panda-gym

Release v2.0.4

Quentin Gallouédec

Jul 05, 2022

GETTING STARTED

1 Installation 1

2 Quick Start 3

3 Environments 5

4 Manual control 7

5 Save and Restore States 9

6 Train with stable-baselines3 11

7 Custom robot 13

8 Custom task 17

9 Custom environment 19

10 PyBullet Class 21

11 Robot 29

12 Task 33

13 Robot-Task 35

14 Panda 37

15 Citing 39

16 Indices and tables 41

Python Module Index 43

Index 45

CHAPTER

ONE

INSTALLATION

`panda-gym` is at least compatible with Windows, MacOS and Ubuntu, and python 3.7, 3.8 and 3.9.

1.1 With pip

To install `panda-gym` with pip, run:

```
pip install panda-gym
```

1.2 From source

To install `panda-gym` from source, run:

```
git clone https://github.com/qgallouedec/panda-gym/
cd panda-gym
pip install -e .
```

CHAPTER TWO

QUICK START

Once panda-gym installed, you can start the "Reach" task by executing the following lines.

```
import gym
import panda_gym

env = gym.make('PandaReach-v2', render=True)

obs = env.reset()
done = False
while not done:
    action = env.action_space.sample() # random action
    obs, reward, done, info = env.step(action)
    env.render() # wait the right amount of time to make the rendering real-time
```

Obviously, since the chosen actions are random, you will not see any learning. To access the section dedicated to the learning of the tasks, refer to the section *Train with stable-baselines3*.

CHAPTER THREE

ENVIRONMENTS

`panda-gym` includes:

- **1 robot:**
  - the Franka Emika Panda robot,
- **6 tasks:**
  - **Reach**: the robot must place its end-effector at a target position,
  - **Push**: the robot has to push a cube to a target position,
  - **Slide**: the robot has to slide an object to a target position,
  - **Pick and place**: the robot has to pick up and place an object at a target position,
  - **Stack**: the robot has to stack two cubes at a target position,
  - **Flip**: the robot must flip the cube to a target orientation,
- **2 control modes:**
  - **End-effector displacement control**: the action corresponds to the displacement of the end-effector.
  - **Joints control**: the action corresponds to the individual motion of each joint,
- **2 reward types:**
  - **Sparse**: the environment return a reward if and only if the task is completed,
  - **Dense**: the closer the agent is to completing the task, the higher the reward.

By default, the reward is sparse and the control mode is the end-effector displacement. The complete set of environments present in the package is presented in the following list.

3.1 Sparce reward, end-effector control (default setting)

- `PandaReach-v2`
- `PandaPush-v2`
- `PandaSlide-v2`
- `PandaPickAndPlace-v2`
- `PandaStack-v2`
- `PandaFlip-v2`

3.2 Dense reward, end-effector control

- PandaReachDense-v2
- PandaPushDense-v2
- PandaSlideDense-v2
- PandaPickAndPlaceDense-v2
- PandaStackDense-v2
- PandaFlipDense-v2

3.3 Sparce reward, joints control

- PandaReachJoints-v2
- PandaPushJoints-v2
- PandaSlideJoints-v2
- PandaPickAndPlaceJoints-v2
- PandaStackJoints-v2
- PandaFlipJoints-v2

3.4 Dense reward, joints control

- PandaReachJointsDense-v2
- PandaPushJointsDense-v2
- PandaSlideJointsDense-v2
- PandaPickAndPlaceJointsDense-v2
- PandaStackJointsDense-v2
- PandaFlipJointsDense-v2

CHAPTER

FOUR

MANUAL CONTROL

It is possible to manually control the robot, giving it deterministic actions, depending on the observations. For example, for the realization of the task Reach, here is a possibility for the realization of the task.

