FRobs_RL Documentation

Release 0.1

Fajardo

Jun 21, 2022

USER GUIDE

FROBS_RL (Flexible Robotics Reinforcement Learning Library) is a flexible robotics reinforcement learning (RL) library. It is primarly designed to be used in robotics applications using the ROS framework. It is written in **Python** and uses libraries based on the *PyTorch* framework to handle the machine learning. The library uses *OpenAI Gym* to create and handle the RL environments, *stable-baselines3* to provide state-of-the-art RL algorithms, *Gazebo* to simulate the physical environments, and *XTerm* to display and launch many of the **ROS** nodes and processes in a lightweight terminal.

FRobs_RL is stored in a Github repository: https://github.com/jmfajardod/frobs_rl

CHAPTER

ONE

MAIN GOALS

- Provide a framework to easily train and deploy RL algorithms in robotics applications using the ROS middleware.
- Provide a framework to easily create RL enviroments for any type of task.
- Provide a framework to easily use, test or create state-of-the-art RL algorithms in robotics applications.

Note: This project is under active development.

CHAPTER

TWO

CONTENTS

2.1 Installation

FRobs_RL have been only tested in Ubuntu 20.04, althought it may work in other OS its performance have not been tested in other OS.

2.1.1 Prerequisites

ROS

As **FRobs_RL** uses ROS as the middleware to interact between the learning algorithm and the robotic hardware the user must have it installed. **FRobs_RL** have been tested using the *Noetic* distribution, the official documentation to install ROS Noetic can be found her Noetic Installation.

An abridged version of the commands to install *ROS* including the *Gazebo* simulator can be seen below.

```
sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_release -sc) main" > /etc/
↪apt/sources.list.d/ros-latest.list'
sudo apt install curl
curl -s https://raw.githubusercontent.com/ros/rosdistro/master/ros.asc | sudo apt-key␣
↪add -
sudo apt update
sudo apt install ros-noetic-desktop-full
echo "source /opt/ros/noetic/setup.bash" >> ~/.bashrc
source ~/.bashrc
sudo apt install python3-rosdep python3-rosinstall python3-rosinstall-generator python3-
↪wstool build-essential
sudo apt install python3-rosdep
sudo rosdep init
rosdep update
```

Catkin Tools

We recommend that the user install Catkin Tools to ease the setup of ROS workspaces and compilation of packages. To install it execute the commands:

```
sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu `lsb_release -sc` main" > /etc/
↪apt/sources.list.d/ros-latest.list'
wget http://packages.ros.org/ros.key -O - | sudo apt-key add -
sudo apt-get update
sudo apt-get install python3-catkin-tools
```

XTerm

FRobs_RL uses the *XTerm* terminal to launch different processes like ROS Nodes or the Gazebo simulator. To install *XTerm* execute:

```
sudo apt install xterm
```

2.1.2 Workspace creation and library compilation

To use the library it is necessary to download and compile the library package using Catkin. To create a new ROS workspace called *rl_ws* and compile the **frobs_rl** package, use the following commands:

```
cd ~
mkdir -p rl_ws/src
cd rl_ws
git clone https://github.com/jmfajardod/frobs_rl src/frobs_rl -b main
rosdep install -y --from-paths src --ignore-src --rosdistro ${ROS_DISTRO}
catkin config --extend /opt/ros/${ROS_DISTRO}
catkin build
source devel/setup.bash
```

After downloading the package and compiling it using Catkin it is necessary to download the *Python* dependencies. To download them using **pip** execute the following commands:

Note: By default **pip** will install a **PyTorch** version without GPU support, if your computer has a supported GPU follow the instructions on the official PyTorch website

---

```
roscd frobs_rl # Only works the setup.bash has been sourced
python3 -m pip install -r requirements.txt
```

2.2 Enviroment creation

To create a new enviroment with a robotic platform and a task the user must create a class in which the robot and task is defined. To facilitate many of the required steps needed to create an *env* (enviroment) following the OpenAI Gym documentation **FRobs_RL** already has a class which implements most of the required functions needed by Gym. The class is called **RobotBasicEnv** and can be inherited by any enviroment regardless of the robot or task. The main functions already implemented in **RobotBasicEnv** are:

- step: This is the funcion that will be executed in each step of the RL loop.
- reset: The funcion called when the enviroment must be reset, wheter it may be because of the sucess or because the time limit has been reached.
- close: The funcion that is executed when the enviroment is closed. It mainly makes sure that all ROS Nodes and the Gazebo simulator are properly closed.

To create a new enviroment the user must inherit the **RobotBasicEnv** and fill the next functions:

- _send_action: The function used to send the commands to the robot.
- _get_observation: The function to execute when observations from the enviroment are needed.
- _get_reward: The function that calculates and returns the reward based on the action of the agent.
- _check_if_done: The function to check if the robot sucess finished the task or reached the goal.
- _check_subs_and_pubs_connection (Optional): The function that check if the ROS subscribers and publishers are connected and properly receiving or sending messages.
- _set_episode_init_params (Optional): The function to set ROS or Gazebo initial parameters for the episode, e.g.: the initial position of the robot, the location of obstacles, etc.

To create a new enviroment we recommend that the user creates two different classes a **CustomRobotEnv** and a **CustomTaskEnv**, in this way the principal funcions are separted in the following way:

- All processes related to the *robot* are located in the **CustomRobotEnv**, this can be the URDF model loading, controllers spawning, spawning the robot in the simulator, etc.
- The **CustomTaskEnv** inherits the **CustomRobotEnv** and are where all processes related directly to the task are implemented, this can be the way to send actions to the robot agent, the process to obtain observations, the reward funcion, etc.

The previous arquitecture has the advantage that a **CustomRobotEnv** can be reused in many tasks, reducing the amount of code needed to create a new enviroment

In the next step a guide will be shown to how to use the templates of the **CustomRobotEnv** and **CustomTaskEnv** classes included in the **FRobs_RL** library.

Note: Althought the previous separation in **CustomRobotEnv** and **CustomTaskEnv** is recommended, the user can program the enviroment in any way inheriting the **RobotBasicEnv**.

---

class robot_BasicEnv.**RobotBasicEnv**(*launch_gazebo=False*, *gazebo_init_paused=True*, *gazebo_use_gui=True*, *gazebo_recording=False*, *gazebo_freq=100*, *world_path=None*, *world_pkg=None*, *world_filename=None*, *gazebo_max_freq=None*, *gazebo_timestep=None*, *spawn_robot=False*, *model_name_in_gazebo='robot'*, *namespace='/robot'*, *pkg_name=None*, *urdf_file=None*, *urdf_folder='/urdf'*, *controller_file=None*, *controller_list=None*, *urdf_xacro_args=None*, *rob_state_publisher_max_freq=None*, *rob_st_term=False*, *model_pos_x=0.0*, *model_pos_y=0.0*, *model_pos_z=0.0*, *model_ori_x=0.0*, *model_ori_y=0.0*, *model_ori_z=0.0*, *model_ori_w=0.0*, *reset_controllers=False*, *reset_mode=1*, *step_mode=1*, *num_gazebo_steps=1*)

Basic enviroment for all the robot environments in the frobs_rl library. To use a custom world, one can use two options: 1) set the path directly to the world file (`world_path`) or set the pkg name and world filename (`world_pkg` and `world_filename`).

Parameters

- **launch_gazebo** (*bool*) – If True, launch Gazebo at the start of the env.
- **gazebo_init_paused** (*bool*) – If True, Gazebo is initialized in a paused state.
- **gazebo_use_gui** (*bool*) – If True, Gazebo is launched with a GUI (through gzclient).
- **gazebo_recording** (*bool*) – If True, Gazebo is launched with a recording of the GUI (through gzclient).
- **gazebo_freq** (*int*) – The publish rate of gazebo in Hz.
- **world_path** (*str*) – If using a custom world then the path to the world.
- **world_pkg** (*str*) – If using a custom world then the package name of the world.
- **world_filename** (*str*) – If using a custom world then the filename of the world.
- **gazebo_max_freq** (*float*) – max update rate for gazebo in real time factor: 1 is real time, 10 is 10 times real time.
- **gazebo_timestep** (*float*) – The timestep of gazebo in seconds.
- **spawn_robot** (*bool*) – If True, the robot is spawned in the environment.
- **model_name_in_gazebo** (*str*) – The name of the model in gazebo.
- **namespace** (*str*) – The namespace of the robot.
- **pkg_name** (*str*) – The package name where the robot model is located.
- **urdf_file** (*str*) – The path to the urdf file of the robot.
- **urdf_folder** (*str*) – The path to the folder where the urdf files are located. Default is "/urdf".
- **urdf_xacro_args** (*str*) – The arguments to be passed to the xacro parser.
- **controller_file** (*str*) – The path to the controllers YAML file of the robot.
- **controller_list** (*list of str*) – The list of controllers to be launched.
- **rob_state_publisher_max_freq** (*int*) – The maximum frequency of the ros state publisher.
- **rob_st_term** (*bool*) – If True, the robot state publisher is launched in a new terminal.
- **model_pos_x** – The x position of the robot in the world.

- **model_pos_y** – The y position of the robot in the world.
- **model_pos_z** – The z position of the robot in the world.
- **model_ori_x** – The x orientation of the robot in the world.
- **model_ori_y** – The y orientation of the robot in the world.
- **model_ori_z** – The z orientation of the robot in the world.
- **model_ori_w** – The w orientation of the robot in the world.
- **reset_controllers** (*bool*) – If True, the controllers are reset at the start of each episode.
- **reset_mode** – If 1, reset Gazebo with a "reset_world" (Does not reset time) If 2, reset Gazebo with a "reset_simulation" (Resets time)
- **step_mode** – If 1, step Gazebo using the "pause_physics" and "unpause_physics" services. If 2, step Gazebo using the "step_simulation" command.
- **num_gazebo_steps** – If using step_mode 2, the number of steps to be taken.

_check_if_done()

Function to check if the episode is done.

If the episode has a success condition then set done as:
self.info['is_success'] = 1.0

_check_subs_and_pubs_connection()

Function to check if the gazebo and ros connections are ready

_get_observation()

Function to get the observation from the enviroment.

_get_reward()

Function to get the reward from the enviroment.

_reset_gazebo()

Function to reset the gazebo simulation.

_send_action(*action*)

Function to send an action to the robot

_set_episode_init_params()

Function to set some parameters, like the position of the robot, at the begining of each episode.

close()

Function to close the environment when training is done.

reset()

Function to reset the enviroment after an episode is done.

step(*action*)

Function to send an action to the robot and get the observation and reward.

2.3 Enviroment templates

As mentioned in the previous step, **FRobs_RL** includes two templates to facilitate the creation of a new Gym enviroment. These templates can be found in the folder of templates and are the two basic templates for the **CustomRobotEnv** and **CustomTaskEnv** classes.

2.3.1 CustomRobotEnv

The **CustomRobotEnv** must the first class to be filled, as the **CustomTaskEnv** will inherit this class. In the template *init* function there are already many parameters included like the parameters to launch Gazebo along the class, the parameters related to the the URDF model loading of the robot, the controllers spawning, namespaces of the parameters, etc. An exert of the function is shown below:

```
"""
If spawning the robot using the given spawner then set the corresponding environment␣
↪variables.
"""
spawn_robot=False
model_name_in_gazebo="robot"
namespace="/robot"
pkg_name=None
urdf_file=None
urdf_folder="/urdf"
controller_file=None
controller_list=None
urdf_xacro_args=None
rob_state_publisher_max_freq= None
model_pos_x=0.0; model_pos_y=0.0; model_pos_z=0.0
model_ori_x=0.0; model_ori_y=0.0; model_ori_z=0.0; model_ori_w=0.0
```

The user only needs to change this parameters and the inhered class **RobotBasicEnv** will spawn the robot with the position and orientation specified. An example of this is shown below:

```
spawn_robot=True
model_name_in_gazebo="kobuki_robot"
namespace="/"
pkg_name="kobuki_maze_rl"
urdf_file="kobuki_lidar.urdf.xacro"
urdf_folder="/urdf"
controller_file=None
controller_list=[]
urdf_xacro_args=None
rob_state_publisher_max_freq=30
model_pos_x=0.0; model_pos_y=10.0; model_pos_z=0.0
model_ori_x=0.0; model_ori_y=0.0; model_ori_z=0.0; model_ori_w=1.0
```

Note: Even if the user decides to spawn the robot without the use of the **RobotBasicEnv** class they can use the functions availiable in **FRobs_RL** like *spawn_model_in_gazebo*, *gazebo_set_time_step*, *urdf_load_from_pkg*, etc.

After filling the *init* function of the **CustomRobotEnv** the user can fill the optional function *_check_subs_and_pubs_connection* if all publishers or subscribers of the enviroment are declared inside this

class.

class CustomRobotEnv.**CustomRobotEnv**

Custom Robot Env, use this for all task envs using the custom robot.

_check_subs_and_pubs_connection()

Function to check if the Gazebo and ROS connections are ready

Finally, the user only needs to change the name of the class so the **CustomTaskEnv** can inherit it. An example is shown below:

```
# With default name
class CustomRobotEnv(robot_BasicEnv.RobotBasicEnv):

# With new name to allow class inheritance
class KobukiLIDAREnv(robot_BasicEnv.RobotBasicEnv):
```

2.3.2 CustomTaskEnv

After creating a **CustomRobotEnv** the user must fill the task related template **CustomTaskEnv**. The first change needed is the class that the Task enviroment will inherit and the name of the Task enviroment; the class to inherit must be appropiately imported and its name should be change in the class, an example is shown below:

```
# With default inherit
from frobs_rl.templates import CustomRobotEnv # Replace with your own robot env
class CustomTaskEnv(CustomRobotEnv.CustomRobotEnv):

# With appropiate inherit of created RobotEnv
from kobuki_maze_rl.robot_env import kobuki_lidar_env
class KobukiMazeEnv(kobuki_lidar_env.KobukiLIDAREnv):
```

After this change the user must fill the *init* function of the class where the action and observations spaces must be filled according to the task, some supported spaces are the Discrete space or the Box space, which is a continous space with limit. An example of how to fill the spaces is shown below:

```
# Using Discrete spaces
self.action_space = spaces.Discrete(n_actions)
self.observation_space = spaces.Discrete(n_observations)

# Using Box spaces
self.action_space = spaces.Box(low=0, high=1, shape=(1,), dtype=np.float32)
self.observation_space = spaces.Box(low=0, high=1, shape=(1,), dtype=np.float32)
```

Also, the user can create different subscribers or publishers in the *init* function of this class whether is to obtain observations or to publish some markers, etc.

After filling the *init* function must fill the following functions:

- _send_action: The function used to send the commands to the robot.
- _get_observation: The function to execute when observations from the enviroment are needed.
- _get_reward: The function that calculates and returns the reward based on the action of the agent.
- _check_if_done: The function to check if the robot sucess finished the task or reached the goal.

- _set_episode_init_params (Optional): The function to set ROS or Gazebo initial parameters for the episode, e.g.: the initial position of the robot, the location of obstacles, etc.

class CustomTaskEnv.**CustomTaskEnv**

Custom Task Env, use this env to implement a task using the robot defined in the CustomRobotEnv

_check_if_done()

Function to check if the episode is done.

If the episode has a success condition then set done as:
self.info['is_success'] = 1.0

_get_observation()

Function to get the observation from the environment.

_get_reward()

Function to get the reward from the environment.

_send_action(*action*)

Function to send an action to the robot

_set_episode_init_params()

Function to set some parameters, like the position of the robot, at the beginning of each episode.

Note: Some examples of how to fill these functions can be seen in the resources repository FRobs_RL Resources.

Note: The user can also create other functions in the class according to their needs.

Finally, the user must change the Gym *register* function, located at the top of the file, so the enviroment gets properly registered and can be called from another Python script. The user must select an unique ID (String), they must change the entry point (the location of the file and name of the class) and optionally select a maximum number of steps per episode. An example of how to change the *register* function is shown below:

```
# Default register
register(
    id='CustomTaskEnv-v0',
    entry_point='frobs_rl.templates.CustomTaskEnv:CustomTaskEnv',
    max_episode_steps=10000,
)

# Changed register
register(
    id='KobukiMazeEnv-v0',
    entry_point='kobuki_maze_rl.task_env.kobuki_maze:KobukiMazeEnv',
    max_episode_steps=100000000000,
)
```

2.3.3 Templates

class CustomRobotEnv.**CustomRobotEnv**

Custom Robot Env, use this for all task envs using the custom robot.

