panda-gym

Release v3.0.5

Quentin Gallouédec

Apr 27, 2023

GETTING STARTED

CHAPTER

ONE

INSTALLATION

panda-gym is at least compatible with Windows, MacOS and Ubuntu, and python 3.7, 3.8, 3.9 and 3.10.

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 gymnasium as gym
import panda_gym

env = gym.make('PandaReach-v3', render_mode="human")

observation, info = env.reset()

for _ in range(1000):
    action = env.action_space.sample() # random action
    observation, reward, terminated, truncated, info = env.step(action)

    if terminated or truncated:
        observation, info = env.reset()
```

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-v3`
- `PandaPush-v3`
- `PandaSlide-v3`
- `PandaPickAndPlace-v3`
- `PandaStack-v3`
- `PandaFlip-v3`

3.2 Dense reward, end-effector control

- PandaReachDense-v3
- PandaPushDense-v3
- PandaSlideDense-v3
- PandaPickAndPlaceDense-v3
- PandaStackDense-v3
- PandaFlipDense-v3

3.3 Sparce reward, joints control

- PandaReachJoints-v3
- PandaPushJoints-v3
- PandaSlideJoints-v3
- PandaPickAndPlaceJoints-v3
- PandaStackJoints-v3
- PandaFlipJoints-v3

3.4 Dense reward, joints control

- PandaReachJointsDense-v3
- PandaPushJointsDense-v3
- PandaSlideJointsDense-v3
- PandaPickAndPlaceJointsDense-v3
- PandaStackJointsDense-v3
- PandaFlipJointsDense-v3

CHAPTER

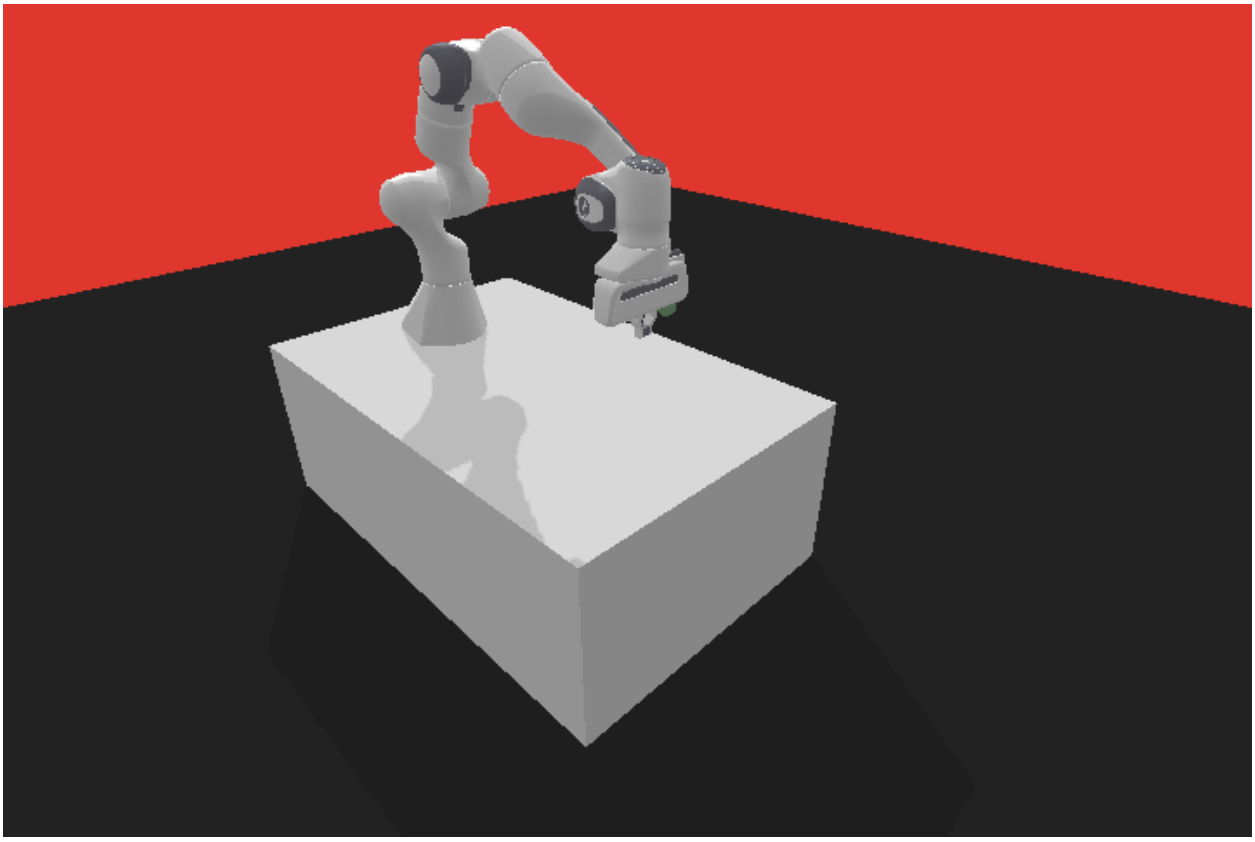
FOUR

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 gymnasium as gym
import panda_gym

env = gym.make("PandaReach-v3", render_mode="human")
observation, info = env.reset()

for _ in range(1000):
    current_position = observation["observation"][0:3]
    desired_position = observation["desired_goal"][0:3]
    action = 5.0 * (desired_position - current_position)
    observation, reward, terminated, truncated, info = env.step(action)

    if terminated or truncated:
        observation, info = env.reset()

env.close()
```