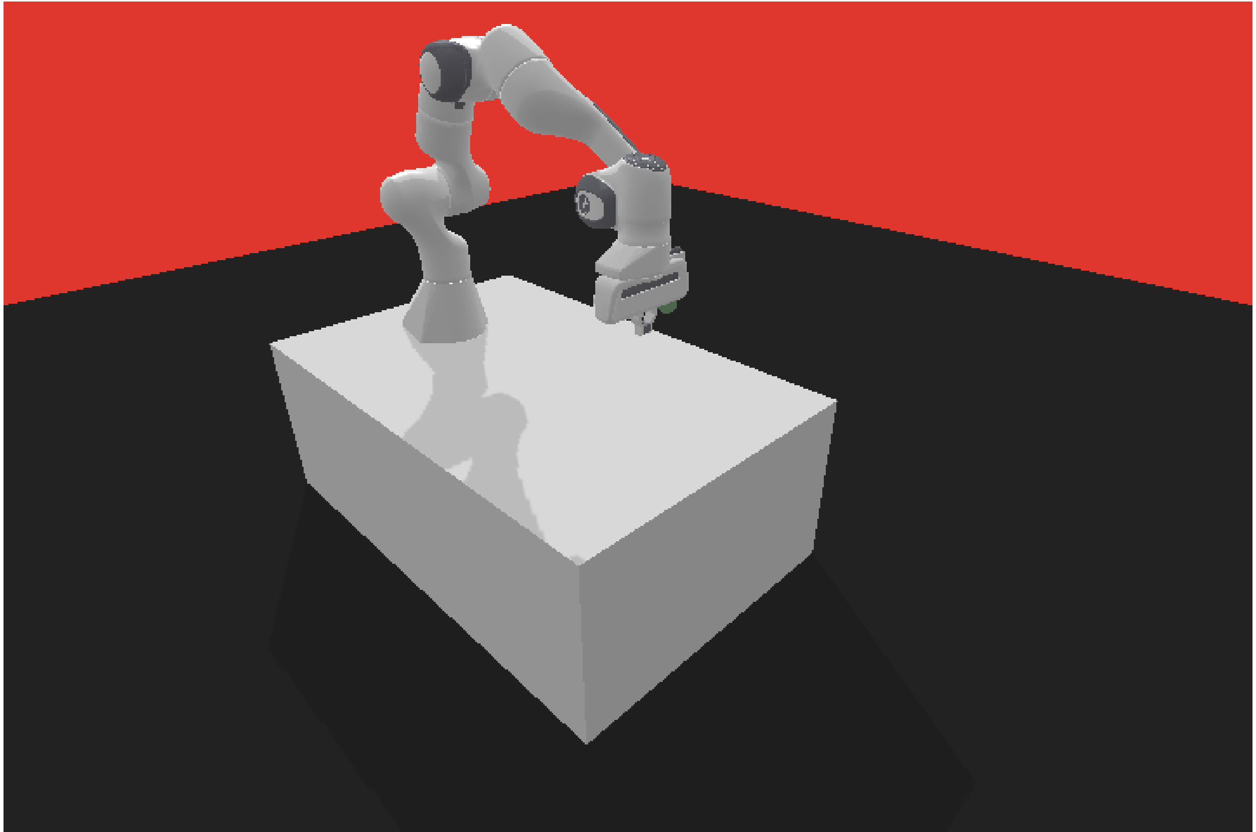
```
import gym
import panda_gym

env = gym.make("PandaReach-v2", render=True)
obs = env.reset()
done = False

while not done:
    current_position = obs["observation"][0:3]
    desired_position = obs["desired_goal"][0:3]
    action = 5.0 * (desired_position - current_position)
    obs, reward, done, info = env.step(action)
    env.render()

env.close()
```

The result is as follows.



CHAPTER

FIVE

SAVE AND RESTORE STATES

It is possible to save a state of the entire simulation environment. This is useful if your application requires lookahead search. Below is an example of a greedy random search.

```
import gym
import panda_gym

env = gym.make("PandaReachDense-v2", render=True)
obs = env.reset()

while True:
    state_id = env.save_state()
    best_action = None
    rew = best_rew = env.task.compute_reward(
        obs["achieved_goal"], obs["desired_goal"], None)

    while rew <= best_rew:
        env.restore_state(state_id)
        a = env.action_space.sample()
        _, rew, _, _ = env.step(a)

    env.restore_state(state_id)
    obs, _, _, _ = env.step(a)
    env.remove_state(state_id)

env.close()
```

CHAPTER

SIX

TRAIN WITH STABLE-BASELINES3

You can train the environments with any OpenAI/gym compatible library. In this documentation we explain how to use one of them: stable-baselines3 (SB3).

6.1 Install SB3

To install SB3, follow the instructions from its documentation Install stable-baselines3.

Alternatively, you can install `panda-gym` and SB3 directly with a single command:

```
pip install panda-gym[extra]
```

> **Warning:** If you use *zsh* terminal, the syntax is `pip install 'panda-gym[extra]'`

6.2 Train

Now that SB3 is installed, you can run the following code to train an agent. You can use every algorithm compatible with `Box` action space, see stable-baselines3/RL Algorithm). In the following example, a DDPG agent is trained to solve th Reach task.

```python
import gym
import panda_gym
from stable_baselines3 import DDPG

env = gym.make("PandaReach-v2")
model = DDPG(policy="MultiInputPolicy", env=env)
model.train(30000)
```

> **Note:** Here we provide the canonical code for training with SB3. For any information on the setting of hyperparameters, verbosity, saving the model, ... please read the SB3 documentation.

6.3 Bonus: Train with RL Baselines3 Zoo

RL Baselines3 Zoo is the training framework associated with SB3. It provides scripts for training, evaluating agents, setting hyperparameters, plotting results and recording video. It also contains already optimized hypermeters, including for some `panda-gym` environments.

> **Warning:** The current version of RL Baselines3 Zoo provides hyperparameters for version 1 of `panda-gym`, but not for version 2. Before training with RL Baselines3 Zoo, you will have to set your own hyperparameters by editing `hyperparameters/<ALGO>.yml`. For more information, please read the README of RL Baselines3 Zoo.

6.3.1 Train

To use it, follow the instructions for its installation, then use the following command.

```
python train.py --algo <ALGO> --env <ENV>
```

For example, to train an agent with TQC on `PandaPickAndPlace-v2`:

```
python train.py --algo tqc --env PandaPickAndPlace-v2
```

6.3.2 Enjoy

To visualize the trained agent, follow the instructions in the SB3 documentation. It is necessary to add `--env-kwargs render:True` when running the enjoy script.

```
python enjoy.py --algo <ALGO> --env <ENV> --folder <TRAIN_AGENT_FOLDER> --env-kwargs␣
↪render:True
```

CHAPTER

SEVEN

CUSTOM ROBOT

7.1 Prerequisites

To create your own robot, you will need a URDF file describing the robot.

7.2 Code

To define your own robot, you need to define the following methods and attributes:

- the joint indices,
- the joint forces,
- the `set_action(action)` method,
- the `get_obs()` method and
- the `reset()` method.