_check_if_done()

Function to check if the episode is done.

If the episode has a success condition then set done as:
self.info['is_success'] = 1.0

_check_subs_and_pubs_connection()

Function to check if the Gazebo and ROS connections are ready

_get_observation()

Function to get the observation from the enviroment.

_get_reward()

Function to get the reward from the enviroment.

_send_action(*action*)

Function to send an action to the robot

_set_episode_init_params()

Function to set some parameters, like the position of the robot, at the begining of each episode.

class CustomTaskEnv.**CustomTaskEnv**

Custom Task Env, use this env to implement a task using the robot defined in the CustomRobotEnv

_check_if_done()

Function to check if the episode is done.

If the episode has a success condition then set done as:
self.info['is_success'] = 1.0

_get_observation()

Function to get the observation from the environment.

_get_reward()

Function to get the reward from the environment.

_send_action(*action*)

Function to send an action to the robot

_set_episode_init_params()

Function to set some parameters, like the position of the robot, at the beginning of each episode.

2.4 Training a model

After creating the enviroment the user can simply import the file and initialize the enviroment with the Gym function *make*. After the enviroment is initialized the user can train any kind of model with any RL algorithm using the Gym env. In the Python script the user also needs to initialize the ROS node and can add any commands as neeeded. An example script is shown below:

```
from kobuki_maze_rl.task_env import kobuki_maze
from frobs_rl.common import ros_gazebo
import gym
import rospy

if __name__ == '__main__':
    # Kill all processes related to previous runs
    ros_node.ros_kill_all_processes()

    # Launch Gazebo
    ros_gazebo.launch_Gazebo(paused=True, gui=False)

    # Start node
    rospy.logwarn("Start")
    rospy.init_node('kobuki_maze_train')

    # Launch the task environment
    env = gym.make('KobukiMazeEnv-v0')
```

Note that the string input of the Gym *make* function is the ID of the env used when registering it.

2.4.1 FRobs_RL Env Wrappers

To change some properties of the enviroments without changing the class env code the user can use Gym Wrappers. These wrappers are used to change properties like the observation or action space of the enviroment without the need to change them directly in the class. This is useful when the user might want to normalize the observation or action space (to change the speed of learning) or when the user want to specify a limit of steps per episode. In **FRobs_RL** the previous wrappers are already included with the names:

- NormalizeActionWrapper
- NormalizeObservWrapper
- TimeLimitWrapper

To use them the user only needs to import them from the library and pass the initialized enviroment. An example where the three wrappers are used is shown below:

```
from kobuki_maze_rl.task_env import kobuki_maze
from frobs_rl.common import ros_gazebo
import gym
import rospy

from frobs_rl.wrappers.NormalizeActionWrapper import NormalizeActionWrapper
from frobs_rl.wrappers.TimeLimitWrapper import TimeLimitWrapper
from frobs_rl.wrappers.NormalizeObservWrapper import NormalizeObservWrapper

if __name__ == '__main__':
    # Kill all processes related to previous runs
    ros_node.ros_kill_all_processes()

    # Launch Gazebo
    ros_gazebo.launch_Gazebo(paused=True, gui=False)
```

(continues on next page)

(continued from previous page)

```
    # Start node
    rospy.logwarn("Start")
    rospy.init_node('kobuki_maze_train')

    # Launch the task environment
    env = gym.make('KobukiMazeEnv-v0')

    #--- Normalize action space
    env = NormalizeActionWrapper(env)

    #--- Normalize observation space
    env = NormalizeObservWrapper(env)

    #--- Set max steps
    env = TimeLimitWrapper(env, max_steps=15000)
    env.reset()
```

2.4.2 Included RL models

In the next step the RL models included from stable-baselines3 in the **FRobs_RL** and how to use them is shown.

2.4.3 Enviroment Wrappers

class NormalizeActionWrapper.**NormalizeActionWrapper**(*env*)

Wrapper to normalize the action space.

Parameters
env – (gym.Env) Gym environment that will be wrapped

rescale_action(*scaled_action*)

Rescale the action from [-1, 1] to [low, high]

Parameters
scaled_action (*np.ndarray*) – The action to rescale.

Returns
The rescaled action.

Return type
np.ndarray

reset()

Reset the environment

step(*action*)

Parameters
action (*floar or int*) – Action taken by the agent

Returns
observation, reward, is the episode over, additional informations

Return type
(np.ndarray, float, bool, dict)

class NormalizeObservWrapper.**NormalizeObservWrapper**(*env*)

Wrapper to normalize the observation space.

Parameters
env – (gym.Env) Gym environment that will be wrapped

reset()

Reset the environment

scale_observation(*observation*)

Scale the observation from [low, high] to [-1, 1].

Parameters
observation (*np.ndarray*) – Observation to scale

Returns
scaled observation

Return type
np.ndarray

step(*action*)

Parameters
action (*float or int*) – Action taken by the agent

Returns
observation, reward, is the episode over, additional informations

Return type
(np.ndarray, float, bool, dict)

class TimeLimitWrapper.**TimeLimitWrapper**(*env*, *max_steps=100*)

Wrapper to limit the number of steps per episode.

Parameters

- **env** – (gym.Env) Gym environment that will be wrapped
- **max_steps** – (int) Max number of steps per episode

reset()

Reset the environment

step(*action*)

Parameters
action (*[float] or int*) – Action taken by the agent

Returns
observation, reward, is the episode over, additional informations

Return type
(np.ndarray, float, bool, dict)

2.5 RL Models

FRobs_RL include an API of all the RL algorithms implemented in stable-baselines3, namely PPO, A2C, DDPG, DQN, SAC and TD3. All of these have an API in which all the algorithm parameters are read from the ROS parameter server, which means that the parameters can be loaded from a YAML file or set/changed through the ROS CLI commands.

In the API of the algorithms is configured the logging system, by default **FRobs_RL** saves the training logs in both CSV and Tensorboard files. In the API is also selected that the models are trained periodically in intervals set by the user in the YAML file, in case the user does not want to save the models they just need to change the associated parameter.

FRobs_RL include YAML templates for the parameters of all algorithms in the config folder. To use any of the algorithms the user just needs to create a config folder inside the ROS package where the enviroment to train is located, change the parameters as needed and call the algorithm from the Python script.

2.5.1 Basic YAML parameters

All of the algorithms YAML file has the next parameters:

```yaml
model_params:

    # Training
    training_steps: 5000      # The number of training steps to perform

    # Save params
    save_freq: 1000 # The step interval to save a new model
    save_prefix: ppo_model # Prefix of models saved
    trained_model_name: trained_model # Name of final model to save
    save_replay_buffer: False

    # Load model params
    load_model: False
    model_name: ppo_model_5000_steps

    # Logging parameters
    log_folder: PPO_1
    log_interval: 4 # The number of episodes between logs
    reset_num_timesteps: False # If true, will reset the number of timesteps to 0 every␣
↪training

    # Use custom policy - Only MlpPolicy is supported (Only used when new model is␣
↪created)
    use_custom_policy: False
    policy_params:
        net_arch: [400, 300] # List of hidden layer sizes
        activation_fn: relu  # relu, tanh, elu or selu
        features_extractor_class: FlattenExtractor # FlattenExtractor,␣
↪BaseFeaturesExtractor or CombinedExtractor
        optimizer_class: Adam # Adam, Adadelta, Adagrad, RMSprop or SGD
```

In these parameters the user can select how many steps to train the model (*training_steps*), how often the models will be saved (*save_freq*), the name of the final model saved after all the training steps (*trained_model_name*), wheter to load a previously saved model and its name (*load_model*, *model_name*), etc.

For the logging parameters the **user needs to remember to** change the log folder parameter every time a train script is executed, as **FRobs_RL** checks if the folder is created and will raise an error if a folder with the same name exists.

Finally, the user can choose is they want a custom neural network (*use_custom_policy*) and its arquitecture/parameters like the optimizer or the activation function.

2.5.2 Algorithm parameters

The algorithm specific parameters are located below the general parameters shown above. The algorithm related parameters include wheter to use action noise, or gSDE folder along the algorithm parameters like the learning rate, batch size, etc. The default algorithm specific parameters for PPO are shown below:

```
# Use SDE
use_sde: False
sde_params:
    sde_sample_freq: -1

# PPO parameters
ppo_params:
    learning_rate: 0.0003
    n_steps: 100    # The number of steps to run for each environment per update (i.e.␣
↪rollout buffer size is n_steps * n_envs where n_envs is number of environment copies␣
↪running in parallel)
    batch_size: 100 # Minibatch size
    n_epochs: 5     # Number of epoch when optimizing the surrogate loss
    gamma: 0.99
    gae_lambda: 0.95
    clip_range: 0.2
    ent_coef: 0.0
    vf_coef: 0.5
    max_grad_norm: 0.5
```

2.5.3 Use of algorithms

After copying the YAML template to the config folder of the user ROS package the user just need to import the algorithm from the library and set the env, the save path where the models will be saved, the log path where all the logs will be saved and the location and name of the YAML paremeter file. After creating the algorithm the user needs to call the *train* method to initiate the learning process which will be the number of steps specific in the YAML file (*training_steps*).

An example using the TD3 algorithm is shown below:

```
from kobuki_maze_rl.task_env import kobuki_maze
from frobs_rl.common import ros_gazebo, ros_node
import gym
import rospy

# Import TD3 algorithm
from frobs_rl.models.td3 import TD3

if __name__ == '__main__':
    # Kill all processes related to previous runs
    ros_node.ros_kill_all_processes()
```

(continues on next page)

(continued from previous page)

```
    # Launch Gazebo
    ros_gazebo.launch_Gazebo(paused=True, gui=False)

    # Start node
    rospy.logwarn("Start")
    rospy.init_node('kobuki_maze_train')

    # Launch the task environment
    env = gym.make('KobukiMazeEnv-v0')

     #--- Normalize action space
    env = NormalizeActionWrapper(env)

    #--- Normalize observation space
    env = NormalizeObservWrapper(env)

    #--- Set max steps
    env = TimeLimitWrapper(env, max_steps=15000)
    env.reset()

     #--- Set the save and log path
    rospack = rospkg.RosPack()
    pkg_path = rospack.get_path("kobuki_maze_rl")


    #-- TD3
    save_path = pkg_path + "/models/td3/"
    log_path = pkg_path + "/logs/td3/"
    model = TD3(env, save_path, log_path, config_file_pkg="kobuki_maze_rl", config_
↪filename="td3.yaml")


    model.train()
    model.save_model()
    model.close_env()

    sys.exit()
```

2.5.4 Custom algorithms

While all the algorithms in *stable-baselines3* are included, the user can use the **FRobs_RL** library with their custom algorithms, althought the user must handle the model saving and logging processes.

It is recommended that if the user wants to implement a new algorithm it is done inheriting the base RL class of stable-baselines3 located in the official website sb3 base RL class or Github repository

2.5.5 Basic algorithm API

Below is the documentation of the basic API of the RL algorithms inherited by every one of the included algorithms.

class basic_model.**BasicModel**(*env*, *save_model_path*, *log_path*, *ns='/'*, *load_trained=False*)

Base class for all the algorithms supported by the frobs_rl library.

Parameters

- **env** – The environment to be used.
- **save_model_path** – The path to save the model.
- **log_path** – The path to save the log.
- **ns** – The namespace of the parameters.
- **load_trained** – Whether or not to load a trained model.

check_env() → bool

Use the stable-baselines check_env method to check the environment.

Returns
True if the environment is correct, False otherwise.

Return type
bool

close_env() → bool

Use the env close method to close the environment.

Returns
True if the environment was closed, False otherwise.

Return type
bool

predict(*observation*, *state=None*, *mask=None*, *deterministic=False*)

Get the current action based on the observation, state or mask

Parameters

- **observation** (*ndarray*) – The enviroment observation
- **state** (*ndarray*) – The previous states of the enviroment, used in recurrent policies.
- **mask** (*ndarray*) – The mask of the last states, used in recurrent policies.
- **deterministic** (*bool*) – Whether or not to return deterministic actions.

Returns
The action to be taken and the next state(for recurrent policies)

Return type
ndarray, ndarray

save_model() → bool

Function to save the model.

Returns
True if the model was saved, False otherwise.

Return type
bool

save_replay_buffer() → bool

Funtion to save the replay buffer, to be used the training must be finished or an error will be raised.

Returns
True if the replay buffer was saved, False otherwise.

Return type
bool

set_model_logger() → bool

Function to set the logger of the model.

Returns
True if the logger was set, False otherwise.

Return type
bool

train() → bool

Function to train the model the number of steps specified in the ROS parameter server. The function will automatically save the model after training.

Returns
True if the training was successful, False otherwise.

Return type
bool

2.6 Using trained models

After the user has trained the model they can use it without the need of an algorithm YAML parameter file and without the processes related to training like the saving of the replay buffer of the calculation of the loss function.

To use a trained model the user just only needs to import the type of the model algorithm from the **FRobs_RL** library and use the *load_trained* function. After the trained model is loaded the user can use the *predict* function to obtain the action based on the observation.

An example where a TD3 trained model is loaded and used in two episodes is shown below.

```
from kobuki_maze_rl.task_env import kobuki_maze
from frobs_rl.common import ros_gazebo, ros_node
import gym
import rospy

# Import TD3 algorithm
from frobs_rl.models.td3 import TD3

if __name__ == '__main__':
    # Kill all processes related to previous runs
    ros_node.ros_kill_all_processes()

    # Launch Gazebo
    ros_gazebo.launch_Gazebo(paused=True, gui=False)

    # Start node
    rospy.logwarn("Start")
```

(continues on next page)

(continued from previous page)

```
    rospy.init_node('kobuki_maze_train')

    # Launch the task environment
    env = gym.make('KobukiMazeEnv-v0')

     #--- Normalize action space
    env = NormalizeActionWrapper(env)

    #--- Normalize observation space
    env = NormalizeObservWrapper(env)

    #--- Set max steps
    env = TimeLimitWrapper(env, max_steps=15000)
    env.reset()

     #--- Set the save and log path
    rospack = rospkg.RosPack()
    pkg_path = rospack.get_path("kobuki_maze_rl")
    save_path = pkg_path + "/models/dynamic/td3/"

    #-- TD3 trained
    model = TD3.load_trained(save_path + "trained_model")


    obs = env.reset()
    episodes = 2
    epi_count = 0
    while epi_count < episodes:
        action, _states = model.predict(obs, deterministic=True)
        obs, _, dones, info = env.step(action)
        if dones:
            epi_count += 1
            rospy.logwarn("Episode: " + str(epi_count))
            obs = env.reset()

    env.close()
    sys.exit()
```

basic_model.BasicModel.**predict**(*self*, *observation*, *state=None*, *mask=None*, *deterministic=False*)

Get the current action based on the observation, state or mask

Parameters

- **observation** (*ndarray*) – The enviroment observation
- **state** (*ndarray*) – The previous states of the enviroment, used in recurrent policies.
- **mask** (*ndarray*) – The mask of the last states, used in recurrent policies.
- **deterministic** (*bool*) – Whether or not to return deterministic actions.

Returns

The action to be taken and the next state(for recurrent policies)

Return type

ndarray, ndarray

2.7 Example enviroment

In this page we show how to create a robotic manipulator enviroment with a reacher task and how to then train it.