The result is as follows.



CHAPTER

FIVE

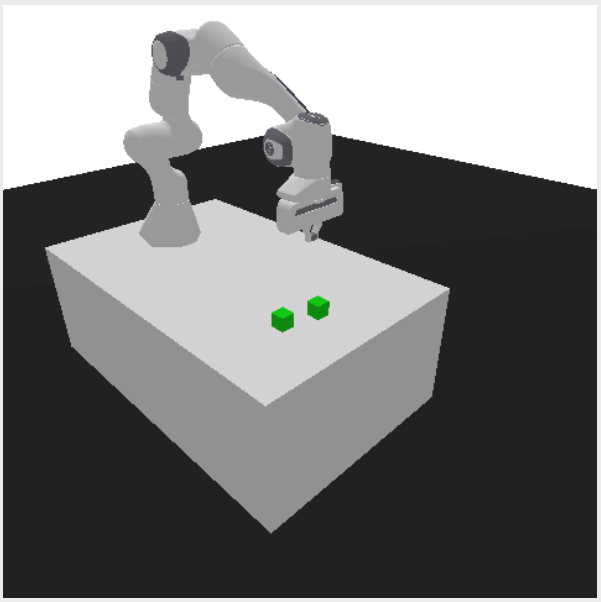
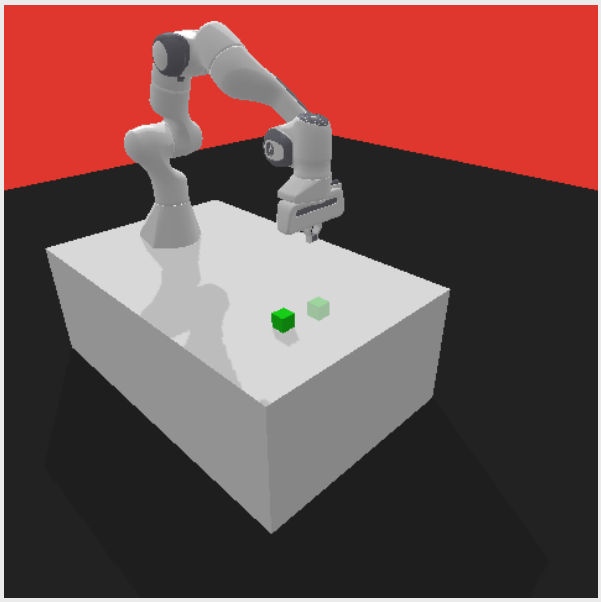
ADVANCED RENDERING

5.1 Renderer

There are two render modes available - "human" and "rgb_array". The "human" mode opens a window to display the live scene, while the "rgb_array" mode renders the scene as an RGB array.