For the purpose of the example, let's use a very simple robot, whose URDF file is given below. It consists in two links and a single joint.

```
<?xml version="1.0"?>
<robot name="my_robot">
    <link name="link0"> ... </link>
    <link name="link1"> ... </link>
    <joint name="joint0" type="continuous">
        <parent link="link0" />
        <child link="link1" />
        ...
    </joint>
</robot>
```

7.2.1 Joint indices

The first step is to identify the joints you want to be able to control with the agent. These joints will be identified by their index in the URDF file. Here, it is the index 0 joint that you want to control (it is also the only one in the URDF file). For the following, you will use `joint_indices=np.array([0])`.

7.2.2 Joint forces

For each joint, you must define a maximum force. This data is usually found in the technical specifications of the robot, and sometimes in the URDF file (`<limit effort="1.0"/>` for a maximum effort of 1.0 Nm). Here, let's consider that the maximum effort is 1.0 Nm. For the following, you will use `joint_forces=np.array([1.0])`.

7.2.3 `set_action` method

The `set_action` method specify what the robot must do with the action. In the example, the robot only uses the action as a target angle for its single joint. Thus:

```
def set_action(self, action):
    self.control_joints(target_angles=action)
```

7.2.4 `get_obs` method

The `get_obs` method returns the observation associated with the robot. In the example, the robot only returns the position of it single joint.

```
def get_obs(self):
    return self.get_joint_angle(joint=0)
```

7.2.5 `reset` method

The `reset` method specify how to reset the robot. In the example, the robot resets its single joint to an angle of 0.

```
def reset(self):
    neutral_angle = np.array([0.0])
    self.set_joint_angles(angles=neutral_angle)
```

7.2.6 Full code

You now have everything you need to define your custom robot. You only have to inherit the class *PyBulletRobot* in the following way.

```
import numpy as np
from gym import spaces

from panda_gym.envs.core import PyBulletRobot


class MyRobot(PyBulletRobot):
```

(continues on next page)

(continued from previous page)

```
    """My robot"""

    def __init__(self, sim):
        action_dim = 1 # = number of joints; here, 1 joint, so dimension = 1
        action_space = spaces.Box(-1.0, 1.0, shape=(action_dim,), dtype=np.float32)
        super().__init__(
            sim,
            body_name="my_robot",  # choose the name you want
            file_name="my_robot.urdf",  # the path of the URDF file
            base_position=np.zeros(3),  # the position of the base
            action_space=action_space,
            joint_indices=np.array([0]),  # list of the indices, as defined in the URDF
            joint_forces=np.array([1.0]),  # force applied when robot is controled (Nm)
        )

    def set_action(self, action):
        self.control_joints(target_angles=action)

    def get_obs(self):
        return self.get_joint_angle(joint=0)

    def reset(self):
        neutral_angle = np.array([0.0])
        self.set_joint_angles(angles=neutral_angle)
```

Obviously, you have to adapt the example to your robot, especially concerning the number and the indices of the joints, as well as the forces applied for the control.

You can also use other types of control, using all the methods of the parent class *PyBulletRobot* and the simulation instance *PyBullet*. For example for inverse kinematics you can use the method *PyBulletRobot.inverse_kinematics*.

7.3 Test it

The robot is ready. To see it move, execute the following code.

```
from panda_gym.pybullet import PyBullet

sim = PyBullet(render=True)
robot = MyRobot(sim)

for _ in range(50):
    robot.set_action(np.array([1.0]))
    sim.step()
    sim.render()
```

To see how to use this robot to define a new environment, see the *custom environment* section.

CHAPTER EIGHT

CUSTOM TASK

8.1 Prerequisites

To create your own robot, you will need its URDF file.

8.2 Code

To define your own task, you need to inherit from *Task*, and define the following 5 methods:

- `reset()`: how the task is reset; you must define *self.goal* in this function
- `get_obs()`: returns the observation
- `get_achieved_goal()`: returns the achieved goal
- `is_success(achieved_goal, desired_goal, info)`: returns whether the task is successfull
- `compute_reward(achieved_goal, desired_goal, info)`: returns the reward

For the purpose of the example, let's consider here a very simple task, consisting in moving a cube toward a target position. The goal position is sampled within a volume of 10 m x 10 m x 10 m.

```
import numpy as np

from panda_gym.envs.core import Task
from panda_gym.utils import distance


class MyTask(Task):
    def __init__(self, sim):
        super().__init__(sim)
        # create an cube
        self.sim.create_box(body_name="object", half_extents=np.array([1, 1, 1]), mass=1.
↪0, position=np.array([0.0, 0.0, 0.0]))

    def reset(self):
        # randomly sample a goal position
        self.goal = np.random.uniform(-10, 10, 3)
        # reset the position of the object
        self.sim.set_base_pose("object", position=np.array([0.0, 0.0, 0.0]),␣
↪orientation=np.array([1.0, 0.0, 0.0, 0.0]))

```

(continues on next page)

(continued from previous page)

```
    def get_obs(self):
        # the observation is the position of the object
        observation = self.sim.get_base_position("object")
        return observation

    def get_achieved_goal(self):
        # the achieved goal is the current position of the object
        achieved_goal = self.sim.get_base_position("object")
        return achieved_goal

    def is_success(self, achieved_goal, desired_goal, info={}):  # info is here for␣
↪consistancy
        # compute the distance between the goal position and the current object position
        d = distance(achieved_goal, desired_goal)
        # return 1.0 if the distance is < 1.0, and 0.0 otherwise
        return np.array(d < 1.0, dtype=np.float64)

    def compute_reward(self, achieved_goal, desired_goal, info={}):  # info is here for␣
↪consistancy
        # for this example, reward = 1.0 if the task is successfull, 0.0 otherwise
        return self.is_success(achieved_goal, desired_goal, info)
```

Obviously, you have to adapt the example to your task.