2.7.1 Creating a RobotEnv

The first thing that we need to do is to create a class that inherits from the *RobotBasicEnv* class. To create this file we can copy the *CustomRobotEnv* found in the *templates* folder of the library. Within this class we will be defining the general specifications of the robot like is URDF model, its controllers, the positions where it will be spawned in the Gazebo simulator, among other things. Althought the file contains many functions we only need to focus on the *__init__* function and the *_check_subs_and_pubs_connection* function. The *__init__* function is the one that will be called when we create an instance of the class. The *_check_subs_and_pubs_connection* function can be used to check if the subscribers and publishers are connected to the correct topics.

```
class CustomRobotEnv(robot_BasicEnv.RobotBasicEnv):
    def __init__(self):
        # Define class variables and functions
        # Initialize parent class
        super(CustomRobotEnv, self).__init__(launch_gazebo, gazebo_init_paused, gazebo_
↪use_gui)
    def _check_subs_and_pubs_connection(self):
        # Checks ROS and Gazebo connections
        raise NotImplementedError()
```

In our example for the reacher task an ABB IRB120 is used as the robot. The URDF model, the controllers and the spawn position are defined in the *__init__* function. As we are using MoveIt some additional functions are defined in the *ABBIRB120MoveItEnv* class.

```
class ABBIRB120MoveItEnv(robot_BasicEnv.RobotBasicEnv):
    """
    Superclass for all ABB IRB120 environments.
    """

    def __init__(self):
        """
        Initializes a new ABBIRB120Env environment.

        Sensor Topic List:
        * /joint_states : JointState received for the joints of the robot

        Actuators Topic List:
        * MoveIt! : MoveIt! action server is used to send the joint positions to the␣
↪robot.
        """
        rospy.loginfo("Starting ABBIRB120MoveIt Env")
        ros_gazebo.gazebo_unpause_physics()

        """
        Robot model and controllers parameters
        """
        self.model_name_in_gazebo="robot1"
        self.namespace="/robot1"
```

(continues on next page)

(continued from previous page)

```
        pkg_name="abb_irb120"
        urdf_file="irb120.urdf.xacro"
        urdf_xacro_args=['use_pos_ctrls:=true']
        model_pos_x=0.0; model_pos_y=0.0; model_pos_z=0.0
        controller_file="irb120_pos_controller.yaml"
        self.controller_list=["joint_state_controller","arm120_controller"]

        """
        Use parameters to allow multiple robots in the environment.
        """
        if rospy.has_param('/ABB_IRB120_Reacher/current_robot_num'):
            num_robot = int(rospy.get_param("/ABB_IRB120_Reacher/current_robot_num") + 1)
            self.model_name_in_gazebo = "robot" + str(num_robot)
            self.namespace = "/robot" + str(num_robot)

        else:
            num_robot = 0
            self.model_name_in_gazebo = "robot" + str(num_robot)
            self.namespace = "/"

        rospy.set_param('/ABB_IRB120_Reacher/current_robot_num', num_robot)
        model_pos_x = int(num_robot / 4) * 2.0
        model_pos_y = int(num_robot % 4) * 2.0
        model_pos_z = 0.0

        """
        Set if the controllers in "controller_list" will be reset at the beginning of␣
↪each episode, default is False.
        """
        reset_controllers=False

        """
        Set the reset mode of gazebo at the beginning of each episode: 1 is "reset_world
↪", 2 is "reset_simulation". Default is 1.
        """
        reset_mode=1

        """
        Set the step mode of Gazebo. 1 is "using ros services", 2 is "using step␣
↪function of gazebo". Default is 1.
        If using the step mode 2 then set the number of steps of Gazebo to take in each␣
↪episode. Default is 1.
        """
        step_mode=1

        """
        Init the parent class with the corresponding variables.
        """
        super(ABBIRB120MoveItEnv, self).__init__(   launch_gazebo=False, spawn_
↪robot=True,
                    model_name_in_gazebo=self.model_name_in_gazebo, namespace=self.
↪namespace, pkg_name=pkg_name,
```

(continues on next page)

(continued from previous page)

```
                    urdf_file=urdf_file, controller_file=controller_file, controller_
↪list=self.controller_list, urdf_xacro_args=urdf_xacro_args,
                    model_pos_x=model_pos_x, model_pos_y=model_pos_y, model_pos_z=model_
↪pos_z,
                    reset_controllers=reset_controllers, reset_mode=reset_mode, step_
↪mode=step_mode)

        """
        Define publisher or subscribers as needed.
        """
        if self.namespace is not None and self.namespace != '/':
            self.joint_state_topic = self.namespace + "/joint_states"
        else:
            self.joint_state_topic = "/joint_states"

        self.joint_names = [ "joint_1",
                             "joint_2",
                             "joint_3",
                             "joint_4",
                             "joint_5",
                             "joint_6"]

        self.joint_state_sub = rospy.Subscriber(self.joint_state_topic, JointState, self.
↪joint_state_callback)
        self.joint_state = JointState()

        """
        Init MoveIt
        """
        ros_launch.ros_launch_from_pkg("abb_irb120_reacher","moveit_init.launch", args=[
↪"namespace:="+str(self.namespace)])
        rospy.wait_for_service("/move_group/trajectory_execution/set_parameters")
        print(rostopic.get_topic_type("/planning_scene", blocking=True))
        print(rostopic.get_topic_type("/move_group/status", blocking=True))

        """
        If using the _check_subs_and_pubs_connection method, then un-comment the lines␣
↪below.
        """
        self._check_subs_and_pubs_connection()

        #--- Start MoveIt Object
        self.move_abb_object = MoveABB()

        """
        Finished __init__ method
        """
        rospy.loginfo("Finished Init of ABBIRB120MoveIt Env")
        ros_gazebo.gazebo_pause_physics()
```

To check if the topics and services are working, we use the *_check_subs_and_pubs_connection* method.

```
def _check_subs_and_pubs_connection(self):
    """
    Function to check if the gazebo and ros connections are ready
    """
    self._check_joint_states_ready()
    return True


def _check_joint_states_ready(self):
    """
    Function to check if the joint states are received
    """
    ros_gazebo.gazebo_unpause_physics()
    print( rostopic.get_topic_type(self.joint_state_topic, blocking=True))
    rospy.logdebug("Current "+ self.joint_state_topic +" READY=>" + str(self.joint_
↪state))

    return True
```

2.7.2 Creating a TaskEnv

After creating the *RobotEnv* class which contains the general functions for the robot, we create the *TaskEnv* class which contains the specific functions for the task. As a task must always have a robot the *TaskEnv* class inherits from the *RobotEnv* class. Within this class we define the specific functions for the task, like the observation, the reward, the reset, the action space, the observation space, the goal and the termination. To create a file for the task we can copy the template file *CustomTaskEnv* from the *templates* folder.

The template has different functions for the observation, the reward, the reset, the action space, the observation space, the goal and the termination. It will also register the enviroment to the OpenAI Gym environment list so that it can be used as a part of the OpenAI Gym library.

```
register(
    id='CustomTaskEnv-v0',
    entry_point='frobs_rl.templates.CustomTaskEnv:CustomTaskEnv',
    max_episode_steps=10000,
)

class CustomTaskEnv(CustomRobotEnv.CustomRobotEnv):
    """
    Custom Task Env, use this env to implement a task using the robot defined in the␣
↪CustomRobotEnv
    """

    def __init__(self):
        """
        Describe the task.
        """
        rospy.loginfo("Starting Task Env")

        """
        Init super class.
        """
```

(continues on next page)

(continued from previous page)

```
        super(CustomTaskEnv, self).__init__()

        """
        Define the action and observation space.
        """
        # self.action_space = spaces.Discrete(n_actions)
        # self.action_space = spaces.Box(low=0, high=1, shape=(1,), dtype=np.float32)

        # self.observation_space = spaces.Discrete(n_observations)
        # self.observation_space = spaces.Box(low=0, high=1, shape=(1,), dtype=np.
↪float32)

        """
        Define subscribers or publishers as needed.
        """
        # self.pub1 = rospy.Publisher('/robot/controller_manager/command', JointState,␣
↪queue_size=1)
        # self.sub1 = rospy.Subscriber('/robot/joint_states', JointState, self.callback1)

        """
        Finished __init__ method
        """
        rospy.loginfo("Finished Init of Custom Task env")

    #---------------------------------------------------------#
    #   Custom available methods for the CustomTaskEnv      #

    def _set_episode_init_params(self):
        """
        Function to set some parameters, like the position of the robot, at the␣
↪beginning of each episode.
        """
        raise NotImplementedError()

    def _send_action(self, action):
        """
        Function to send an action to the robot
        """
        raise NotImplementedError()


    def _get_observation(self):
        """
        Function to get the observation from the environment.
        """
        raise NotImplementedError()

    def _get_reward(self):
        """
        Function to get the reward from the environment.
        """
        raise NotImplementedError()
```

(continues on next page)

(continued from previous page)

```
    def _check_if_done(self):
        """
        Function to check if the episode is done.

        If the episode has a success condition then set done as:
            self.info['is_success'] = 1.0
        """
        raise NotImplementedError()
```

For our example task, we begin by defining the observation and action space in the *__init__* method, along with initializing the *ABBIRB120MoveItEnv*, which is our *RobotEnv*.

```
class ABBIRB120ReacherEnv(abb_irb120_moveit.ABBIRB120MoveItEnv):

    def __init__(self):
        """
        Reacher enviroment for the ABB IRB120 robot.
        """
        rospy.logwarn("Starting ABBIRB120ReacherEnv Task Env")

        """
        Load YAML param file
        """
        ros_params.ros_load_yaml_from_pkg("abb_irb120_reacher", "reacher_task.yaml", ns=
↪"/")
        self.get_params()

        """
        Define the action and observation space.
        """
        #--- Define the ACTION SPACE
        # Define a continuous space using BOX and defining its limits
        self.action_space = spaces.Box(low=np.array(self.min_joint_values), high=np.
↪array(self.max_joint_values), dtype=np.float32)

        #--- Define the OBSERVATION SPACE
        observations_high_goal_pos_range = np.array(np.array([self.position_goal_max["x
↪"], self.position_goal_max["y"], self.position_goal_max["z"]]))
        observations_low_goal_pos_range  = np.array(np.array([self.position_goal_min["x
↪"], self.position_goal_min["y"], self.position_goal_min["z"]]))

        observations_high_vec_EE_GOAL = np.array([1.0, 1.0, 1.0])
        observations_low_vec_EE_GOAL  = np.array([-1.0, -1.0, -1.0])

        #- With Vector from EE to goal, Goal pos and joint angles
        high = np.concatenate([observations_high_vec_EE_GOAL, observations_high_goal_pos_
↪range, self.max_joint_values, ])
        low  = np.concatenate([observations_low_vec_EE_GOAL,  observations_low_goal_pos_
↪range,  self.min_joint_values, ])

        #--- Observation space
```

(continues on next page)

(continued from previous page)

```
        self.observation_space = spaces.Box(low=low, high=high, dtype=np.float32)

        #-- Action space for sampling
        self.goal_space = spaces.Box(low=observations_low_goal_pos_range,␣
↪high=observations_high_goal_pos_range, dtype=np.float32)

        """
        Define subscribers or publishers as needed.
        """

        self.goal_subs  = rospy.Subscriber("goal_pos", Point, self.goal_callback)
        if self.training:
            ros_node.ros_node_from_pkg("abb_irb120_reacher", "pos_publisher.py", name=
↪"pos_publisher", ns="/")
            rospy.wait_for_service("set_init_point")
            self.set_init_goal_client = rospy.ServiceProxy("set_init_point",␣
↪SetLinkState)

        """
        Init super class.
        """
        super(ABBIRB120ReacherEnv, self).__init__()

        """
        Finished __init__ method
        """
        rospy.logwarn("Finished Init of ABBIRB120ReacherEnv Task Env")
```

After defining the observation and action space we define the methods to execute an action and get the observation.

```
def _send_action(self, action):
    """
    The action are the joint positions
    """
    rospy.logwarn("=== Action: {}".format(action))

    #--- Make actions as deltas
    action = self.joint_values + action
    action = np.clip(action, self.min_joint_values, self.max_joint_values)

    self.movement_result = self.set_trajectory_joints(action)
    if not self.movement_result:
        rospy.logwarn("Movement_result failed with the action of : " + str(action))


def _get_observation(self):
    """
    It returns the position of the EndEffector as observation.
    And the distance from the desired point
    Orientation for the moment is not considered
    """
```

(continues on next page)

(continued from previous page)

```
    #--- Get Current Joint values
    self.joint_values = self.get_joint_angles()

    #--- Get current goal
    current_goal = self.goal

    #--- Get EE position
    ee_pos_v = self.get_ee_pose() # Get a geometry_msgs/PoseStamped msg
    self.ee_pos = np.array([ee_pos_v.pose.position.x, ee_pos_v.pose.position.y, ee_pos_v.
↪pose.position.z])

    #--- Vector to goal
    vec_EE_GOAL = current_goal - self.ee_pos
    vec_EE_GOAL = vec_EE_GOAL / np.linalg.norm(vec_EE_GOAL)

    obs = np.concatenate((
        vec_EE_GOAL,              # Vector from EE to Goal
        current_goal,             # Position of Goal
        self.joint_values         # Current joint angles
        ),
        axis=None
    )

    rospy.logwarn("OBSERVATIONS====>>>>>>>>"+str(obs))
    return obs.copy()
```

Then we proceed by defining the reward method and the method for episode termination.

```
def _get_reward(self):
    """
    Given a success of the execution of the action
    Calculate the reward: binary => 1 for success, 0 for failure
    """

    #--- Get current EE pos
    current_pos = self.ee_pos # If using ARRAY

    #- Init reward
    reward = 0

    #- Check if the EE reached the goal
    done = False
    done = self.calculate_if_done(self.movement_result, self.goal, current_pos)
    if done:
        if self.pos_dynamic is False:
            rospy.logwarn("SUCCESS Reached a Desired Position!")
            self.info['is_success'] = 1.0

        #- Success reward
        reward += self.reached_goal_reward
    else:
        #- Distance from EE to Goal reward
```

(continues on next page)

(continued from previous page)

```
        dist2goal = scipy.spatial.distance.euclidean(current_pos, self.goal)
        reward += - self.mult_dist_reward*dist2goal

        #- Constant reward
        reward += self.step_reward

    self.pub_marker.publish(self.goal_marker)

    #- Check if joints are in limits
    joint_angles = np.array(self.joint_values)
    min_joint_values = np.array(self.min_joint_values)
    max_joint_values = np.array(self.max_joint_values)
    in_limits = np.any(joint_angles<=(min_joint_values+0.0001)) or np.any(joint_angles>
↪=(max_joint_values-0.0001))
    reward += in_limits*self.joint_limits_reward

    rospy.logwarn(">>>REWARD>>>"+str(reward))

    return reward

def _check_if_done(self):
    """
    Check if the EE is close enough to the goal
    """

    #--- Get current EE based on the observation
    current_pos = self.ee_pos # If using ARRAY

    #--- Function used to calculate
    done = self.calculate_if_done(self.movement_result, self.goal, current_pos)
    if done:
        rospy.logdebug("Reached a Desired Position!")

    #--- If the position is dynamic the episode is never done
    if self.pos_dynamic is True:
        done = False

    return done
```

Note: It is important to say that we can add as many methods to the class as we require.

Finally, we add the register function at the beginning of the file to register the enviroment within the OpenAI Gym library.

```
register(
    id='ABBIRB120ReacherEnv-v0',
    entry_point='abb_irb120_reacher.task_env.irb120_reacher:ABBIRB120ReacherEnv',
    max_episode_steps=10000
)
```

2.7.3 Setting the training parameters

After we have defined the Task enviroment we need to create a YAML file with the training parameters. Based on the algorithm that we choose we can copy the template file from the *config* folder of *FRobs_RL* and modify it.