When it comes to renderers, there are two options: OpenGL and Tiny Renderer. The OpenGL engine is used when the render mode is set to "human". However, when the render mode is set to "rgb_array" you have the choice between using either the OpenGL or Tiny Renderer engine. The Tiny Renderer can work in headless mode, but it may not produce as high-quality results as the OpenGL engine, particularly in regards to transparency, background and shadows. If headless mode is not a requirement, it is recommended to use the OpenGL engine. To do this, pass renderer="OpenGL" to the gymnasium.make function:

```
import gymnasium as gym
import panda_gym

env = gym.make("PandaReach-v3", render_mode="rgb_array", renderer="OpenGL")
env.reset()
image = env.render()  # RGB rendering of shape (480, 720, 3)
env.close()
```

renderer="Tiny"	renderer="OpenGL"
	

5.2 Viewpoint

You can render from a different point of view than the default. For this, the following arguments are available:

- `render_width` (int, optional): Image width. Defaults to 720.
- `render_height` (int, optional): Image height. Defaults to 480.
- `render_target_position` (np.ndarray, optional): Camera targetting this postion, as (x, y, z). Defaults to [0., 0., 0.].
- `render_distance` (float, optional): Distance of the camera. Defaults to 1.4.
- `render_yaw` (float, optional): Yaw of the camera. Defaults to 45.
- `render_pitch` (float, optional): Pitch of the camera. Defaults to -30.
- `render_roll` (int, optional): Rool of the camera. Defaults to 0.

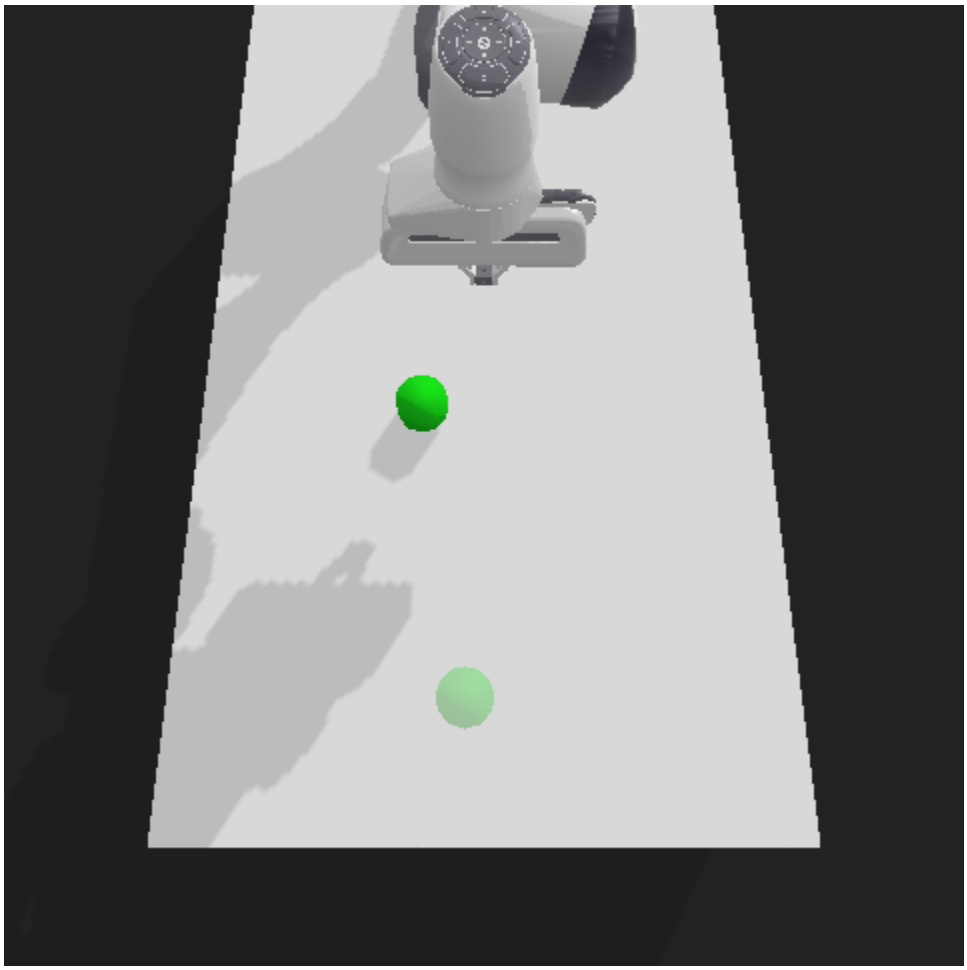
5.2.1 Example

```
import gymnasium as gym
import panda_gym


env = gym.make(
    "PandaSlide-v3",
    render_mode="rgb_array",
    renderer="OpenGL",
    render_width=480,
    render_height=480,
```