8.3 Test it

The task is ready. To test it, execute the following code.

```
from panda_gym.pybullet import PyBullet

sim = PyBullet(render=True)
task = MyTask(sim)

task.reset()
print(task.get_obs())
print(task.get_achieved_goal())
print(task.is_success(task.get_achieved_goal(), task.get_goal()))
print(task.compute_reward(task.get_achieved_goal(), task.get_goal()))
```

CHAPTER NINE

CUSTOM ENVIRONMENT

A customized environment is the junction of a **task** and a **robot**. You can choose to *define your own task*, or use one of the tasks present in the package. Similarly, you can choose to *define your own robot*, or use one of the robots present in the package.

Then, you have to inherit from the `RobotTaskEnv` class, in the following way.

```
from panda_gym.envs.core import RobotTaskEnv
from panda_gym.pybullet import PyBullet


class MyRobotTaskEnv(RobotTaskEnv):
    """My robot-task environment."""

    def __init__(self, render=False):
        sim = PyBullet(render=render)
        robot = MyRobot(sim)
        task = MyTask(sim)
        super().__init__(robot, task)
```

That's it.

9.1 Test it

You can now test your environment by running the following code.

```
env = MyRobotTaskEnv(render=True)

obs = env.reset()
done = False
while not done:
    action = env.action_space.sample() # random action
    obs, reward, done, info = env.step(action)
    env.render() # wait a bit to give a realistic temporal rendering
```

CHAPTER

TEN

PYBULLET CLASS

class panda_gym.pybullet.**PyBullet**(*render: bool = False*, *n_substeps: int = 20*, *background_color: Optional[ndarray] = None*)

Convenient class to use PyBullet physics engine.

Parameters

- **render** (*bool, optional*) – Enable rendering. Defaults to False.
- **n_substeps** (*int, optional*) – Number of sim substep when step() is called. Defaults to 20.
- **background_color** (*np.ndarray, optional*) – The background color as (red, green, blue). Defaults to np.array([223, 54, 45]).

close() → None

Close the simulation.

control_joints(*body: str*, *joints: ndarray*, *target_angles: ndarray*, *forces: ndarray*) → None

Control the joints motor.

Parameters

- **body** (*str*) – Body unique name.
- **joints** (*np.ndarray*) – List of joint indices, as a list of ints.
- **target_angles** (*np.ndarray*) – List of target angles, as a list of floats.
- **forces** (*np.ndarray*) – Forces to apply, as a list of floats.

create_box(*body_name: str*, *half_extents: ndarray*, *mass: float*, *position: ndarray*, *rgba_color: Optional[ndarray] = None*, *specular_color: Optional[ndarray] = None*, *ghost: bool = False*, *lateral_friction: Optional[float] = None*, *spinning_friction: Optional[float] = None*, *texture: Optional[str] = None*) → None

Create a box.

Parameters

- **body_name** (*str*) – The name of the body. Must be unique in the sim.
- **half_extents** (*np.ndarray*) – Half size of the box in meters, as (x, y, z).
- **mass** (*float*) – The mass in kg.
- **position** (*np.ndarray*) – The position, as (x, y, z).
- **rgba_color** (*np.ndarray, optional*) – Body color, as (r, g, b, a). Defaults as [0, 0, 0, 0]

- **specular_color** (*np.ndarray, optional*) – Specular color, as (r, g, b). Defaults to [0, 0, 0].
- **ghost** (*bool, optional*) – Whether the body can collide. Defaults to False.
- **lateral_friction** (*float or None, optional*) – Lateral friction. If None, use the default pybullet value. Defaults to None.
- **spinning_friction** (*float or None, optional*) – Spinning friction. If None, use the default pybullet value. Defaults to None.
- **texture** (*str or None, optional*) – Texture file name. Defaults to None.

create_cylinder(*body_name: str, radius: float, height: float, mass: float, position: ndarray, rgba_color: Optional[ndarray] = None, specular_color: Optional[ndarray] = None, ghost: bool = False, lateral_friction: Optional[float] = None, spinning_friction: Optional[float] = None*) → None

Create a cylinder.

Parameters

- **body_name** (*str*) – The name of the body. Must be unique in the sim.
- **radius** (*float*) – The radius in meter.
- **height** (*float*) – The height in meter.
- **mass** (*float*) – The mass in kg.
- **position** (*np.ndarray*) – The position, as (x, y, z).
- **rgba_color** (*np.ndarray, optional*) – Body color, as (r, g, b, a). Defaults as [0, 0, 0, 0]
- **specular_color** (*np.ndarray, optional*) – Specular color, as (r, g, b). Defaults to [0, 0, 0].
- **ghost** (*bool, optional*) – Whether the body can collide. Defaults to False.
- **lateral_friction** (*float or None, optional*) – Lateral friction. If None, use the default pybullet value. Defaults to None.
- **spinning_friction** (*float or None, optional*) – Spinning friction. If None, use the default pybullet value. Defaults to None.

create_plane(*z_offset: float*) → None

Create a plane. (Actually, it is a thin box.)

Parameters

z_offset (*float*) – Offset of the plane.

create_sphere(*body_name: str, radius: float, mass: float, position: ndarray, rgba_color: Optional[ndarray] = None, specular_color: Optional[ndarray] = None, ghost: bool = False, lateral_friction: Optional[float] = None, spinning_friction: Optional[float] = None*) → None

Create a sphere.

Parameters

- **body_name** (*str*) – The name of the body. Must be unique in the sim.
- **radius** (*float*) – The radius in meter.
- **mass** (*float*) – The mass in kg.