In this example we will use the *TD3* algorithm and we will set the number of training steps to 100000. We will also set a saving frequency of the model, so the trained model is being constantly saved and not only at the end of the training. The saving frequency will be set to 5000 steps. We will set the log folder to *TD3_New*, in this folder the tensorboard logs will be saved.Finally, we can set parameters to use a custom policy architecture and the *TD3* specific parameters. After all the changes are made to the template the file will look as follows:

```
model_params:

    # Training
    training_steps: 100000      # The number of training steps to perform

    # Save params
    save_freq: 5000
    save_prefix: td3_model
    trained_model_name: trained_model
    save_replay_buffer: False

    # Load model params - Used to load a trained model and continue training
    load_model: False
    model_name: trained_model_01_09_2021_10_11_11

    # Logging parameters
    log_folder: TD3_New
    log_interval: 1 # The number of episodes between logs
    reset_num_timesteps: False # If true, will reset the number of timesteps to 0 every␣
↪training

    # Use custom policy - Only used when new model is created
    use_custom_policy: False
    policy_params:
        net_arch: [400, 300] # List of hidden layer sizes
        activation_fn: relu  # relu, tanh, elu or selu
        features_extractor_class: FlattenExtractor # FlattenExtractor,␣
↪BaseFeaturesExtractor or CombinedExtractor
        optimizer_class: Adam # Adam, Adadelta, Adagrad, RMSprop or SGD

    # UseAction noise
    use_action_noise: True # For now only Gaussian noise is supported
    action_noise:
        mean: 0.0
        sigma: 0.01

    # TD3 parameters
    td3_params:
        learning_rate: 0.001
        buffer_size: 1000000
        learning_starts: 100
        batch_size: 100
        tau: 0.005
```

(continues on next page)

(continued from previous page)

```
        gamma: 0.99
        gradient_steps: -1
        policy_delay: 2
        target_policy_noise: 0.2
        target_noise_clip: 0.5
        train_freq:
        freq: 20
        unit: step  # episode or step
```

2.7.4 Training the model

Finally, we will need to create a simple script to train the model which calls the *train* function of the *ABBIRB120ReacherEnv* class and uses the *TD3* algorithm with the parameters defined in the YAML file.

In the following script, the Gazebo simulator is launched, then the environment is created and some wrappers are applied to the environment. The environment is then trained using the *TD3* algorithm with the parameters defined in the YAML file. And finally, the trained model is saved.

```
if __name__ == '__main__':

# Kill all processes related to previous runs
ros_node.ros_kill_all_processes()

# Launch Gazebo
ros_gazebo.launch_Gazebo(paused=True, gui=False)

# Start node
rospy.logwarn("Start")
rospy.init_node('train_irb120_reacher')

# Launch the task environment
env = gym.make('ABBIRB120ReacherEnv-v0')

#--- Normalize action space
env = NormalizeActionWrapper(env)

#--- Normalize observation space
env = NormalizeObservWrapper(env)

#--- Set max steps
env = TimeLimitWrapper(env, max_steps=100)
env.reset()

#--- Set the save and log path
rospack = rospkg.RosPack()
pkg_path = rospack.get_path("abb_irb120_reacher")

#-- TD3
save_path = pkg_path + "/models/static_reacher/td3/" # Path where the model will be saved
log_path = pkg_path + "/logs/aux/td3/"               # Path where the logs will be saved
model = TD3(env, save_path, log_path, config_file_pkg="abb_irb120_reacher", config_
↪filename="td3_aux.yaml")
```

(continues on next page)

(continued from previous page)

```
#-- Train the model and save it
model.train()
model.save_model()
model.close_env()

sys.exit()
```

2.8 API

API Docs

2.8.1 ros_controllers

Functions related to the handling of the start, stop, reset, load, unload, etc, of ROS controllers.

ros_controllers.**load_controller_srv**(*controller_name*, *ns=None*, *max_retries=5*) → bool

Function to load a controller on the namespace.

Parameters

- **controller_name** (*string*) – name of the controller to load
- **ns** (*string*) – namespace
- **max_retries** (*int*) – number of retries to load the controller.

Returns
true if the controller is loaded.

Return type
bool

ros_controllers.**load_controller_list_srv**(*controller_list*, *ns=None*, *max_retries=5*) → None

Function to load a list of controllers on the namespace.

Parameters

- **controller_list** (*list of strings*) – list of controllers to load
- **ns** (*string*) – namespace
- **max_retries** (*int*) – number of retries to load the controller.

ros_controllers.**unload_controller_srv**(*controller_name*, *ns=None*, *max_retries=5*) → bool

Function to unload a controller on the namespace.

Parameters

- **controller_name** (*string*) – name of the controller to unload
- **ns** (*string*) – namespace
- **max_retries** (*int*) – number of retries to unload the controller.

Returns
true if the controller is unloaded.

Return type
bool

ros_controllers.**unload_controller_list_srv**(*controller_list*, *ns=None*, *max_retries=5*) → None

Function to unload a list of controllers on the namespace.

Parameters

- **controller_list** (*list of strings*) – list of controllers to unload
- **ns** (*string*) – namespace
- **max_retries** (*int*) – number of retries to unload the controller.

ros_controllers.**switch_controllers_srv**(*start_controllers*, *stop_controllers*, *ns=None*, *strictness=1*, *start_asap=False*, *timeout=3.0*, *max_retries=5*) → bool

Function to switch controllers on the namespace.

Parameters

- **start_controllers** (*list of strings*) – list of controllers to start
- **stop_controllers** (*list of strings*) – list of controllers to stop
- **ns** (*string*) – namespace
- **strictness** (*int*) – strictness of the controller manager: BEST_EFFORT or STRICT (1 and 2, respectively).
- **start_asap** (*bool*) – start the controllers as soon as their hardware dependencies are ready, will wait for all interfaces to be ready otherwise.
- **timeout** (*float*) – the timeout in seconds before aborting pending controllers. Zero for infinite.
- **max_retries** (*int*) – number of retries to switch the controller.

Returns
true if the operation is successful.

Return type
bool

ros_controllers.**start_controllers_srv**(*start_controllers*, *ns=None*, *strictness=1*, *start_asap=False*, *timeout=3.0*) → bool

Function to start controllers on the namespace.

Parameters

- **start_controllers** (*list of strings*) – list of controllers to start
- **ns** (*string*) – namespace
- **strictness** (*int*) – strictness of the controller manager: BEST_EFFORT or STRICT (1 and 2, respectively).
- **start_asap** (*bool*) – start the controllers as soon as their hardware dependencies are ready, will wait for all interfaces to be ready otherwise.
- **timeout** (*float*) – the timeout in seconds before aborting pending controllers. Zero for infinite.

Returns
true if the operation is successful.

Return type
bool

ros_controllers.**stop_controllers_srv**(*stop_controllers*, *ns=None*, *strictness=1*, *start_asap=False*, *timeout=3.0*) → bool

Function to start controllers on the namespace.

Parameters

- **stop_controllers** (*list of strings*) – list of controllers to stop
- **ns** (*string*) – namespace
- **strictness** (*int*) – strictness of the controller manager: BEST_EFFORT or STRICT (1 and 2, respectively).
- **start_asap** (*bool*) – start the controllers as soon as their hardware dependencies are ready, will wait for all interfaces to be ready otherwise.
- **timeout** (*float*) – the timeout in seconds before aborting pending controllers. Zero for infinite.

Returns
true if the operation is successful.

Return type
bool

ros_controllers.**reset_controllers_srv**(*reset_controllers*, *max_retries=10*, *ns=None*, *strictness=1*, *start_asap=False*, *timeout=3.0*) → bool

Function to reset controllers on the namespace.

Parameters

- **reset_controllers** (*list of strings*) – list of controllers to reset
- **max_retries** (*int*) – number of times to retry resetting a controller before giving up.
- **ns** (*string*) – namespace
- **strictness** (*int*) – strictness of the controller manager: BEST_EFFORT or STRICT (1 and 2, respectively).
- **start_asap** (*bool*) – start the controllers as soon as their hardware dependencies are ready, will wait for all interfaces to be ready otherwise.
- **timeout** (*float*) – the timeout in seconds before aborting pending controllers. Zero for infinite.

Returns
true if the operation is successful.

Return type
bool

ros_controllers.**spawn_controllers_srv**(*spawn_controllers*, *ns=None*, *strictness=1*, *start_asap=False*, *timeout=3.0*) → bool

Function to spawn controllers on the namespace.

Parameters

- **spawn_controllers** (*list of strings*) – list of controllers to spawn
- **ns** (*string*) – namespace

- **strictness** (*int*) – strictness of the controller manager: BEST_EFFORT or STRICT (1 and 2, respectively).
- **start_asap** (*bool*) – start the controllers as soon as their hardware dependencies are ready, will wait for all interfaces to be ready otherwise.
- **timeout** (*float*) – the timeout in seconds before aborting pending controllers. Zero for infinite.

Returns
true if the operation is successful.

Return type
bool

ros_controllers.**kill_controllers_srv**(*kill_controllers*, *ns=None*, *strictness=1*, *start_asap=False*, *timeout=3.0*) → bool

Function to kill controllers on the namespace.

Parameters

- **kill_controllers** (*list of strings*) – list of controllers to kill
- **ns** (*string*) – namespace
- **strictness** (*int*) – strictness of the controller manager: BEST_EFFORT or STRICT (1 and 2, respectively).
- **start_asap** (*bool*) – start the controllers as soon as their hardware dependencies are ready, will wait for all interfaces to be ready otherwise.
- **timeout** (*float*) – the timeout in seconds before aborting pending controllers. Zero for infinite.

Returns
true if the operation is successful.

Return type
bool

2.8.2 ros_gazebo

Functions related to launch of the Gazebo simulation, along functions to change simulator parameters and handling of simulator objects.

ros_gazebo.**launch_Gazebo**(*paused=False*, *use_sim_time=True*, *gui=True*, *recording=False*, *debug=False*, *verbose=False*, *output='screen'*, *custom_world_path=None*, *custom_world_pkg=None*, *custom_world_name=None*, *respawn_gazebo=False*, *pub_clock_frequency=100*, *server_required=False*, *gui_required=False*, *launch_new_term=True*) → bool

Launch Gazebo using the ROS network.

Parameters

- **paused** (*bool*) – if True, init gzserver paused.
- **use_sim_time** (*bool*) – if True, use the simulation time.
- **gui** (*bool*) – if True, launch Gazebo with the GUI (gzclient).
- **recording** (*bool*) – if True, record the data from gazebo.

- **debug** (*bool*) – if True, use the debug configuration.
- **verbose** (*bool*) – if True, print debug information.
- **output** (*str [screen, log]*) – choose the output method for gazebo (screen, log).
- **custom_world_path** (*str*) – if not None, use this world path.
- **custom_world_pkg** (*str*) – if the custom_world_path is None, use a world file from this package, specified in custom_world_name.
- **custom_world_name** (*str*) – if the custom_world_path is None, use the world file with this name from the custom_world_pkg.
- **respawn_gazebo** (*bool*) – if True, gazebo will be respawned if it is killed, default is False.
- **pub_clock_frequency** (*int*) – the frequency of the clock publisher (in Hz)
- **server_required** (*bool*) – if True, the launch file will wait until gzserver is running.
- **gui_required** (*bool*) – if True, the launch file will wait until gzclient is running.
- **launch_new_term** (*bool*) – Launch the gazebo node in a new terminal (Xterm).

Returns
True if the launch was successful, False otherwise.

Return type
bool

ros_gazebo.**close_Gazebo**() → bool

Function to close gazebo if its running.

Returns
True if gazebo was closed, False otherwise.

Return type
bool

ros_gazebo.**gazebo_set_max_update_rate**(*max_update_rate*) → bool

Function to set the max update rate for gazebo in real time factor: 1 is real time, 10 is 10 times real time.

Parameters
max_update_rate (*float*) – the max update rate for gazebo in real time factor.

Returns
True if the command was sent and False otherwise.

Return type
bool

ros_gazebo.**gazebo_get_max_update_rate**() → float

Function to get the current max update rate.

Returns
the max update rate.

Return type
float

ros_gazebo.**gazebo_set_time_step**(*new_time_step*) → bool

Function to set the time step for gazebo.

Parameters
new_time_step (*float*) – the new time step.

Returns
True if the command was sent and False otherwise.

Return type
bool

ros_gazebo.**gazebo_get_time_step**() → float

Function to get the current time step for gazebo.

Returns
the time step.

Return type
float

ros_gazebo.**gazebo_set_gravity**(*x*, *y*, *z*) → bool

Function to set the gravity for gazebo.

Parameters

- **x** (*float*) – the x component of the gravity vector.
- **y** (*float*) – the y component of the gravity vector.
- **z** (*float*) – the z component of the gravity vector.

Returns
True if the command was sent and False otherwise.

Return type
bool

ros_gazebo.**gazebo_get_gravity**() → geometry_msgs.msg.Vector3

Function to get the current gravity vector for gazebo.

Returns
the gravity vector.

Return type
Vector3

ros_gazebo.**gazebo_set_ODE_physics**(*auto_disable_bodies*, *sor_pgs_precon_iters*, *sor_pgs_iters*, *sor_pgs_w*, *sor_pgs_rms_error_tol*, *contact_surface_layer*, *contact_max_correcting_vel*, *cfm*, *erp*, *max_contacts*) → bool

Function to set the ODE physics for gazebo.

Returns
True if the command was sent and False otherwise.

ros_gazebo.**gazebo_get_ODE_physics**() → gazebo_msgs.msg.ODEPhysics

Function to get the current ODE physics for gazebo.

Returns
the ODE physics.

Return type
ODEPhysics

ros_gazebo.**gazebo_set_default_properties**() → bool

Function to set the default gazebo properties.

Returns
True if the command was sent and False otherwise.

ros_gazebo.**gazebo_reset_sim**(*retries=5*) → bool

Function to reset the simulation, which reset models to original poses AND reset the simulation time.

Parameters
retries (*int*) – The number of times to retry the service call.

Returns
True if the command was sent and False otherwise.

Return type
bool

ros_gazebo.**gazebo_reset_world**(*retries=5*) → bool

Function to reset the world, which reset models to original poses WITHOUT resetting the simulation time.

Parameters
retries (*int*) – The number of times to retry the service call.

Returns
True if the command was sent and False otherwise.

Return type
bool

ros_gazebo.**gazebo_pause_physics**(*retries=5*) → bool

Function to pause the physics in the simulation.

Parameters
retries (*int*) – The number of times to retry the service call.

Returns
True if the command was sent and False otherwise.

Return type
bool

ros_gazebo.**gazebo_unpause_physics**(*retries=5*) → bool

Function to unpause the physics in the simulation.

Parameters
retries (*int*) – The number of times to retry the service call.

Returns
True if the command was sent and False otherwise.

Return type
bool

ros_gazebo.**gazebo_step_physics**(*steps=1*) → bool

Function to step the physics in the simulation.

Parameters
steps (*int*) – The number of times to step the simulation.

Returns
True if the command was sent and False otherwise.

Return type
bool

ros_gazebo.**gazebo_delete_model**(*model_name*) → bool

Function to delete a model from the simulation.

Parameters
model_name (*str*) – The name of the model to delete.

Returns
True if the command was sent and False otherwise.

Return type
bool

ros_gazebo.**gazebo_spawn_urdf_path**(*model_path*, *model_name='robot1'*, *robot_namespace='/'*, *reference_frame='world'*, *pos_x=0.0*, *pos_y=0.0*, *pos_z=0.0*, *ori_x=0.0*, *ori_y=0.0*, *ori_z=0.0*, *ori_w=1.0*) → bool

Function to spawn a model from a URDF file.

Parameters

- **model_path** (*str*) – The path to the URDF file.
- **model_name** (*str*) – Name of model to spawn
- **robot_namespace** (*str*) – change ROS namespace of gazebo-plugins.
- **reference_frame** (*str*) – Name of the model/body where initial pose is defined. If left empty or specified as "world", gazebo world frame is used
- **pos_x** (*float*) – x position of model in model's reference frame
- **pos_y** (*float*) – y position of model in model's reference frame
- **pos_z** (*float*) – z position of model in model's reference frame
- **ori_x** (*float*) – X part of Quaternion of model orientation in model's reference frame.
- **ori_y** (*float*) – Y part of Quaternion of model orientation in model's reference frame.
- **ori_z** (*float*) – Z part of Quaternion of model orientation in model's reference frame.
- **ori_w** (*float*) – W part of Quaternion of model orientation in model's reference frame.

Returns
[True if the command was sent and False otherwise, Status message].