(continues on next page)

(continued from previous page)

```
    render_target_position=[0.2, 0, 0],
    render_distance=1.0,
    render_yaw=90,
    render_pitch=-70,
    render_roll=0,
)
env.reset()
image = env.render()  # RGB rendering of shape (480, 480, 3)
env.close()
```



CHAPTER

SIX

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 gymnasium as gym
import numpy as np

import panda_gym

env = gym.make("PandaReachDense-v3", render_mode="human")
observation, _ = env.reset()

for _ in range(1000):
    state_id = env.save_state()

    # Sample 5 actions and choose the one that yields the best reward.
    best_reward = -np.inf
    best_action = None
    for _ in range(5):
        env.restore_state(state_id)
        action = env.action_space.sample()
        observation, reward, _, _, _ = env.step(action)
        if reward > best_reward:
            best_reward = reward
            best_action = action

    env.restore_state(state_id)
    env.remove_state(state_id)  # discard the state, as it is no longer needed

    # Step with the best action
    observation, reward, terminated, truncated, info = env.step(best_action)

    if terminated:
        observation, info = env.reset()

env.close()
```

CHAPTER

SEVEN

TRAIN WITH STABLE-BASELINES3

Warning: SB3 is not compatible with `panda-gym` v3 for the moment. (See SB3/PR#780). The following documentation is therefore not yet valid. To use `panda-gym` with SB3, you will have to use `panda-gym==2.0.0`.

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

7.1 Install SB3

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

7.2 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.

```
import gym
import panda_gym
from stable_baselines3 import DDPG

env = gym.make("PandaReach-v2")
model = DDPG(policy="MultiInputPolicy", env=env)
model.train(30_000)
```

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

7.3 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.

7.3.1 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-v3`:

```
python train.py --algo tqc --env PandaPickAndPlace-v3
```

7.3.2 Enjoy

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

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

CHAPTER

EIGHT

CUSTOM ROBOT

8.1 Prerequisites

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

8.2 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>
```

8.2.1 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])`.

8.2.2 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])`.

8.2.3 `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)
```

8.2.4 `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)
```

8.2.5 `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)
```

8.2.6 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 gymnasium 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*.

8.3 Test it

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

```
from panda_gym.pybullet import PyBullet

sim = PyBullet(render_mode="human")
robot = MyRobot(sim)

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

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

CHAPTER

NINE

CUSTOM TASK

9.1 Prerequisites

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

9.2 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 True if the distance is < 1.0, and False otherwise
        return np.array(d < 1.0, dtype=bool)

    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).astype(np.float32)
```

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

9.3 Test it

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

```
from panda_gym.pybullet import PyBullet

sim = PyBullet(render_mode="human")
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

TEN

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_mode):
        sim = PyBullet(render_mode=render_mode)
        robot = MyRobot(sim)
        task = MyTask(sim)
        super().__init__(robot, task)
```

That's it.

10.1 Test it

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

```
env = MyRobotTaskEnv(render_mode="human")

observation, info = env.reset()

for _ in range(1000):
    action = env.action_space.sample() # random action
    observation, reward, terminated, truncated, info = env.step(action)