- **position** (*np.ndarray*) – The position, as (x, y, z).
- **rgba_color** (*np.ndarray, optional*) – Body color, as (r, g, b, a). Defaults as [0, 0, 0, 0]
- **specular_color** (*np.ndarray, optional*) – Specular color, as (r, g, b). Defaults to [0, 0, 0].
- **ghost** (*bool, optional*) – Whether the body can collide. Defaults to False.
- **lateral_friction** (*float or None, optional*) – Lateral friction. If None, use the default pybullet value. Defaults to None.
- **spinning_friction** (*float or None, optional*) – Spinning friction. If None, use the default pybullet value. Defaults to None.

create_table(*length: float*, *width: float*, *height: float*, *x_offset: float = 0.0*, *lateral_friction: Optional[float] = None*, *spinning_friction: Optional[float] = None*) → None

Create a fixed table. Top is z=0, centered in y.

Parameters

- **length** (*float*) – The length of the table (x direction).
- **width** (*float*) – The width of the table (y direction)
- **height** (*float*) – The height of the table.
- **x_offset** (*float, optional*) – The offet in the x direction.
- **lateral_friction** (*float or None, optional*) – Lateral friction. If None, use the default pybullet value. Defaults to None.
- **spinning_friction** (*float or None, optional*) – Spinning friction. If None, use the default pybullet value. Defaults to None.

property dt

Timestep.

get_base_angular_velocity(*body: str*) → ndarray

Get the angular velocity of the body.

Parameters
body (*str*) – Body unique name.

Returns
The angular velocity, as (wx, wy, wz).

Return type
np.ndarray

get_base_orientation(*body: str*) → ndarray

Get the orientation of the body.

Parameters
body (*str*) – Body unique name.

Returns
The orientation, as quaternion (x, y, z, w).

Return type
np.ndarray

get_base_position(*body: str*) → ndarray

Get the position of the body.

Parameters
body (*str*) – Body unique name.

Returns
The position, as (x, y, z).

Return type
np.ndarray

get_base_rotation(*body: str*, *type: str = 'euler'*) → ndarray

Get the rotation of the body.

Parameters

- **body** (*str*) – Body unique name.
- **type** (*str*) – Type of angle, either "euler" or "quaternion"

Returns
The rotation.

Return type
np.ndarray

get_base_velocity(*body: str*) → ndarray

Get the velocity of the body.

Parameters
body (*str*) – Body unique name.

Returns
The velocity, as (vx, vy, vz).

Return type
np.ndarray

get_joint_angle(*body: str*, *joint: int*) → float

Get the angle of the joint of the body.

Parameters

- **body** (*str*) – Body unique name.
- **joint** (*int*) – Joint index in the body

Returns
The angle.

Return type
float

get_joint_velocity(*body: str*, *joint: int*) → float

Get the velocity of the joint of the body.

Parameters

- **body** (*str*) – Body unique name.
- **joint** (*int*) – Joint index in the body

Returns
The velocity.

Return type
float

`get_link_angular_velocity`(*body: str*, *link: int*) → ndarray

Get the angular velocity of the link of the body.

Parameters

- **body** (*str*) – Body unique name.
- **link** (*int*) – Link index in the body.

Returns
The angular velocity, as (wx, wy, wz).

Return type
np.ndarray

`get_link_orientation`(*body: str*, *link: int*) → ndarray

Get the orientation of the link of the body.

Parameters

- **body** (*str*) – Body unique name.
- **link** (*int*) – Link index in the body.

Returns
The rotation, as (rx, ry, rz).

Return type
np.ndarray

`get_link_position`(*body: str*, *link: int*) → ndarray

Get the position of the link of the body.

Parameters

- **body** (*str*) – Body unique name.
- **link** (*int*) – Link index in the body.

Returns
The position, as (x, y, z).

Return type
np.ndarray

`get_link_velocity`(*body: str*, *link: int*) → ndarray

Get the velocity of the link of the body.

Parameters

- **body** (*str*) – Body unique name.
- **link** (*int*) – Link index in the body.

Returns
The velocity, as (vx, vy, vz).

Return type
np.ndarray

`inverse_kinematics`(*body: str*, *link: int*, *position: ndarray*, *orientation: ndarray*) → ndarray

Compute the inverse kinematics and return the new joint state.

Parameters

- **body** (*str*) – Body unique name.
- **link** (*int*) – Link index in the body.
- **position** (*np.ndarray*) – Desired position of the end-effector, as (x, y, z).
- **orientation** (*np.ndarray*) – Desired orientation of the end-effector as quaternion (x, y, z, w).

Returns

The new joint state.

Return type

np.ndarray

`loadURDF`(*body_name: str*, ***kwargs: Any*) → None

Load URDF file.

Parameters

body_name (*str*) – The name of the body. Must be unique in the sim.

`no_rendering`() → Iterator[None]

Disable rendering within this context.

`place_visualizer`(*target_position: ndarray*, *distance: float*, *yaw: float*, *pitch: float*) → None

Orient the camera used for rendering.

Parameters

- **target** (*np.ndarray*) – Target position, as (x, y, z).
- **distance** (*float*) – Distance from the target position.
- **yaw** (*float*) – Yaw.
- **pitch** (*float*) – Pitch.

`remove_state`(*state_id: int*) → None

Remove a simulation state. This will make this state_id available again for returning in save_state().

Parameters

state_id – The simulation state id returned by save_state().

`render`(*mode: str = 'human'*, *width: int = 720*, *height: int = 480*, *target_position: Optional[ndarray] = None*, *distance: float = 1.4*, *yaw: float = 45*, *pitch: float = - 30*, *roll: float = 0*) → Optional[ndarray]

Render.