Return type
[bool, str]

ros_gazebo.**gazebo_spawn_urdf_pkg**(*pkg_name*, *file_name*, *file_folder='/urdf'*, *model_name='robot1'*, *robot_namespace='/'*, *reference_frame='world'*, *pos_x=0.0*, *pos_y=0.0*, *pos_z=0.0*, *ori_x=0.0*, *ori_y=0.0*, *ori_z=0.0*, *ori_w=1.0*) → bool

Function to spawn a model from a URDF file.

Parameters

- **pkg_name** (*str*) – Name of the package to import the URDF file from.
- **file_name** (*str*) – Name of the URDF file to import.
- **file_folder** (*str*) – Folder where the URDF file is located. Defaults to "/urdf".
- **model_name** (*str*) – Name of model to spawn
- **robot_namespace** (*str*) – change ROS namespace of gazebo-plugins.
- **reference_frame** (*str*) – Name of the model/body where initial pose is defined. If left empty or specified as "world", gazebo world frame is used
- **pos_x** (*float*) – x position of model in model's reference frame

- **pos_y** (*float*) – y position of model in model's reference frame
- **pos_z** (*float*) – z position of model in model's reference frame
- **ori_x** (*float*) – X part of Quaternion of model orientation in model's reference frame.
- **ori_y** (*float*) – Y part of Quaternion of model orientation in model's reference frame.
- **ori_z** (*float*) – Z part of Quaternion of model orientation in model's reference frame.
- **ori_w** (*float*) – W part of Quaternion of model orientation in model's reference frame.

Returns
[True if the command was sent and False otherwise, Status message].

Return type
[bool, str]

ros_gazebo.**gazebo_spawn_urdf_string**(*model_string*, *model_name='robot1'*, *robot_namespace='/'*, *reference_frame='world'*, *pos_x=0.0*, *pos_y=0.0*, *pos_z=0.0*, *ori_x=0.0*, *ori_y=0.0*, *ori_z=0.0*, *ori_w=1.0*) → bool

Function to spawn a model from a URDF file.

Parameters

- **model_string** (*str*) – URDF string to import.
- **model_name** (*str*) – Name of model to spawn
- **robot_namespace** (*str*) – change ROS namespace of gazebo-plugins.
- **reference_frame** (*str*) – Name of the model/body where initial pose is defined. If left empty or specified as "world", gazebo world frame is used.
- **pos_x** (*float*) – x position of model in model's reference frame
- **pos_y** (*float*) – y position of model in model's reference frame
- **pos_z** (*float*) – z position of model in model's reference frame
- **ori_x** (*float*) – X part of Quaternion of model orientation in model's reference frame.
- **ori_y** (*float*) – Y part of Quaternion of model orientation in model's reference frame.
- **ori_z** (*float*) – Z part of Quaternion of model orientation in model's reference frame.
- **ori_w** (*float*) – W part of Quaternion of model orientation in model's reference frame.

Returns
[True if the command was sent and False otherwise, Status message].

Return type
[bool, str]

ros_gazebo.**gazebo_spawn_urdf_param**(*param_name*, *model_name='robot1'*, *robot_namespace='/'*, *reference_frame='world'*, *pos_x=0.0*, *pos_y=0.0*, *pos_z=0.0*, *ori_x=0.0*, *ori_y=0.0*, *ori_z=0.0*, *ori_w=1.0*) → bool

Function to spawn a model from a URDF in the param server.

Parameters

- **param_name** (*str*) – Name of model to spawn
- **model_name** (*str*) – Name of model to spawn
- **robot_namespace** (*str*) – change ROS namespace of gazebo-plugins.

- **reference_frame** (*str*) – Name of the model/body where initial pose is defined. If left empty or specified as "world", gazebo world frame is used.
- **pos_x** (*float*) – x position of model in model's reference frame
- **pos_y** (*float*) – y position of model in model's reference frame
- **pos_z** (*float*) – z position of model in model's reference frame
- **ori_x** (*float*) – X part of Quaternion of model orientation in model's reference frame.
- **ori_y** (*float*) – Y part of Quaternion of model orientation in model's reference frame.
- **ori_z** (*float*) – Z part of Quaternion of model orientation in model's reference frame.
- **ori_w** (*float*) – W part of Quaternion of model orientation in model's reference frame.

Returns
[True if the command was sent and False otherwise, Status message].

Return type
[bool, str]

ros_gazebo.**gazebo_spawn_sdf_path**(*model_path*, *model_name='robot1'*, *robot_namespace='/'*, *reference_frame='world'*, *pos_x=0.0*, *pos_y=0.0*, *pos_z=0.0*, *ori_x=0.0*, *ori_y=0.0*, *ori_z=0.0*, *ori_w=1.0*) → bool

Function to spawn a model from a SDF file.

Parameters

- **model_path** (*str*) – The path to the SDF file.
- **model_name** (*str*) – Name of model to spawn
- **robot_namespace** (*str*) – change ROS namespace of gazebo-plugins.
- **reference_frame** (*str*) – Name of the model/body where initial pose is defined. If left empty or specified as "world", gazebo world frame is used
- **pos_x** (*float*) – x position of model in model's reference frame
- **pos_y** (*float*) – y position of model in model's reference frame
- **pos_z** (*float*) – z position of model in model's reference frame
- **ori_x** (*float*) – X part of Quaternion of model orientation in model's reference frame.
- **ori_y** (*float*) – Y part of Quaternion of model orientation in model's reference frame.
- **ori_z** (*float*) – Z part of Quaternion of model orientation in model's reference frame.
- **ori_w** (*float*) – W part of Quaternion of model orientation in model's reference frame.

Returns
True if the command was sent and False otherwise.

Return type
bool

ros_gazebo.**gazebo_spawn_sdf_pkg**(*pkg_name*, *file_name*, *file_folder='/sdf'*, *model_name='robot1'*, *robot_namespace='/'*, *reference_frame='world'*, *pos_x=0.0*, *pos_y=0.0*, *pos_z=0.0*, *ori_x=0.0*, *ori_y=0.0*, *ori_z=0.0*, *ori_w=1.0*) → bool

Function to spawn a model from a SDF file.

Parameters

- **pkg_name** (*str*) – Name of the package to import the SDF file from.

- **file_name** (*str*) – Name of the SDF file to import.
- **file_folder** (*str*) – Folder where the SDF file is located. Default is "/sdf".
- **model_name** (*str*) – Name of model to spawn
- **robot_namespace** (*str*) – change ROS namespace of gazebo-plugins.
- **reference_frame** (*str*) – Name of the model/body where initial pose is defined. If left empty or specified as "world", gazebo world frame is used
- **pos_x** (*float*) – x position of model in model's reference frame
- **pos_y** (*float*) – y position of model in model's reference frame
- **pos_z** (*float*) – z position of model in model's reference frame
- **ori_x** (*float*) – X part of Quaternion of model orientation in model's reference frame.
- **ori_y** (*float*) – Y part of Quaternion of model orientation in model's reference frame.
- **ori_z** (*float*) – Z part of Quaternion of model orientation in model's reference frame.
- **ori_w** (*float*) – W part of Quaternion of model orientation in model's reference frame.

Returns
True if the command was sent and False otherwise.

Return type
bool

ros_gazebo.**gazebo_spawn_sdf_string**(*model_string*, *model_name='robot1'*, *robot_namespace='/'*, *reference_frame='world'*, *pos_x=0.0*, *pos_y=0.0*, *pos_z=0.0*, *ori_x=0.0*, *ori_y=0.0*, *ori_z=0.0*, *ori_w=1.0*) → bool

Function to spawn a model from a SDF file.

Parameters

- **model_string** (*str*) – SDF string to import.
- **model_name** (*str*) – Name of model to spawn
- **robot_namespace** (*str*) – change ROS namespace of gazebo-plugins.
- **reference_frame** (*str*) – Name of the model/body where initial pose is defined. If left empty or specified as "world", gazebo world frame is used.
- **pos_x** (*float*) – x position of model in model's reference frame
- **pos_y** (*float*) – y position of model in model's reference frame
- **pos_z** (*float*) – z position of model in model's reference frame
- **ori_x** (*float*) – X part of Quaternion of model orientation in model's reference frame.
- **ori_y** (*float*) – Y part of Quaternion of model orientation in model's reference frame.
- **ori_z** (*float*) – Z part of Quaternion of model orientation in model's reference frame.
- **ori_w** (*float*) – W part of Quaternion of model orientation in model's reference frame.

Returns
True if the command was sent and False otherwise.

Return type
bool

ros_gazebo.**gazebo_spawn_sdf_param**(*param_name*, *model_name='robot1'*, *robot_namespace='/'*, *reference_frame='world'*, *pos_x=0.0*, *pos_y=0.0*, *pos_z=0.0*, *ori_x=0.0*, *ori_y=0.0*, *ori_z=0.0*, *ori_w=1.0*) → bool

Function to spawn a model from a SDF in the param server.

Parameters

- **param_name** (*str*) – Name of model to spawn
- **model_name** (*str*) – Name of model to spawn
- **robot_namespace** (*str*) – change ROS namespace of gazebo-plugins.
- **reference_frame** (*str*) – Name of the model/body where initial pose is defined. If left empty or specified as "world", gazebo world frame is used.
- **pos_x** (*float*) – x position of model in model's reference frame
- **pos_y** (*float*) – y position of model in model's reference frame
- **pos_z** (*float*) – z position of model in model's reference frame
- **ori_x** (*float*) – X part of Quaternion of model orientation in model's reference frame.
- **ori_y** (*float*) – Y part of Quaternion of model orientation in model's reference frame.
- **ori_z** (*float*) – Z part of Quaternion of model orientation in model's reference frame.
- **ori_w** (*float*) – W part of Quaternion of model orientation in model's reference frame.

Returns
True if the command was sent and False otherwise.

Return type
bool

ros_gazebo.**gazebo_get_model_state**(*model_name*, *relative_entity_name='world'*)

Function to get the state of a model.

Parameters

- **model_name** (*str*) – Name of model to get the state of.
- **relative_entity_name** (*str*) – Return pose and twist relative to this entity (an entity can be a model, body, or geom).

Returns
The header of the message, the pose of the model, the twist of the model, and success.

ros_gazebo.**gazebo_set_model_state**(*model_name*, *ref_frame='world'*, *pos_x=0.0*, *pos_y=0.0*, *pos_z=0.0*, *ori_x=0.0*, *ori_y=0.0*, *ori_z=0.0*, *ori_w=0.0*, *lin_vel_x=0.0*, *lin_vel_y=0.0*, *lin_vel_z=0.0*, *ang_vel_x=0.0*, *ang_vel_y=0.0*, *ang_vel_z=0.0*, *sleep_time=0.05*) → bool

Function to set the model name in gazebo.

Parameters

- **model_name** (*str*) – Name of model to set.
- **ref_frame** (*str*) – Reference frame of model.
- **pos_x** (*float*) – x position of model in model's reference frame
- **pos_y** (*float*) – y position of model in model's reference frame
- **pos_z** (*float*) – z position of model in model's reference frame

- **ori_x** (*float*) – X part of Quaternion of model orientation in model's reference frame.
- **ori_y** (*float*) – Y part of Quaternion of model orientation in model's reference frame.
- **ori_z** (*float*) – Z part of Quaternion of model orientation in model's reference frame.
- **ori_w** (*float*) – W part of Quaternion of model orientation in model's reference frame.
- **lin_vel_x** (*float*) – X part of linear velocity of model in model's reference frame.
- **lin_vel_y** (*float*) – Y part of linear velocity of model in model's reference frame.
- **lin_vel_z** (*float*) – Z part of linear velocity of model in model's reference frame.
- **ang_vel_x** (*float*) – X part of angular velocity of model in model's reference frame.
- **ang_vel_y** (*float*) – Y part of angular velocity of model in model's reference frame.
- **ang_vel_z** (*float*) – Z part of angular velocity of model in model's reference frame.
- **sleep_time** (*float*) – Time to sleep bewteen sending request and getting response.

Returns
True if the command was sent and False otherwise.

Return type
bool

2.8.3 ros_launch

Functions related to the handling of the ROS launch files.

ros_launch.**ros_launch_from_pkg**(*pkg_name*, *launch_file*, *args=None*, *launch_new_term=True*) → bool

Function to execute a roslaunch from package with args. :param pkg_name: Name of the package where the launch file is located. :type pkg_name: str

Parameters

- **launch_file** (*str*) – Name of the launch file.
- **args** (*list of str*) – Args to pass to the launch file.
- **launch_new_term** (*bool*) – Launch the process in a new terminal (Xterm).

Returns
True if the launch file was executed.

Return type
bool

ros_launch.**ros_launch_from_path**(*launch_file_path*, *args=None*, *launch_new_term=True*) → bool

Function to execute a roslaunch from a path with args. :param launch_file_path: Path of the launch file. :type launch_file_path: str

Parameters

- **args** (*list str*) – Args to pass to the launch file.
- **launch_new_term** (*bool*) – Launch the process in a new terminal (Xterm).

Returns
True if the launch file was executed.

Return type
bool

ros_launch.**ros_kill_launch_process**() → bool

Function to kill all roslaunch processes.

Returns
True if the roslaunch processes were killed.

Return type
bool

2.8.4 ros_node

Functions related to the handling of ROS nodes.

ros_node.**ros_node_from_pkg**(*pkg_name*, *node_name*, *launch_master=False*, *launch_master_term=True*, *launch_new_term=True*, *name=None*, *ns='/'*, *output='log'*) → bool

Function to launch a ROS node from a package.

Parameters

- **pkg_name** (*str*) – Name of the package to launch the node from.
- **node_name** (*str*) – Name of the node to launch.
- **launch_master** (*bool*) – If ROSMASTER is not running launch it.
- **launch_master_term** (*bool*) – If launch ROSMASTER do it in an external terminal.
- **launch_new_term** (*bool*) – Launch the process in a new terminal (Xterm).
- **name** (*str*) – Name to give the node to be launched.
- **ns** (*str*) – Namespace to give the node to be launched.
- **output** (*str*) – log, screen, or None.

Returns
True if the node was launched, False otherwise.

Return type
bool

ros_node.**ros_kill_node**(*node_name*) → bool

Function to kill a ROS node.

Parameters
node_name (*str*) – Name of the node to kill.

Returns
True if the node was killed, False otherwise.

Return type
bool

ros_node.**ros_kill_all_nodes**() → bool

Function to kill all running ROS nodes.

Returns
True if all nodes were killed, False otherwise.

Return type
bool

ros_node.**ros_kill_master**() → bool

Function to kill the ROS master.

Returns
True if the master was killed, False otherwise.

Return type
bool

ros_node.**ros_kill_all_processes**() → bool

Function to kill all running ROS related processes.

Returns
True if all processes were killed, False otherwise.

Return type
bool

2.8.5 ros_params

Functions to load parameters to the ROS parameter server.

ros_params.**ros_load_yaml_from_pkg**(*pkg_name*, *file_name*, *ns='/'*) → bool

Fetch a YAML file from a package and load it into the ROS Parameter Server.

Parameters

- **pkg_name** (*str*) – name of package.
- **file_name** (*str*) – name of file.
- **ns** (*str*) – namespace to load parameters into.

Returns
True if file was loaded, false otherwise.

Return type
bool

ros_params.**ros_load_yaml_from_path**(*file_path*, *ns='/'*) → bool

Fetch a YAML file from a path and load it into the ROS Parameter Server.

Parameters

- **file_path** (*str*) – path to file.
- **ns** (*str*) – namespace to load parameters into.

Returns
True if file was loaded, false otherwise.

Return type
bool

2.8.6 ros_spawn

Functions to spawn the robots inside the Gazebo simulator.

ros_spawn.**init_robot_state_pub**(*namespace='/'*, *max_pub_freq=None*, *launch_new_term=False*) → bool

Funtion to initialize the robot state publisher.

Parameters

- **namespace** (*str*) – Namespace of the robot.
- **max_pub_freq** (*float*) – Maximum frequency of the publisher.
- **launch_new_term** (*bool*) – Launch the process in a new terminal (Xterm).

Returns

Return true if the publisher was initialized.