    if terminated or truncated:
        observation, info = env.reset()
```

CHAPTER

ELEVEN

PYBULLET CLASS

class panda_gym.pybullet.**PyBullet**(*render_mode: str = 'rgb_array'*, *n_substeps: int = 20*, *background_color: ndarray | None = None*, *renderer: str = 'Tiny'*)

Convenient class to use PyBullet physics engine.

Parameters

- **render_mode** (*str, optional*) – Render mode. Defaults to "rgb_array".
- **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]).
- **renderer** (*str, optional*) – Renderer, either "Tiny" or OpenGL". Defaults to "Tiny" if render mode is "human" and "OpenGL" if render mode is "rgb_array". Only "OpenGL" is available for human render mode.

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: ndarray | None = None*, *specular_color: ndarray | None = None*, *ghost: bool = False*, *lateral_friction: float | None = None*, *spinning_friction: float | None = None*, *texture: str | None = 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: ndarray | None = None*, *specular_color: ndarray | None = None*, *ghost: bool = False*, *lateral_friction: float | None = None*, *spinning_friction: float | None = 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: ndarray | None = None*, *specular_color: ndarray | None = None*, *ghost: bool = False*, *lateral_friction: float | None = None*, *spinning_friction: float | None = 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: float | None = None*, *spinning_friction: float | None = 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
np.ndarray – The angular velocity, as (wx, wy, wz).

get_base_orientation(*body: str*) → ndarray

Get the orientation of the body.

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

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

get_base_position(*body: str*) → ndarray

Get the position of the body.

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

Returns
np.ndarray – The position, as (x, y, z).

`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
np.ndarray – The rotation.

`get_base_velocity`(*body: str*) → ndarray
Get the velocity of the body.

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

Returns
np.ndarray – The velocity, as (vx, vy, vz).

`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
float – The angle.

`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
float – The velocity.

`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
np.ndarray – The angular velocity, as (wx, wy, wz).

`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
np.ndarray – The rotation, as (rx, ry, rz).

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
np.ndarray – The position, as (x, y, z).

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
np.ndarray – The velocity, as (vx, vy, vz).

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
np.ndarray – The new joint state.

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(*width: int = 720*, *height: int = 480*, *target_position: ndarray | None = None*, *distance: float = 1.4*, *yaw: float = 45*, *pitch: float = -30*, *roll: float = 0*) → ndarray | None

Render.

If render mode is "rgb_array", return an RGB array of the scene. Else, do nothing and return None.

Parameters

- **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.
- **mode** (*str, optional*) – Deprecated: This argument is deprecated and will be removed in a future version. Use the render_mode argument of the constructor instead.

Returns

RGB np.ndarray or None – An RGB array if mode is 'rgb_array', else 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

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

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

TWELVE

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
float – Joint angle

get_joint_velocity(*joint: int*) → float

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

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

Returns
np.ndarray – Joint velocity as (wx, wy, wz)

get_link_position(*link: int*) → ndarray

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

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

Returns
np.ndarray – Position as (x, y, z)

get_link_velocity(*link: int*) → ndarray

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

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

Returns
np.ndarray – Velocity as (vx, vy, vz)

abstract get_obs() → ndarray

Return the observation associated to the robot.

Returns
np.ndarray – The observation.

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

THIRTEEN

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] = {}*) → ndarray

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] = {}*) → ndarray

Returns whether the achieved goal match the desired goal.

abstract reset() → None

Reset the task: sample a new goal.

CHAPTER

FOURTEEN

ROBOT-TASK

class panda_gym.envs.core.**RobotTaskEnv**(*robot: PyBulletRobot*, *task: Task*, *render_width: int = 720*, *render_height: int = 480*, *render_target_position: ndarray | None = None*, *render_distance: float = 1.4*, *render_yaw: float = 45*, *render_pitch: float = -30*, *render_roll: float = 0*)

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

Parameters

- **robot** (PyBulletRobot) – The robot.
- **task** (Task) – The task.
- **render_width** (*int, optional*) – Image width. Defaults to 720.
- **render_height** (*int, optional*) – Image height. Defaults to 480.
- **render_target_position** (*np.ndarray, optional*) – Camera targetting this postion, as (x, y, z). Defaults to [0., 0., 0.].
- **render_distance** (*float, optional*) – Distance of the camera. Defaults to 1.4.
- **render_yaw** (*float, optional*) – Yaw of the camera. Defaults to 45.
- **render_pitch** (*float, optional*) – Pitch of the camera. Defaults to -30.
- **render_roll** (*int, optional*) – Rool of the camera. Defaults to 0.

close() → None

After the user has finished using the environment, close contains the code necessary to "clean up" the environment.

This is critical for closing rendering windows, database or HTTP connections.