If mode is "human", make the rendering real-time. All other arguments are unused. If mode is "rgb_array", return an RGB array of the scene.

Parameters

- **mode** (*str*) – "human" of "rgb_array". If "human", this method waits for the time necessary to have a realistic temporal rendering and all other args are ignored. Else, return an RGB array.
- **width** (*int, optional*) – Image width. Defaults to 720.
- **height** (*int, optional*) – Image height. Defaults to 480.

- **target_position** (*np.ndarray, optional*) – Camera targetting this postion, as (x, y, z). Defaults to [0., 0., 0.].
- **distance** (*float, optional*) – Distance of the camera. Defaults to 1.4.
- **yaw** (*float, optional*) – Yaw of the camera. Defaults to 45.
- **pitch** (*float, optional*) – Pitch of the camera. Defaults to -30.
- **roll** (*int, optional*) – Rool of the camera. Defaults to 0.

Returns
An RGB array if mode is 'rgb_array', else None.

Return type
RGB np.ndarray or None

`restore_state`(*state_id: int*) → None

Restore a simulation state.

Parameters
state_id – The simulation state id returned by save_state().

`save_state`() → int

Save the current simulation state.

Returns
A state id assigned by PyBullet, which is the first non-negative integer available for indexing.

Return type
int

`set_base_pose`(*body: str*, *position: ndarray*, *orientation: ndarray*) → None

Set the position of the body.

Parameters

- **body** (*str*) – Body unique name.
- **position** (*np.ndarray*) – The position, as (x, y, z).
- **orientation** (*np.ndarray*) – The target orientation as quaternion (x, y, z, w).

`set_joint_angle`(*body: str*, *joint: int*, *angle: float*) → None

Set the angle of the joint of the body.

Parameters

- **body** (*str*) – Body unique name.
- **joint** (*int*) – Joint index in the body.
- **angle** (*float*) – Target angle.

`set_joint_angles`(*body: str*, *joints: ndarray*, *angles: ndarray*) → None

Set the angles of the joints of the body.

Parameters

- **body** (*str*) – Body unique name.
- **joints** (*np.ndarray*) – List of joint indices, as a list of ints.
- **angles** (*np.ndarray*) – List of target angles, as a list of floats.

set_lateral_friction(*body: str*, *link: int*, *lateral_friction: float*) → None

Set the lateral friction of a link.

Parameters

- **body** (*str*) – Body unique name.
- **link** (*int*) – Link index in the body.
- **lateral_friction** (*float*) – Lateral friction.

set_spinning_friction(*body: str*, *link: int*, *spinning_friction: float*) → None

Set the spinning friction of a link.

Parameters

- **body** (*str*) – Body unique name.
- **link** (*int*) – Link index in the body.
- **spinning_friction** (*float*) – Spinning friction.

step() → None

Step the simulation.

CHAPTER ELEVEN

ROBOT

class panda_gym.envs.core.**PyBulletRobot**(*sim: PyBullet*, *body_name: str*, *file_name: str*, *base_position: ndarray*, *action_space: Space*, *joint_indices: ndarray*, *joint_forces: ndarray*)

Base class for robot env.

Parameters

- **sim** (PyBullet) – Simulation instance.
- **body_name** (*str*) – The name of the robot within the simulation.
- **file_name** (*str*) – Path of the urdf file.
- **base_position** (*np.ndarray*) – Position of the base of the robot as (x, y, z).

control_joints(*target_angles: ndarray*) → None

Control the joints of the robot.

Parameters
target_angles (*np.ndarray*) – The target angles. The length of the array must equal to the number of joints.

get_joint_angle(*joint: int*) → float

Returns the angle of a joint

Parameters
joint (*int*) – The joint index.

Returns
Joint angle

Return type
float

get_joint_velocity(*joint: int*) → float

Returns the velocity of a joint as (wx, wy, wz)

Parameters
joint (*int*) – The joint index.

Returns
Joint velocity as (wx, wy, wz)

Return type
np.ndarray

`get_link_position`(*link: int*) → ndarray

Returns the position of a link as (x, y, z)

Parameters
`link` (*int*) – The link index.

Returns
Position as (x, y, z)

Return type
np.ndarray

`get_link_velocity`(*link: int*) → ndarray

Returns the velocity of a link as (vx, vy, vz)

Parameters
`link` (*int*) – The link index.

Returns
Velocity as (vx, vy, vz)

Return type
np.ndarray

`abstract get_obs`() → ndarray

Return the observation associated to the robot.

Returns
The observation.

Return type
np.ndarray

`inverse_kinematics`(*link: int*, *position: ndarray*, *orientation: ndarray*) → ndarray

Compute the inverse kinematics and return the new joint values.

Parameters

- **link** (*int*) – The link.
- **position** (*x, y, z*) – Desired position of the link.
- **orientation** (*x, y, z, w*) – Desired orientation of the link.

Returns
List of joint values.

`abstract reset`() → None

Reset the robot and return the observation.

`abstract set_action`(*action: ndarray*) → None

Set the action. Must be called just before sim.step().

Parameters
`action` (*np.ndarray*) – The action.

`set_joint_angles`(*angles: ndarray*) → None

Set the joint position of a body. Can induce collisions.

Parameters
`angles` (*list*) – Joint angles.

setup() → None

Called after robot loading.