Return type

bool

ros_spawn.**spawn_model_in_gazebo**(*pkg_name*, *model_urdf_file*, *controllers_file*, *controllers_list=[]*, *model_urdf_folder='/urdf'*, *ns='/'*, *args_xacro=None*, *max_pub_freq=None*, *rob_st_term=False*, *gazebo_name='robot1'*, *gaz_ref_frame='world'*, *pos_x=0.0*, *pos_y=0.0*, *pos_z=0.0*, *ori_w=0.0*, *ori_x=0.0*, *ori_y=0.0*, *ori_z=0.0*)

Function to spawn a model in gazebo.

Parameters

- **pkg_name** (*str*) – Package name of the model.
- **model_urdf_file** (*str*) – Name of the model urdf file.
- **controllers_file** (*str*) – Name of the controllers file. If None then no controllers will be loaded.
- **controllers_list** (*list*) – List of the controllers to be loaded.
- **model_urdf_folder** (*str*) – Folder where the model urdf file is located. Default is "/urdf".
- **ns** (*str*) – Namespace of the model. Default is "/".
- **args_xacro** (*list*) – Arguments to be passed to xacro.
- **max_pub_freq** (*float*) – Maximum frequency of the robot state publisher.
- **rob_st_term** (*bool*) – Launch the robot state publisher in a new terminal (Xterm).
- **gazebo_name** (*str*) – Name of the gazebo model.
- **gaz_ref_frame** (*str*) – Reference frame of the gazebo model.
- **pos_x** (*float*) – X position of the gazebo model.
- **pos_y** (*float*) – Y position of the gazebo model.
- **pos_z** (*float*) – Z position of the gazebo model.
- **ori_w** (*float*) – W orientation of the gazebo model.
- **ori_x** (*float*) – X orientation of the gazebo model.
- **ori_y** (*float*) – Y orientation of the gazebo model.
- **ori_z** (*float*) – Z orientation of the gazebo model.

Returns
Return true if the model was spawned.

Return type
bool

2.8.7 ros_urdf

Functions to parse and load to the ROS parameter server the URDF files.

ros_urdf.**urdf_load_from_pkg**(*pkg_name*, *model_name*, *param_name*, *folder='/urdf'*, *ns=None*, *args_xacro=None*) → bool

Function to load a URDF from a ROS package to the parameter server.

Parameters

- **pkg_name** (*str*) – The ROS package name.
- **model_name** (*str*) – The model file name.
- **param_name** (*str*) – The parameter name.
- **folder** (*str*) – The folder where the model is located. Default: "/urdf"
- **ns** (*str*) – The namespace of the parameter.
- **args_xacro** (*list of str.*) – The xacro arguments in a list. Eg: ['arg1:=True','arg2:=10.0']

Returns
True if the URDF was loaded, False otherwise.

Return type
bool

ros_urdf.**urdf_load_from_path**(*model_path*, *param_name*, *ns=None*, *args_xacro=None*) → bool

Function to load a URDF from a file to the parameter server.

Parameters

- **model_path** (*str*) – The model file path.
- **param_name** (*str*) – The parameter name.
- **ns** (*str*) – The namespace of the parameter.
- **args_xacro** (*list of str.*) – The xacro arguments in a list. Eg: ['arg1:=True','arg2:=10.0']

Returns
True if the URDF was loaded, False otherwise.

Return type
bool

ros_urdf.**urdf_parse_from_pkg**(*pkg_name*, *model_name*, *folder='/urdf'*, *args_xacro=None*) → str

Function to parse a URDF from a ROS package and return the URDF string.

Parameters

- **pkg_name** (*str*) – The ROS package name.
- **model_name** (*str*) – The model file name.

- **folder** (*str*) – The folder where the model is located. Default: "/urdf"
- **args_xacro** (*list of str.*) – The xacro arguments in a list. Eg: ['arg1:=True','arg2:=10.0']

Returns
The URDF string or None if the pkg or file was not found.

Return type
str

ros_urdf.**urdf_parse_from_path**(*model_path*, *args_xacro=None*) → bool

Function to parse a URDF from a file and return the URDF string.

Parameters

- **model_path** (*str*) – The model file path.
- **args_xacro** (*list of str.*) – The xacro arguments in a list. Eg: ['arg1:=True','arg2:=10.0']

Returns
The URDF string or None if the file was not found.

Return type
str

2.8.8 robot_BasicEnv

Basic robot enviroment, inheriting the Gym basic env. This class is inherited by every other robot enviroment in the **FRobs_RL** library.

class robot_BasicEnv.**RobotBasicEnv**(*launch_gazebo=False*, *gazebo_init_paused=True*, *gazebo_use_gui=True*, *gazebo_recording=False*, *gazebo_freq=100*, *world_path=None*, *world_pkg=None*, *world_filename=None*, *gazebo_max_freq=None*, *gazebo_timestep=None*, *spawn_robot=False*, *model_name_in_gazebo='robot'*, *namespace='/robot'*, *pkg_name=None*, *urdf_file=None*, *urdf_folder='/urdf'*, *controller_file=None*, *controller_list=None*, *urdf_xacro_args=None*, *rob_state_publisher_max_freq=None*, *rob_st_term=False*, *model_pos_x=0.0*, *model_pos_y=0.0*, *model_pos_z=0.0*, *model_ori_x=0.0*, *model_ori_y=0.0*, *model_ori_z=0.0*, *model_ori_w=0.0*, *reset_controllers=False*, *reset_mode=1*, *step_mode=1*, *num_gazebo_steps=1*)

Basic enviroment for all the robot environments in the frobs_rl library. To use a custom world, one can use two options: 1) set the path directly to the world file (`world_path`) or set the pkg name and world filename (`world_pkg` and `world_filename`).

Parameters

- **launch_gazebo** (*bool*) – If True, launch Gazebo at the start of the env.
- **gazebo_init_paused** (*bool*) – If True, Gazebo is initialized in a paused state.
- **gazebo_use_gui** (*bool*) – If True, Gazebo is launched with a GUI (through gzclient).
- **gazebo_recording** (*bool*) – If True, Gazebo is launched with a recording of the GUI (through gzclient).
- **gazebo_freq** (*int*) – The publish rate of gazebo in Hz.

- **world_path** (*str*) – If using a custom world then the path to the world.
- **world_pkg** (*str*) – If using a custom world then the package name of the world.
- **world_filename** (*str*) – If using a custom world then the filename of the world.
- **gazebo_max_freq** (*float*) – max update rate for gazebo in real time factor: 1 is real time, 10 is 10 times real time.
- **gazebo_timestep** (*float*) – The timestep of gazebo in seconds.
- **spawn_robot** (*bool*) – If True, the robot is spawned in the environment.
- **model_name_in_gazebo** (*str*) – The name of the model in gazebo.
- **namespace** (*str*) – The namespace of the robot.
- **pkg_name** (*str*) – The package name where the robot model is located.
- **urdf_file** (*str*) – The path to the urdf file of the robot.
- **urdf_folder** (*str*) – The path to the folder where the urdf files are located. Default is "/urdf".
- **urdf_xacro_args** (*str*) – The arguments to be passed to the xacro parser.
- **controller_file** (*str*) – The path to the controllers YAML file of the robot.
- **controller_list** (*list of str*) – The list of controllers to be launched.
- **rob_state_publisher_max_freq** (*int*) – The maximum frequency of the ros state publisher.
- **rob_st_term** (*bool*) – If True, the robot state publisher is launched in a new terminal.
- **model_pos_x** – The x position of the robot in the world.
- **model_pos_y** – The y position of the robot in the world.
- **model_pos_z** – The z position of the robot in the world.
- **model_ori_x** – The x orientation of the robot in the world.
- **model_ori_y** – The y orientation of the robot in the world.
- **model_ori_z** – The z orientation of the robot in the world.
- **model_ori_w** – The w orientation of the robot in the world.
- **reset_controllers** (*bool*) – If True, the controllers are reset at the start of each episode.
- **reset_mode** – If 1, reset Gazebo with a "reset_world" (Does not reset time) If 2, reset Gazebo with a "reset_simulation" (Resets time)
- **step_mode** – If 1, step Gazebo using the "pause_physics" and "unpause_physics" services. If 2, step Gazebo using the "step_simulation" command.
- **num_gazebo_steps** – If using step_mode 2, the number of steps to be taken.

step(*action*)

Function to send an action to the robot and get the observation and reward.

reset()

Function to reset the enviroment after an episode is done.

close()

Function to close the environment when training is done.

_send_action(*action*)

Function to send an action to the robot

_get_observation()

Function to get the observation from the enviroment.

_get_reward()

Function to get the reward from the enviroment.

_check_if_done()

Function to check if the episode is done.

If the episode has a success condition then set done as:
self.info['is_success'] = 1.0

_reset_gazebo()

Function to reset the gazebo simulation.

_check_subs_and_pubs_connection()

Function to check if the gazebo and ros connections are ready

_set_episode_init_params()

Function to set some parameters, like the position of the robot, at the begining of each episode.

2.8.9 Custom Robot enviroment

Template for a custom robot enviroment. The **CustomRobotEnv** must the first class to be filled, as the **CustomTaskEnv** will inherit this class.

```
#!/bin/python3

from gym import spaces
from gym.envs.registration import register
from frobs_rl.envs import robot_BasicEnv
import rospy

#- Uncomment the library modules as neeeed
# from frobs_rl.common import ros_gazebo
# from frobs_rl.common import ros_controllers
# from frobs_rl.common import ros_node
# from frobs_rl.common import ros_launch
# from frobs_rl.common import ros_params
# from frobs_rl.common import ros_urdf
# from frobs_rl.common import ros_spawn

"""
Although it is best to register only the task environment, one can also register the
robot environment.
"""
# register(
#         id='CustomRobotEnv-v0',
#         entry_point='frobs_rl.templates.CustomRobotEnv:CustomRobotEnv',
#         max_episode_steps=10000,
#     )
```

(continues on next page)

(continued from previous page)

```
class CustomRobotEnv(robot_BasicEnv.RobotBasicEnv):
    """
    Custom Robot Env, use this for all task envs using the custom robot.
    """

    def __init__(self):
        """
        Describe the robot used in the env.
        """
        rospy.loginfo("Starting Custom Robot Env")

        """
        If launching Gazebo with the env then set the corresponding environment␣
↪variables.
        """
        launch_gazebo=False
        gazebo_init_paused=True
        gazebo_use_gui=True
        gazebo_recording=False
        gazebo_freq=100
        gazebo_max_freq=None
        gazebo_timestep=None

        """
        If launching Gazebo with a custom world then set the corresponding environment␣
↪variables.
        """
        world_path=None
        world_pkg=None
        world_filename=None

        """
        If spawning the robot using the given spawner then set the corresponding␣
↪environment variables.
        """
        spawn_robot=False
        model_name_in_gazebo="robot"
        namespace="/robot"
        pkg_name=None
        urdf_file=None
        urdf_folder="/urdf"
        controller_file=None
        controller_list=None
        urdf_xacro_args=None
        rob_state_publisher_max_freq= None
        model_pos_x=0.0; model_pos_y=0.0; model_pos_z=0.0
        model_ori_x=0.0; model_ori_y=0.0; model_ori_z=0.0; model_ori_w=0.0

        """
        Set if the controllers in "controller_list" will be reset at the beginning of␣
↪each episode, default is False.
```

(continues on next page)

(continued from previous page)

```
        """
        reset_controllers=False

        """
        Set the reset mode of gazebo at the beginning of each episode: 1 is "reset_world
↪", 2 is "reset_simulation". Default is 1.
        """
        reset_mode=1

        """
        Set the step mode of Gazebo. 1 is "using ROS services", 2 is "using step␣
↪function of Gazebo". Default is 1.
        If using the step mode 2 then set the number of steps of Gazebo to take in each␣
↪episode. Default is 1.
        """
        step_mode=1
        num_gazebo_steps=1

        """
        Init the parent class with the corresponding variables.
        """
        super(CustomRobotEnv, self).__init__(   launch_gazebo=launch_gazebo, gazebo_init_
↪paused=gazebo_init_paused,
                    gazebo_use_gui=gazebo_use_gui, gazebo_recording=gazebo_recording,␣
↪gazebo_freq=gazebo_freq, world_path=world_path,
                    world_pkg=world_pkg, world_filename=world_filename, gazebo_max_
↪freq=gazebo_max_freq, gazebo_timestep=gazebo_timestep,
                    spawn_robot=spawn_robot, model_name_in_gazebo=model_name_in_gazebo,␣
↪namespace=namespace, pkg_name=pkg_name,
                    urdf_file=urdf_file, urdf_folder=urdf_folder, controller_
↪file=controller_file, controller_list=controller_list,
                    urdf_xacro_args=urdf_xacro_args, rob_state_publisher_max_freq= rob_
↪state_publisher_max_freq,
                    model_pos_x=model_pos_x, model_pos_y=model_pos_y, model_pos_z=model_
↪pos_z,
                    model_ori_x=model_ori_x, model_ori_y=model_ori_y, model_ori_z=model_
↪ori_z, model_ori_w=model_ori_w,
                    reset_controllers=reset_controllers, reset_mode=reset_mode, step_
↪mode=step_mode, num_gazebo_steps=num_gazebo_steps)

        """
        Define publisher or subscribers as needed.
        """
        # self.pub1 = rospy.Publisher('/robot/controller_manager/command', JointState,␣
↪queue_size=1)
        # self.sub1 = rospy.Subscriber('/robot/joint_states', JointState, self.callback1)

        """
        If using the __check_subs_and_pubs_connection method, then un-comment the lines␣
↪below.
        """
        # ros_gazebo.gazebo_unpause_physics()
```

(continues on next page)

(continued from previous page)

```
        # self._check_subs_and_pubs_connection()
        # ros_gazebo.gazebo_pause_physics()

        """
        Finished __init__ method
        """
        rospy.loginfo("Finished Init of Custom Robot env")

    #-----------------------------------------#
    #   Custom methods for the CustomRobotEnv  #

    def _check_subs_and_pubs_connection(self):
        """
        Function to check if the Gazebo and ROS connections are ready
        """
        return True

    #-------------------------------------------------------#
    #   Custom available methods for the CustomRobotEnv     #
    #   Although it is best to implement these methods in   #
    #   the Task Env, one can use them here if needed.      #

    def _send_action(self, action):
        """
        Function to send an action to the robot
        """
        raise NotImplementedError()


    def _get_observation(self):
        """
        Function to get the observation from the enviroment.
        """
        raise NotImplementedError()

    def _get_reward(self):
        """
        Function to get the reward from the enviroment.
        """
        raise NotImplementedError()

    def _check_if_done(self):
        """
        Function to check if the episode is done.

        If the episode has a success condition then set done as:
            self.info['is_success'] = 1.0
        """
        raise NotImplementedError()

    def _set_episode_init_params(self):
        """
```

(continues on next page)

(continued from previous page)

```
        Function to set some parameters, like the position of the robot, at the begining␣
↪of each episode.
        """
        raise NotImplementedError()
```

2.8.10 Custom Task enviroment

Template for a custom task enviroment. In this class the functions related to the action, observation and reward must be filled.

```
#!/bin/python3

from gym import spaces
from gym.envs.registration import register
from frobs_rl.templates import CustomRobotEnv # Replace with your own robot env
import rospy

#- Uncomment the library modules as neeeed
# from frobs_rl.common import ros_gazebo
# from frobs_rl.common import ros_controllers
# from frobs_rl.common import ros_node
# from frobs_rl.common import ros_launch
# from frobs_rl.common import ros_params
# from frobs_rl.common import ros_urdf
# from frobs_rl.common import ros_spawn

register(
        id='CustomTaskEnv-v0',
        entry_point='frobs_rl.templates.CustomTaskEnv:CustomTaskEnv',
        max_episode_steps=10000,
    )

class CustomTaskEnv(CustomRobotEnv.CustomRobotEnv):
    """
    Custom Task Env, use this env to implement a task using the robot defined in the␣
↪CustomRobotEnv
    """

    def __init__(self):
        """
        Describe the task.
        """
        rospy.loginfo("Starting Task Env")

        """
        Init super class.
        """
        super(CustomTaskEnv, self).__init__()

        """
        Define the action and observation space.
```

(continues on next page)

(continued from previous page)

```
        """
        # self.action_space = spaces.Discrete(n_actions)
        # self.action_space = spaces.Box(low=0, high=1, shape=(1,), dtype=np.float32)

        # self.observation_space = spaces.Discrete(n_observations)
        # self.observation_space = spaces.Box(low=0, high=1, shape=(1,), dtype=np.
↪float32)

        """
        Define subscribers or publishers as needed.
        """
        # self.pub1 = rospy.Publisher('/robot/controller_manager/command', JointState,␣
↪queue_size=1)
        # self.sub1 = rospy.Subscriber('/robot/joint_states', JointState, self.callback1)

        """
        Finished __init__ method
        """
        rospy.loginfo("Finished Init of Custom Task env")

    #-----------------------------------------------------------#
    #   Custom available methods for the CustomTaskEnv      #

    def _set_episode_init_params(self):
        """
        Function to set some parameters, like the position of the robot, at the␣
↪beginning of each episode.
        """
        raise NotImplementedError()

    def _send_action(self, action):
        """
        Function to send an action to the robot
        """
        raise NotImplementedError()


    def _get_observation(self):
        """
        Function to get the observation from the environment.
        """
        raise NotImplementedError()

    def _get_reward(self):
        """
        Function to get the reward from the environment.
        """
        raise NotImplementedError()

    def _check_if_done(self):
        """
        Function to check if the episode is done.
```

(continues on next page)

(continued from previous page)

```
        If the episode has a success condition then set done as:
            self.info['is_success'] = 1.0
        """
        raise NotImplementedError()
```

2.8.11 Basic Model

Basic model class which is inherited by all models supported by the **FRobs_RL** library.

class basic_model.**BasicModel**(*env*, *save_model_path*, *log_path*, *ns='/'*, *load_trained=False*)

Base class for all the algorithms supported by the frobs_rl library.

