p()

get_loads_q()

get_maintenance()

get_planned_loads_p()

get_planned_loads_q()

get_planned_prods_p()

get_planned_prods_v()

get_prods_p()

get_prods_v()

11.9 Indices and tables

- genindex
- genindex
- modindex

Python Module Index

p

Index

Symbols

A

C

D

E

F

G

I

L

M