CHAPTER TWELVE

TASK

class panda_gym.envs.core.**Task**(*sim: PyBullet*)

Base class for tasks. :param sim: Simulation instance. :type sim: PyBullet

abstract compute_reward(*achieved_goal: ndarray*, *desired_goal: ndarray*, *info: Dict[str, Any] = {}*) → Union[ndarray, float]

Compute reward associated to the achieved and the desired goal.

abstract get_achieved_goal() → ndarray

Return the achieved goal.

get_goal() → ndarray

Return the current goal.

abstract get_obs() → ndarray

Return the observation associated to the task.

abstract is_success(*achieved_goal: ndarray*, *desired_goal: ndarray*, *info: Dict[str, Any] = {}*) → Union[ndarray, float]

Returns whether the achieved goal match the desired goal.

abstract reset() → None

Reset the task: sample a new goal.

CHAPTER

THIRTEEN

ROBOT-TASK

class panda_gym.envs.core.**RobotTaskEnv**(*robot: PyBulletRobot*, *task: Task*)

Robotic task goal env, as the junction of a task and a robot.

Parameters

- **robot** (PyBulletRobot) – The robot.
- **task** (Task) – The task.

close() → None

Override close in your subclass to perform any necessary cleanup.

Environments will automatically close() themselves when garbage collected or when the program exits.

render(*mode: str*, *width: int = 720*, *height: int = 480*, *target_position: Optional[ndarray] = None*, *distance: float = 1.4*, *yaw: float = 45*, *pitch: float = - 30*, *roll: float = 0*) → Optional[ndarray]

Render.

If mode is "human", make the rendering real-time. All other arguments are unused. If mode is "rgb_array", return an RGB array of the scene.

Parameters

- **mode** (*str*) – "human" of "rgb_array". If "human", this method waits for the time necessary to have a realistic temporal rendering and all other args are ignored. Else, return an RGB array.
- **width** (*int, optional*) – Image width. Defaults to 720.
- **height** (*int, optional*) – Image height. Defaults to 480.
- **target_position** (*np.ndarray, optional*) – Camera targetting this postion, as (x, y, z). Defaults to [0., 0., 0.].
- **distance** (*float, optional*) – Distance of the camera. Defaults to 1.4.
- **yaw** (*float, optional*) – Yaw of the camera. Defaults to 45.
- **pitch** (*float, optional*) – Pitch of the camera. Defaults to -30.
- **roll** (*int, optional*) – Rool of the camera. Defaults to 0.

Returns

An RGB array if mode is 'rgb_array', else None.

Return type

RGB np.ndarray or None

`reset`(*seed: Optional[int] = None*) → Dict[str, ndarray]

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

This method should also reset the environment's random number generator(s) if *seed* is an integer or if the environment has not yet initialized a random number generator. If the environment already has a random number generator and *reset* is called with *seed=None*, the RNG should not be reset. Moreover, *reset* should (in the typical use case) be called with an integer seed right after initialization and then never again.

Returns
the initial observation. info (optional dictionary): a dictionary containing extra information, this is only returned if return_info is set to true

Return type
observation (object)

`step`(*action: ndarray*) → Tuple[Dict[str, ndarray], float, bool, Dict[str, Any]]

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

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

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

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

Return type
observation (object)

CHAPTER FOURTEEN

PANDA

class panda_gym.envs.robots.panda.**Panda**(*sim: PyBullet*, *block_gripper: bool = False*, *base_position: Optional[ndarray] = None*, *control_type: str = 'ee'*)

Panda robot in PyBullet.

Parameters

- **sim** (PyBullet) – Simulation instance.
- **block_gripper** (*bool, optional*) – Whether the gripper is blocked. Defaults to False.
- **base_position** (*np.ndarray, optionnal*) – Position of the base base of the robot, as (x, y, z). Defaults to (0, 0, 0).
- **control_type** (*str, optional*) – "ee" to control end-effector displacement or "joints" to control joint angles. Defaults to "ee".

arm_joint_ctrl_to_target_arm_angles(*arm_joint_ctrl: ndarray*) → ndarray

Compute the target arm angles from the arm joint control.

Parameters
arm_joint_ctrl (*np.ndarray*) – Control of the 7 joints.

Returns
Target arm angles, as the angles of the 7 arm joints.

Return type
np.ndarray

ee_displacement_to_target_arm_angles(*ee_displacement: ndarray*) → ndarray

Compute the target arm angles from the end-effector displacement.

Parameters
ee_displacement (*np.ndarray*) – End-effector displacement, as (dx, dy, dy).

Returns
Target arm angles, as the angles of the 7 arm joints.

Return type
np.ndarray

get_ee_position() → ndarray

Returns the position of the ned-effector as (x, y, z)

get_ee_velocity() → ndarray

Returns the velocity of the end-effector as (vx, vy, vz)

`get_fingers_width()` → float

Get the distance between the fingers.

`get_obs()` → ndarray

Return the observation associated to the robot.

Returns
The observation.

Return type
np.ndarray

`reset()` → None

Reset the robot and return the observation.

`set_action`(*action: ndarray*) → None

Set the action. Must be called just before sim.step().

Parameters
`action` (*np.ndarray*) – The action.

`set_joint_neutral()` → None

Set the robot to its neutral pose.