Parameters

- **env** – The environment to be used.
- **save_model_path** – The path to save the model.
- **log_path** – The path to save the log.
- **ns** – The namespace of the parameters.
- **load_trained** – Whether or not to load a trained model.

check_env() → bool

Use the stable-baselines check_env method to check the environment.

Returns
True if the environment is correct, False otherwise.

Return type
bool

close_env() → bool

Use the env close method to close the environment.

Returns
True if the environment was closed, False otherwise.

Return type
bool

predict(*observation*, *state=None*, *mask=None*, *deterministic=False*)

Get the current action based on the observation, state or mask

Parameters

- **observation** (*ndarray*) – The enviroment observation
- **state** (*ndarray*) – The previous states of the enviroment, used in recurrent policies.
- **mask** (*ndarray*) – The mask of the last states, used in recurrent policies.
- **deterministic** (*bool*) – Whether or not to return deterministic actions.

Returns
The action to be taken and the next state(for recurrent policies)

Return type
ndarray, ndarray

save_model() → bool

Function to save the model.

Returns
True if the model was saved, False otherwise.

Return type
bool

save_replay_buffer() → bool

Funtion to save the replay buffer, to be used the training must be finished or an error will be raised.

Returns
True if the replay buffer was saved, False otherwise.

Return type
bool

set_model_logger() → bool

Function to set the logger of the model.

Returns
True if the logger was set, False otherwise.

Return type
bool

train() → bool

Function to train the model the number of steps specified in the ROS parameter server. The function will automatically save the model after training.

Returns
True if the training was successful, False otherwise.

Return type
bool

2.8.12 Advantage Actor Critic (A2C)

API for the A2C algorithm implemented in A2C stable_baselines3.

Original paper: https://arxiv.org/abs/1602.01783 OpenAI blog post: https://openai.com/blog/baselines-acktr-a2c/

YAML parameters template

```yaml
model_params:

  # Training
  training_steps: 5000      # The number of training steps to perform

  # Save params
  save_freq: 1000
  save_prefix: a2c_model
  trained_model_name: trained_model
  save_replay_buffer: False
```

(continues on next page)

(continued from previous page)

```yaml
  # Load model params
  load_model: False
  model_name: a2c_model_5000_steps

  # Logging parameters
  log_folder: A2C_1
  log_interval: 4 # The number of episodes between logs
  reset_num_timesteps: False # If true, will reset the number of timesteps to 0 every training

  # Use custom policy - Only MlpPolicy is supported (Only used when new model is created)
  use_custom_policy: False
  policy_params:
    net_arch: [400, 300] # List of hidden layer sizes
    activation_fn: relu  # relu, tanh, elu or selu
    features_extractor_class: FlattenExtractor # FlattenExtractor, BaseFeaturesExtractor or CombinedExtractor
    optimizer_class: Adam # Adam, Adadelta, Adagrad, RMSprop or SGD

  # Use SDE
  use_sde: False
  sde_params:
    sde_sample_freq: -1

  # A2C parameters
  a2c_params:
    learning_rate: 0.0007
    n_steps: 5
    gamma: 0.99
    gae_lambda: 1.0
    ent_coef: 0.0
    vf_coef: 0.5
    max_grad_norm: 0.5
    use_rms_prop: True
    rms_prop_eps: 0.00001
    normalize_advantage: False
```

Class docs

class a2c.**A2C**(*env*, *save_model_path*, *log_path*, *load_trained=False*, *config_file_pkg='frobs_rl'*, *config_filename='a2c_config.yaml'*, *ns='/'*)

Advantage Actor-Critic (A2C) algorithm.

Paper: https://arxiv.org/abs/1602.01783

Parameters

- **env** – The environment to be used.
- **save_model_path** – The path to save the model.
- **log_path** – The path to save the log.
- **load_trained** – If True, load a trained model.

- **config_file_pkg** – The package where the config file is located. Default: frobs_rl.
- **config_filename** – The name of the config file. Default: a2c_config.yaml.
- **ns** – The namespace of the ROS parameters. Default: "/".

check_env() → bool

Use the stable-baselines check_env method to check the environment.

Returns
True if the environment is correct, False otherwise.

Return type
bool

close_env() → bool

Use the env close method to close the environment.

Returns
True if the environment was closed, False otherwise.

Return type
bool

predict(*observation*, *state=None*, *mask=None*, *deterministic=False*)

Get the current action based on the observation, state or mask

Parameters

- **observation** (*ndarray*) – The enviroment observation
- **state** (*ndarray*) – The previous states of the enviroment, used in recurrent policies.
- **mask** (*ndarray*) – The mask of the last states, used in recurrent policies.
- **deterministic** (*bool*) – Whether or not to return deterministic actions.

Returns
The action to be taken and the next state(for recurrent policies)

Return type
ndarray, ndarray

save_model() → bool

Function to save the model.

Returns
True if the model was saved, False otherwise.

Return type
bool

save_replay_buffer() → bool

Funtion to save the replay buffer, to be used the training must be finished or an error will be raised.

Returns
True if the replay buffer was saved, False otherwise.

Return type
bool

set_model_logger() → bool

Function to set the logger of the model.

Returns
True if the logger was set, False otherwise.

Return type
bool

`train()` → bool

Function to train the model the number of steps specified in the ROS parameter server. The function will automatically save the model after training.

Returns
True if the training was successful, False otherwise.

Return type
bool

`load_trained(env=None)`

Load a trained model. Use only with predict function, as the logs will not be saved.

Parameters

- **model_path** (*str*) – The path to the trained model.
- **env** (*gym.Env*) – The environment to be used.

Returns
The loaded model.

Return type
frobs_rl.A2C

2.8.13 Deep Deterministic Policy Gradient (DDPG)

API for the DDPG algorithm implemented in DDPG stable_baselines3.

Original paper: https://arxiv.org/abs/1509.02971 OpenAI blog post: https://spinningup.openai.com/en/latest/algorithms/ddpg.html

YAML parameters template

```yaml
model_params:

  # Training
  training_steps: 5000      # The number of training steps to perform

  # Save params
  save_freq: 1000
  save_prefix: ddpg_model
  trained_model_name: trained_model
  save_replay_buffer: False

  # Load model params
  load_model: False
  model_name: ddpg_model_5000_steps

  # Logging parameters
```

(continues on next page)

(continued from previous page)

```yaml
  log_folder: DDPG_1
  log_interval: 4 # The number of episodes between logs
  reset_num_timesteps: False # If true, will reset the number of timesteps to 0 every training

  # Use custom policy - Only MlpPolicy is supported (Only used when new model is created)
  use_custom_policy: False
  policy_params:
    net_arch: [400, 300] # List of hidden layer sizes
    activation_fn: relu  # relu, tanh, elu or selu
    features_extractor_class: FlattenExtractor # FlattenExtractor, BaseFeaturesExtractor or CombinedExtractor
    optimizer_class: Adam # Adam, Adadelta, Adagrad, RMSprop or SGD

  # UseAction noise
  use_action_noise: True # For now only Gaussian noise is supported
  action_noise:
    mean: 0.0
    sigma: 0.01

  # DDPG parameters
  ddpg_params:
    learning_rate: 0.001
    buffer_size: 1000000
    learning_starts: 100
    batch_size: 100
    tau: 0.005
    gamma: 0.99
    gradient_steps: -1
    train_freq:
      freq: 20
      unit: step # episode or step
```

Class docs

class ddpg.**DDPG**(*env*, *save_model_path*, *log_path*, *load_trained=False*, *config_file_pkg='frobs_rl'*, *config_filename='ddpg_config.yaml'*, *ns='/'*)

Deep Deterministic Policy Gradient (DDPG) algorithm.

Paper: https://arxiv.org/abs/1509.02971

Parameters

- **env** – The environment to be used.
- **save_model_path** – The path to save the model.
- **log_path** – The path to save the log.
- **load_trained** – Whether to load a trained model.
- **config_file_pkg** – The package where the config file is located. Default: frobs_rl.
- **config_filename** – The name of the config file. Default: ddpg_config.yaml.
- **ns** – The namespace of the ROS parameters. Default: "/".

check_env() → bool

Use the stable-baselines check_env method to check the environment.

Returns
True if the environment is correct, False otherwise.

Return type
bool

close_env() → bool

Use the env close method to close the environment.

Returns
True if the environment was closed, False otherwise.

Return type
bool

predict(*observation*, *state=None*, *mask=None*, *deterministic=False*)

Get the current action based on the observation, state or mask

Parameters

- **observation** (*ndarray*) – The enviroment observation
- **state** (*ndarray*) – The previous states of the enviroment, used in recurrent policies.
- **mask** (*ndarray*) – The mask of the last states, used in recurrent policies.
- **deterministic** (*bool*) – Whether or not to return deterministic actions.

Returns
The action to be taken and the next state(for recurrent policies)

Return type
ndarray, ndarray

save_model() → bool

Function to save the model.

Returns
True if the model was saved, False otherwise.

Return type
bool

save_replay_buffer() → bool

Funtion to save the replay buffer, to be used the training must be finished or an error will be raised.

Returns
True if the replay buffer was saved, False otherwise.

Return type
bool

set_model_logger() → bool

Function to set the logger of the model.

Returns
True if the logger was set, False otherwise.

Return type
bool

train() → bool

Function to train the model the number of steps specified in the ROS parameter server. The function will automatically save the model after training.

Returns
True if the training was successful, False otherwise.

Return type
bool

load_trained(*env=None*)

Load a trained model. Use only with predict function, as the logs will not be saved.

Parameters

- **model_path** (*str*) – The path to the trained model.
- **env** (*gym.Env*) – The environment to be used.

Returns
The trained model.

Return type
frobs_rl.DDPG

2.8.14 Deep Q Network (DQN)

API for the DQN algorithm implemented in DQN stable_baselines3.

Original paper: https://arxiv.org/abs/1312.5602 Nature paper: https://www.nature.com/articles/nature14236

YAML parameters template

```yaml
model_params:

  # Training
  training_steps: 5000      # The number of training steps to perform

  # Save params
  save_freq: 1000
  save_prefix: dqn_model
  trained_model_name: trained_model
  save_replay_buffer: False

  # Load model params
  load_model: False
  model_name: dqn_model_5000_steps

  # Logging parameters
  log_folder: DQN_1
  log_interval: 4 # The number of episodes between logs
  reset_num_timesteps: False # If true, will reset the number of timesteps to 0 every␣
↪training

  # Use custom policy - Only MlpPolicy is supported (Only used when new model is created)
```

(continues on next page)

(continued from previous page)

```
  use_custom_policy: False
  policy_params:
    net_arch: [400, 300] # List of hidden layer sizes
    activation_fn: relu  # relu, tanh, elu or selu
    features_extractor_class: FlattenExtractor # FlattenExtractor, BaseFeaturesExtractor␣
↪or CombinedExtractor
    optimizer_class: Adam # Adam, Adadelta, Adagrad, RMSprop or SGD

  # DQN parameters
  dqn_params:
    learning_rate: 0.0001
    buffer_size: 1000000
    learning_starts: 50000
    batch_size: 32
    tau: 1.0
    gamma: 0.99
    gradient_steps: 1
    target_update_interval: 10000
    exploration_fraction: 0.1
    exploration_initial_eps: 1.0
    exploration_final_eps: 0.05
    max_grad_norm: 10
    train_freq:
      freq: 20
      unit: step  # episode or step
```

Class docs

class dqn.**DQN**(*env*, *save_model_path*, *log_path*, *load_trained=False*, *config_file_pkg='frobs_rl'*, *config_filename='dqn_config.yaml'*, *ns='/'*)

Deep Q Network (DQN) algorithm.

Paper: https://arxiv.org/abs/1312.5602

Parameters

- **env** – The environment to be used.
- **save_model_path** – The path to save the model.
- **log_path** – The path to save the log.
- **load_trained** – Whether to load a trained model.
- **config_file_pkg** – The package where the config file is located. Default: frobs_rl.
- **config_filename** – The name of the config file. Default: dqn_config.yaml.
- **ns** – The namespace of the ROS parameters. Default: "/".

check_env() → bool

Use the stable-baselines check_env method to check the environment.

Returns

True if the environment is correct, False otherwise.

Return type
bool

`close_env()` → bool

Use the env close method to close the environment.

Returns
True if the environment was closed, False otherwise.

Return type
bool

`predict(observation, state=None, mask=None, deterministic=False)`

Get the current action based on the observation, state or mask

Parameters

- **observation** (*ndarray*) – The enviroment observation
- **state** (*ndarray*) – The previous states of the enviroment, used in recurrent policies.
- **mask** (*ndarray*) – The mask of the last states, used in recurrent policies.
- **deterministic** (*bool*) – Whether or not to return deterministic actions.

Returns
The action to be taken and the next state(for recurrent policies)

Return type
ndarray, ndarray

`save_model()` → bool

Function to save the model.

Returns
True if the model was saved, False otherwise.

Return type
bool

`save_replay_buffer()` → bool

Funtion to save the replay buffer, to be used the training must be finished or an error will be raised.

Returns
True if the replay buffer was saved, False otherwise.

Return type
bool

`set_model_logger()` → bool

Function to set the logger of the model.

Returns
True if the logger was set, False otherwise.

Return type
bool

`train()` → bool

Function to train the model the number of steps specified in the ROS parameter server. The function will automatically save the model after training.

Returns
True if the training was successful, False otherwise.

Return type
bool

load_trained(*env=None*)
Load a trained model. Use only with predict function, as the logs will not be saved.

Parameters

- **model_path** (*str*) – The path to the trained model.
- **env** (*gym.Env*) – The environment to be used.

Returns
The trained model.

Return type
frobs_rl.DQN

2.8.15 Proximal Policy Optimization (PPO)

API for the PPO algorithm implemented in PPO stable_baselines3.