remove_state(*state_id: int*) → None

Remove a saved state.

Parameters
state_id (*int*) – State unique identifier.

render() → ndarray | None

Render.

If render mode is "rgb_array", return an RGB array of the scene. Else, do nothing and return None.

Returns
RGB np.ndarray or None – An RGB array if mode is 'rgb_array', else None.

reset(*seed: int | None = None*, *options: dict | None = None*) → Tuple[Dict[str, ndarray], Dict[str, Any]]

Resets the environment to an initial internal state, returning an initial observation and info.

This method generates a new starting state often with some randomness to ensure that the agent explores the state space and learns a generalised policy about the environment. This randomness can be controlled with the `seed` parameter otherwise if the environment already has a random number generator and *reset()* is called with `seed=None`, the RNG is not reset.

Therefore, *reset()* should (in the typical use case) be called with a seed right after initialization and then never again.

For Custom environments, the first line of *reset()* should be `super().reset(seed=seed)` which implements the seeding correctly.

Changed in version v0.25: The `return_info` parameter was removed and now info is expected to be returned.

Parameters

- **seed** (*optional int*) – The seed that is used to initialize the environment's PRNG (*np_random*). If the environment does not already have a PRNG and `seed=None` (the default option) is passed, a seed will be chosen from some source of entropy (e.g. timestamp or /dev/urandom). However, if the environment already has a PRNG and `seed=None` is passed, the PRNG will *not* be reset. If you pass an integer, the PRNG will be reset even if it already exists. Usually, you want to pass an integer *right after the environment has been initialized and then never again*. Please refer to the minimal example above to see this paradigm in action.
- **options** (*optional dict*) – Additional information to specify how the environment is reset (optional, depending on the specific environment)

Returns

- **observation** (*ObsType*) – Observation of the initial state. This will be an element of `observation_space` (typically a numpy array) and is analogous to the observation returned by *step()*.
- **info** (*dictionary*) – This dictionary contains auxiliary information complementing `observation`. It should be analogous to the `info` returned by *step()*.

restore_state(*state_id: int*) → None

Resotre the state associated with the unique identifier.

Parameters

state_id (*int*) – State unique identifier.

save_state() → int

Save the current state of the envrionment. Restore with *restore_state*.

Returns

int – State unique identifier.

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

Run one timestep of the environment's dynamics using the agent actions.

When the end of an episode is reached (`terminated or truncated`), it is necessary to call *reset()* to reset this environment's state for the next episode.

Changed in version 0.26: The Step API was changed removing `done` in favor of `terminated` and `truncated` to make it clearer to users when the environment had terminated or truncated which is critical for reinforcement learning bootstrapping algorithms.

Parameters

action (*ActType*) – an action provided by the agent to update the environment state.

Returns

- **observation** (*ObsType*) – An element of the environment's `observation_space` as the next observation due to the agent actions. An example is a numpy array containing the positions and velocities of the pole in CartPole.
- **reward** (*SupportsFloat*) – The reward as a result of taking the action.
- **terminated** (*bool*) – Whether the agent reaches the terminal state (as defined under the MDP of the task) which can be positive or negative. An example is reaching the goal state or moving into the lava from the Sutton and Barton, Gridworld. If true, the user needs to call *reset()*.
- **truncated** (*bool*) – Whether the truncation condition outside the scope of the MDP is satisfied. Typically, this is a timelimit, but could also be used to indicate an agent physically going out of bounds. Can be used to end the episode prematurely before a terminal state is reached. If true, the user needs to call *reset()*.
- **info** (*dict*) – Contains auxiliary diagnostic information (helpful for debugging, learning, and logging). This might, for instance, contain: metrics that describe the agent's performance state, variables that are hidden from observations, or individual reward terms that are combined to produce the total reward. In OpenAI Gym <v26, it contains "TimeLimit.truncated" to distinguish truncation and termination, however this is deprecated in favour of returning terminated and truncated variables.
- **done** (*bool*) – (Deprecated) A boolean value for if the episode has ended, in which case further *step()* calls will return undefined results. This was removed in OpenAI Gym v26 in favor of terminated and truncated attributes. A done signal may be emitted for different reasons: Maybe the task underlying the environment was solved successfully, a certain timelimit was exceeded, or the physics simulation has entered an invalid state.

CHAPTER

FIFTEEN

PANDA

class panda_gym.envs.robots.panda.**Panda**(*sim: PyBullet*, *block_gripper: bool = False*, *base_position: ndarray | None = 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
np.ndarray – Target arm angles, as the angles of the 7 arm joints.

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
np.ndarray – Target arm angles, as the angles of the 7 arm joints.

get_ee_position() → ndarray

Returns the position of the end-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
np.ndarray – The observation.

`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

SIXTEEN

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

SEVENTEEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

p

INDEX

I

L

M

N

P

R

S

T