CHAPTER

FIFTEEN

CITING

To cite this project in publications:

```
@article{gallouedec2021pandagym,
title        = {{panda-gym: Open-Source Goal-Conditioned Environments for Robotic
↪Learning}},
author       = {Gallou{\'e}dec, Quentin and Cazin, Nicolas and Dellandr{\'e}a, Emmanuel
↪and Chen, Liming},
year         = 2021,
journal      = {4th Robot Learning Workshop: Self-Supervised and Lifelong Learning at
↪NeurIPS},
}
```

CHAPTER

SIXTEEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

p

panda_gym.envs.robots.panda, 37
panda_gym.pybullet, 21

INDEX

A

arm_joint_ctrl_to_target_arm_angles() (*panda_gym.envs.robots.panda.Panda method*), 37

C

close() (*panda_gym.envs.core.RobotTaskEnv method*), 35
close() (*panda_gym.pybullet.PyBullet method*), 21
compute_reward() (*panda_gym.envs.core.Task method*), 33
control_joints() (*panda_gym.envs.core.PyBulletRobot method*), 29
control_joints() (*panda_gym.pybullet.PyBullet method*), 21
create_box() (*panda_gym.pybullet.PyBullet method*), 21
create_cylinder() (*panda_gym.pybullet.PyBullet method*), 22
create_plane() (*panda_gym.pybullet.PyBullet method*), 22
create_sphere() (*panda_gym.pybullet.PyBullet method*), 22
create_table() (*panda_gym.pybullet.PyBullet method*), 23

D

dt (*panda_gym.pybullet.PyBullet property*), 23

E

ee_displacement_to_target_arm_angles() (*panda_gym.envs.robots.panda.Panda method*), 37

G

get_achieved_goal() (*panda_gym.envs.core.Task method*), 33
get_base_angular_velocity() (*panda_gym.pybullet.PyBullet method*), 23
get_base_orientation() (*panda_gym.pybullet.PyBullet method*), 23
get_base_position() (*panda_gym.pybullet.PyBullet method*), 23
get_base_rotation() (*panda_gym.pybullet.PyBullet method*), 24
get_base_velocity() (*panda_gym.pybullet.PyBullet method*), 24
get_ee_position() (*panda_gym.envs.robots.panda.Panda method*), 37
get_ee_velocity() (*panda_gym.envs.robots.panda.Panda method*), 37
get_fingers_width() (*panda_gym.envs.robots.panda.Panda method*), 37
get_goal() (*panda_gym.envs.core.Task method*), 33
get_joint_angle() (*panda_gym.envs.core.PyBulletRobot method*), 29
get_joint_angle() (*panda_gym.pybullet.PyBullet method*), 24
get_joint_velocity() (*panda_gym.envs.core.PyBulletRobot method*), 29
get_joint_velocity() (*panda_gym.pybullet.PyBullet method*), 24
get_link_angular_velocity() (*panda_gym.pybullet.PyBullet method*), 25
get_link_orientation() (*panda_gym.pybullet.PyBullet method*), 25
get_link_position() (*panda_gym.envs.core.PyBulletRobot method*), 29
get_link_position() (*panda_gym.pybullet.PyBullet method*), 25
get_link_velocity() (*panda_gym.envs.core.PyBulletRobot method*), 30
get_link_velocity() (*panda_gym.pybullet.PyBullet method*), 25
get_obs() (*panda_gym.envs.core.PyBulletRobot method*), 30
get_obs() (*panda_gym.envs.core.Task method*), 33
get_obs() (*panda_gym.envs.robots.panda.Panda method*), 38

I

inverse_kinematics() (*panda_gym.envs.core.PyBulletRobot method*), 30
inverse_kinematics() (*panda_gym.pybullet.PyBullet method*), 25
is_success() (*panda_gym.envs.core.Task method*), 33

L

loadURDF() (*panda_gym.pybullet.PyBullet method*), 26

M

module
    panda_gym.envs.robots.panda, 37
    panda_gym.pybullet, 21

N

no_rendering() (*panda_gym.pybullet.PyBullet method*), 26

P

Panda (*class in panda_gym.envs.robots.panda*), 37
panda_gym.envs.robots.panda
    module, 37
panda_gym.pybullet
    module, 21
place_visualizer() (*panda_gym.pybullet.PyBullet method*), 26
PyBullet (*class in panda_gym.pybullet*), 21
PyBulletRobot (*class in panda_gym.envs.core*), 29

R

remove_state() (*panda_gym.pybullet.PyBullet method*), 26
render() (*panda_gym.envs.core.RobotTaskEnv method*), 35
render() (*panda_gym.pybullet.PyBullet method*), 26
reset() (*panda_gym.envs.core.PyBulletRobot method*), 30
reset() (*panda_gym.envs.core.RobotTaskEnv method*), 35
reset() (*panda_gym.envs.core.Task method*), 33
reset() (*panda_gym.envs.robots.panda.Panda method*), 38
restore_state() (*panda_gym.pybullet.PyBullet method*), 27
RobotTaskEnv (*class in panda_gym.envs.core*), 35

S

save_state() (*panda_gym.pybullet.PyBullet method*), 27
set_action() (*panda_gym.envs.core.PyBulletRobot method*), 30
set_action() (*panda_gym.envs.robots.panda.Panda method*), 38
set_base_pose() (*panda_gym.pybullet.PyBullet method*), 27
set_joint_angle() (*panda_gym.pybullet.PyBullet method*), 27
set_joint_angles() (*panda_gym.envs.core.PyBulletRobot method*), 30
set_joint_angles() (*panda_gym.pybullet.PyBullet method*), 27
set_joint_neutral() (*panda_gym.envs.robots.panda.Panda method*), 38
set_lateral_friction() (*panda_gym.pybullet.PyBullet method*), 27
set_spinning_friction() (*panda_gym.pybullet.PyBullet method*), 28
setup() (*panda_gym.envs.core.PyBulletRobot method*), 30
step() (*panda_gym.envs.core.RobotTaskEnv method*), 36
step() (*panda_gym.pybullet.PyBullet method*), 28

T

Task (*class in panda_gym.envs.core*), 33