Original paper: https://arxiv.org/abs/1707.06347 OpenAI blog post: https://blog.openai.com/openai-baselines-ppo/

YAML parameters template

```yaml
model_params:

  # Training
  training_steps: 5000      # The number of training steps to perform

  # Save params
  save_freq: 1000
  save_prefix: ppo_model
  trained_model_name: trained_model
  save_replay_buffer: False # PPO does not support saving replay buffer

  # Load model params
  load_model: False
  model_name: ppo_model_5000_steps

  # Logging parameters
  log_folder: PPO_1
  log_interval: 4 # The number of episodes between logs
  reset_num_timesteps: False # If true, will reset the number of timesteps to 0 every␣
↪training

  # Use custom policy - Only MlpPolicy is supported (Only used when new model is created)
  use_custom_policy: False
  policy_params:
    net_arch: [400, 300] # List of hidden layer sizes
    activation_fn: relu  # relu, tanh, elu or selu
    features_extractor_class: FlattenExtractor # FlattenExtractor, BaseFeaturesExtractor␣
↪or CombinedExtractor
    optimizer_class: Adam # Adam, Adadelta, Adagrad, RMSprop or SGD
```

(continues on next page)

(continued from previous page)

```
  # Use SDE
  use_sde: False
  sde_params:
    sde_sample_freq: -1

  # PPO parameters
  ppo_params:
    learning_rate: 0.0003
    n_steps: 100    # The number of steps to run for each environment per update (i.e.␣
↪rollout buffer size is n_steps * n_envs where n_envs is number of environment copies␣
↪running in parallel)
    batch_size: 100 # Minibatch size
    n_epochs: 5     # Number of epoch when optimizing the surrogate loss
    gamma: 0.99
    gae_lambda: 0.95
    clip_range: 0.2
    ent_coef: 0.0
    vf_coef: 0.5
    max_grad_norm: 0.5
```

Class docs

class ppo.**PPO**(*env*, *save_model_path*, *log_path*, *load_trained=False*, *config_file_pkg='frobs_rl'*, *config_filename='ppo_config.yaml'*, *ns='/'*)

Proximal Policy Optimization (PPO) algorithm.

Paper: https://arxiv.org/abs/1707.06347

Parameters

- **env** – The environment to be used.
- **save_model_path** – The path to save the model.
- **log_path** – The path to save the log.
- **load_trained** – Whether to load a trained model or not.
- **config_file_pkg** – The package where the config file is located. Default: frobs_rl.
- **config_filename** – The name of the config file. Default: ppo_config.yaml.
- **ns** – The namespace of the ROS parameters. Default: "/".

check_env() → bool

Use the stable-baselines check_env method to check the environment.

Returns
True if the environment is correct, False otherwise.

Return type
bool

close_env() → bool

Use the env close method to close the environment.

Returns
True if the environment was closed, False otherwise.

Return type
bool

predict(*observation*, *state=None*, *mask=None*, *deterministic=False*)

Get the current action based on the observation, state or mask

Parameters

- **observation** (*ndarray*) – The enviroment observation
- **state** (*ndarray*) – The previous states of the enviroment, used in recurrent policies.
- **mask** (*ndarray*) – The mask of the last states, used in recurrent policies.
- **deterministic** (*bool*) – Whether or not to return deterministic actions.

Returns
The action to be taken and the next state(for recurrent policies)

Return type
ndarray, ndarray

save_model() → bool

Function to save the model.

Returns
True if the model was saved, False otherwise.

Return type
bool

save_replay_buffer() → bool

Funtion to save the replay buffer, to be used the training must be finished or an error will be raised.

Returns
True if the replay buffer was saved, False otherwise.

Return type
bool

set_model_logger() → bool

Function to set the logger of the model.

Returns
True if the logger was set, False otherwise.

Return type
bool

train() → bool

Function to train the model the number of steps specified in the ROS parameter server. The function will automatically save the model after training.

Returns
True if the training was successful, False otherwise.

Return type
bool

load_trained(*env=None*)

Load a trained model. Use only with predict function, as the logs will not be saved.

Parameters

- **model_path** (*str*) – The path to the trained model.
- **env** (*gym.Env*) – The environment to be used.

Returns

The loaded model.

Return type

frobs_rl.PPO

2.8.16 Soft Actor Critic (SAC)

API for the SAC algorithm implemented in SAC stable_baselines3.

Original paper: https://arxiv.org/abs/1801.01290 OpenAI blog post: https://spinningup.openai.com/en/latest/algorithms/sac.html

YAML parameters template

```yaml
model_params:

  # Training
  training_steps: 5000      # The number of training steps to perform

  # Save params
  save_freq: 1000
  save_prefix: sac_model
  trained_model_name: trained_model
  save_replay_buffer: False

  # Load model params
  load_model: False
  model_name: sac_model_5000_steps

  # Logging parameters
  log_folder: SAC_1
  log_interval: 4 # The number of episodes between logs
  reset_num_timesteps: False # If true, will reset the number of timesteps to 0 every␣
↪training

  # Use custom policy - Only MlpPolicy is supported (Only used when new model is created)
  use_custom_policy: False
  policy_params:
    net_arch: [400, 300] # List of hidden layer sizes
    activation_fn: relu  # relu, tanh, elu or selu
    features_extractor_class: FlattenExtractor # FlattenExtractor, BaseFeaturesExtractor␣
↪or CombinedExtractor
    optimizer_class: Adam # Adam, Adadelta, Adagrad, RMSprop or SGD
```

(continues on next page)

(continued from previous page)

```yaml
  # UseAction noise
  use_action_noise: True # For now only Gaussian noise is supported
  action_noise:
    mean: 0.0
    sigma: 0.01

  # Use SDE
  use_sde: True
  sde_params:
    sde_sample_freq: -1
    use_sde_at_warmup: False

  # SAC parameters
  sac_params:
    learning_rate: 0.0003
    buffer_size: 1000000
    learning_starts: 100
    batch_size: 256
    tau: 0.005
    gamma: 0.99
    gradient_steps: 1
    ent_coef: auto
    target_update_interval: 1
    target_entropy: auto
    train_freq:
      freq: 20
      unit: step  # episode or step
```

Class docs

class sac.**SAC**(*env*, *save_model_path*, *log_path*, *load_trained=False*, *config_file_pkg='frobs_rl'*, *config_filename='sac_config.yaml'*, *ns='/'*)

Soft Actor-Critic (SAC) algorithm.

Paper: https://arxiv.org/abs/1801.01290

Parameters

- **env** – The environment to be used.
- **save_model_path** – The path to save the model.
- **log_path** – The path to save the log.
- **load_trained** – Whether to load a trained model or not.
- **config_file_pkg** – The package where the config file is located. Default: frobs_rl.
- **config_filename** – The name of the config file. Default: sac_config.yaml.
- **ns** – The namespace of the ROS parameters. Default: "/".

check_env() → bool

Use the stable-baselines check_env method to check the environment.

Returns

True if the environment is correct, False otherwise.

Return type
bool

`close_env()` → bool

Use the env close method to close the environment.

Returns
True if the environment was closed, False otherwise.

Return type
bool

`predict(observation, state=None, mask=None, deterministic=False)`

Get the current action based on the observation, state or mask

Parameters

- **observation** (*ndarray*) – The enviroment observation
- **state** (*ndarray*) – The previous states of the enviroment, used in recurrent policies.
- **mask** (*ndarray*) – The mask of the last states, used in recurrent policies.
- **deterministic** (*bool*) – Whether or not to return deterministic actions.

Returns
The action to be taken and the next state(for recurrent policies)

Return type
ndarray, ndarray

`save_model()` → bool

Function to save the model.

Returns
True if the model was saved, False otherwise.

Return type
bool

`save_replay_buffer()` → bool

Funtion to save the replay buffer, to be used the training must be finished or an error will be raised.

Returns
True if the replay buffer was saved, False otherwise.

Return type
bool

`set_model_logger()` → bool

Function to set the logger of the model.

Returns
True if the logger was set, False otherwise.

Return type
bool

`train()` → bool

Function to train the model the number of steps specified in the ROS parameter server. The function will automatically save the model after training.

Returns
True if the training was successful, False otherwise.

Return type
bool

load_trained(*env=None*)
Load a trained model. Use only with predict function, as the logs will not be saved.

Parameters

- **model_path** (*str*) – The path to the trained model.
- **env** (*gym.Env*) – The environment to be used.

Returns
The trained model.

Return type
frobs_rl.SAC

2.8.17 Twin Delayed Deep Deterministic Policy Gradient (TD3)

API for the TD3 algorithm implemented in TD3 stable_baselines3.

Original paper: https://arxiv.org/pdf/1802.09477.pdf OpenAI blog post: https://spinningup.openai.com/en/latest/algorithms/td3.html

YAML parameters template

```yaml
model_params:

  # Training
  training_steps: 5000      # The number of training steps to perform

  # Save params
  save_freq: 1000
  save_prefix: td3_model
  trained_model_name: trained_model
  save_replay_buffer: False

  # Load model params
  load_model: False
  model_name: td3_model_5000_steps

  # Logging parameters
  log_folder: TD3_1
  log_interval: 4 # The number of episodes between logs
  reset_num_timesteps: False # If true, will reset the number of timesteps to 0 every training

  # Use custom policy - Only MlpPolicy is supported (Only used when new model is created)
  use_custom_policy: False
  policy_params:
    net_arch: [400, 300] # List of hidden layer sizes
    activation_fn: relu  # relu, tanh, elu or selu
    features_extractor_class: FlattenExtractor # FlattenExtractor, BaseFeaturesExtractor or CombinedExtractor
```

(continues on next page)

(continued from previous page)

```
    optimizer_class: Adam # Adam, Adadelta, Adagrad, RMSprop or SGD

  # UseAction noise
  use_action_noise: True # For now only Gaussian noise is supported
  action_noise:
    mean: 0.0
    sigma: 0.01

  # TD3 parameters
  td3_params:
    learning_rate: 0.001
    buffer_size: 1000000
    learning_starts: 100
    batch_size: 100
    tau: 0.005
    gamma: 0.99
    gradient_steps: -1
    policy_delay: 2
    target_policy_noise: 0.2
    target_noise_clip: 0.5
    train_freq:
      freq: 20
      unit: step  # episode or step
```

Class docs

class td3.**TD3**(*env*, *save_model_path*, *log_path*, *load_trained=False*, *config_file_pkg='frobs_rl'*, *config_filename='td3_config.yaml'*, *ns='/'*)

Twin Delayed DDPG (TD3) algorithm.

Paper: https://arxiv.org/abs/1802.09477

Parameters

- **env** – The environment to be used.
- **save_model_path** – The path to save the model.
- **log_path** – The path to save the log.
- **load_trained** – Whether to load a trained model(Used in load function).
- **config_file_pkg** – The package where the config file is located. Default: frobs_rl.
- **config_filename** – The name of the config file. Default: td3_config.yaml.
- **ns** – The namespace of the ROS parameters. Default: "/".

check_env() → bool

Use the stable-baselines check_env method to check the environment.

Returns
True if the environment is correct, False otherwise.

Return type
bool

close_env() → bool

Use the env close method to close the environment.

Returns
True if the environment was closed, False otherwise.

Return type
bool

predict(*observation*, *state=None*, *mask=None*, *deterministic=False*)

Get the current action based on the observation, state or mask

Parameters

- **observation** (*ndarray*) – The enviroment observation
- **state** (*ndarray*) – The previous states of the enviroment, used in recurrent policies.
- **mask** (*ndarray*) – The mask of the last states, used in recurrent policies.
- **deterministic** (*bool*) – Whether or not to return deterministic actions.

Returns
The action to be taken and the next state(for recurrent policies)

Return type
ndarray, ndarray

save_model() → bool

Function to save the model.

Returns
True if the model was saved, False otherwise.

Return type
bool

save_replay_buffer() → bool

Funtion to save the replay buffer, to be used the training must be finished or an error will be raised.

Returns
True if the replay buffer was saved, False otherwise.

Return type
bool

set_model_logger() → bool

Function to set the logger of the model.

Returns
True if the logger was set, False otherwise.

Return type
bool

train() → bool

Function to train the model the number of steps specified in the ROS parameter server. The function will automatically save the model after training.

Returns
True if the training was successful, False otherwise.

Return type
bool

load_trained(*env=None*)

Load a trained model. Use only with predict function, as the logs will not be saved.

Parameters

- **model_path** (*str*) – The path to the trained model.
- **env** (*gym.Env*) – The environment to be used.

Returns

The loaded model.

Return type

frobs_rl.TD3

2.8.18 Normalize Action Wrapper

Enviroment wrapper to normalize the action space.

Note: Only works for Box action spaces.

Example use

```
from kobuki_maze_rl.task_env import kobuki_maze
from frobs_rl.common import ros_gazebo
import gym
import rospy

from frobs_rl.wrappers.NormalizeActionWrapper import NormalizeActionWrapper

if __name__ == '__main__':

    # Launch Gazebo
    ros_gazebo.launch_Gazebo(paused=True, gui=False)

    # Start node
    rospy.logwarn("Start")
    rospy.init_node('kobuki_maze_train')

    # Launch the task environment
    env = gym.make('KobukiMazeEnv-v0')

    #--- Normalize action space
    env = NormalizeActionWrapper(env)
```

Wrapper documentation

class NormalizeActionWrapper.**NormalizeActionWrapper**(*env*)

Wrapper to normalize the action space.

Parameters
env – (gym.Env) Gym environment that will be wrapped

rescale_action(*scaled_action*)

Rescale the action from [-1, 1] to [low, high]

Parameters
scaled_action (*np.ndarray*) – The action to rescale.

Returns
The rescaled action.

Return type
np.ndarray

reset()

Reset the environment

step(*action*)

Parameters
action (*floar or int*) – Action taken by the agent

Returns
observation, reward, is the episode over, additional informations

Return type
(np.ndarray, float, bool, dict)

2.8.19 Normalize Observation Wrapper

Enviroment wrapper to normalize the observation space.

Note: Only works for Box observation spaces.

Example use

```
from kobuki_maze_rl.task_env import kobuki_maze
from frobs_rl.common import ros_gazebo
import gym
import rospy

from frobs_rl.wrappers.NormalizeObservWrapper import NormalizeObservWrapper

if __name__ == '__main__':

    # Launch Gazebo
    ros_gazebo.launch_Gazebo(paused=True, gui=False)
```

(continues on next page)

(continued from previous page)

```
    # Start node
    rospy.logwarn("Start")
    rospy.init_node('kobuki_maze_train')

    # Launch the task environment
    env = gym.make('KobukiMazeEnv-v0')

    #--- Normalize observation space
    env = NormalizeObservWrapper(env)
```

Wrapper documentation

class NormalizeObservWrapper.**NormalizeObservWrapper**(*env*)

Wrapper to normalize the observation space.

Parameters
env – (gym.Env) Gym environment that will be wrapped

reset()

Reset the environment

scale_observation(*observation*)

Scale the observation from [low, high] to [-1, 1].

Parameters
observation (*np.ndarray*) – Observation to scale

Returns
scaled observation

Return type
np.ndarray

step(*action*)

Parameters
action (*float or int*) – Action taken by the agent

Returns
observation, reward, is the episode over, additional informations

Return type
(np.ndarray, float, bool, dict)

2.8.20 Time Limit Wrapper

Enviroment wrapper to set a maximum number of steps per episode. If the agent does not succesfully end the task or reach the goal the episode will end and the property `is_success` will be set to 0.

Example use

```
from kobuki_maze_rl.task_env import kobuki_maze
from frobs_rl.common import ros_gazebo
import gym
import rospy

from frobs_rl.wrappers.TimeLimitWrapper import TimeLimitWrapper

if __name__ == '__main__':

    # Launch Gazebo
    ros_gazebo.launch_Gazebo(paused=True, gui=False)

    # Start node
    rospy.logwarn("Start")
    rospy.init_node('kobuki_maze_train')

    # Launch the task environment
    env = gym.make('KobukiMazeEnv-v0')

    #--- Set max steps
    env = TimeLimitWrapper(env, max_steps=15000)
```

Wrapper documentation

class TimeLimitWrapper.**TimeLimitWrapper**(*env*, *max_steps=100*)

Wrapper to limit the number of steps per episode.

Parameters

- **env** – (gym.Env) Gym environment that will be wrapped
- **max_steps** – (int) Max number of steps per episode

reset()

Reset the environment

step(*action*)

Parameters
action (*[float] or int*) – Action taken by the agent

Returns
observation, reward, is the episode over, additional informations

Return type
(np.ndarray, float, bool, dict)

PYTHON MODULE INDEX

n

r

t

INDEX

Symbols

A

C

D

G

I

K

T

U