AIRobot Documentation

Release 0.1

Tao Chen; Anthony Simeonov; Pulkit Agrawal

Sep 10, 2022

Introduction:

CHAPTER 1

AIRobot

AIRobot is a python library to interface with robots. It follows a similar architecture from PyRobot.

1.1 Citation

If you use AIRobot in your research, please use the following BibTeX entry.

```
@misc{airobot2019,
  author={Tao Chen and Anthony Simeonov and Pulkit Agrawal},
  title={{AIRobot}},
  publisher={GitHub},
  journal={GitHub repository},
  howpublished={\url{https://github.com/Improbable-AI/airobot}},
  year={2019}
}
```

1.2 Installation

1.2.1 Pre-installation steps for ROS (only required if you want to use the real robot)

Option 1:

If you want to use ROS to interface with robots, please install ROS Kinetic first.

If you want to use ROS for UR5e robots, please install ur5e driver.

If you are using RealSense cameras, please follow the instructions here or here.

Option 2:

Use the dockerfile provided in this library. Checkout the instructions for docker usage. **Docker is not needed if you just want to use the pybullet simulation environment.**

1.2.2 Install AIRobot

You might want to install it in a virtual environment.

If you are using ROS, you can use virtualenv. Note that Anaconda doesn't work well with ROS. And only Python 2.7 is recommended for ROS at this point.

If you only want to use the robots in the PyBullet simulation environment, then you can use Python 2.7 or Python 3.7. And you can use either virtualenv for Python 2.7, venv for Python 3.7 or Anaconda.

```
git clone https://github.com/Improbable-AI/airobot.git
cd airobot
pip install -e .
```

1.3 Supported Robots

- UR5e with ROS
- UR5e in PyBullet
- ABB YuMi in PyBullet
- Franka Robot in Pybullet

If you want to use Sawyer or LoCoBot, you can use PyRobot.

1.4 Getting Started

A sample script is shown below. You can find more examples here

```
from airobot import Robot
# create a UR5e robot in pybullet
robot = Robot('ur5e',
              pb=True,
              pb_cfg={'gui': True})
```

(continues on next page)

(continued from previous page)

```
robot.arm.go_home()
robot.arm.move_ee_xyz([0.1, 0.1, 0.1])
```

1.5 Credits

AIRobot is maintained by the Improbable AI team at MIT. It follows a similar architecture in PyRobot. Contributors include:

- Tao Chen
- Anthony Simeonov
- Pulkit Agrawal

1.6 Build API Docs

Run the following commands to build the API webpage.

```
cd docs
./make_api.sh
```

Then you can use any web browser to open the API doc (**docs/build/html/index.html**)

1.7 Run tests

pytest is used for unit tests. TODO: more test cases need to be added.

```
cd airobot/tests
./run_pytest.sh
```

1.8 License

MIT license

1.9 Coding Style

AIRobot uses Google style for formatting docstrings. We use Flake8 to perform additional formatting and semantic checking of code.

1.10 Acknowledgement

We gratefully acknowledge the support from Sony Research Grant and DARPA Machine Common Sense Program.

CHAPTER 2

AIRobot API

2.1 airobot.arm

2.1.1 airobot.arm.arm

class airobot.arm.arm.**ARM**(*cfgs*, *eetool_cfg=None*)

Bases: object

Base class for robots.

Parameters

- **cfgs** (*YACS CfgNode*) – configurations for the robot
- **eetool_cfg** (*dict*) – arguments to pass in the constructor of the end effector tool class. Defaults to None

Variables

- **cfgs** (*YACS CfgNode*) – configurations for the robot
- **eetool** (EndEffectorTool) – end effector tool

compute_ik(*pos*, *ori=None*, **args*, ***kwargs*)

Compute the inverse kinematics solution given the position and orientation of the end effector.

Parameters

- **pos** (*list or np.ndarray*) – position (shape: $[3,]$)
- **ori** (*list or np.ndarray*) – orientation. It can be euler angles ([roll, pitch, yaw], shape: $[4,]$), or quaternion ([qx, qy, qz, qw], shape: $[4,]$), or rotation matrix (shape: $[3, 3]$). If it's None, the solver will use the current end effector orientation as the target orientation

Returns *list* – inverse kinematics solution (joint angles)

get_ee_pose()

Return the end effector pose.

Returns

4-element tuple containing

- np.ndarray: x, y, z position of the EE (shape: $[3]$)
- np.ndarray: quaternion representation ([x, y, z, w]) of the EE orientation (shape: $[4]$)
- np.ndarray: rotation matrix representation of the EE orientation (shape: $[3, 3]$)
- np.ndarray: euler angle representation of the EE orientation (roll, pitch, yaw with static reference frame) (shape: $[3]$)

get_jpos(*joint_name=None*)
Return the joint position(s).

Parameters **joint_name** (*str*) – If it's None, it will return joint positions of all the actuated joints. Otherwise, it will return the joint position of the specified joint

Returns

One of the following

- float: joint position given joint_name
- list: joint positions if joint_name is None

get_jtorq(*joint_name=None*)
Return the joint torque(s).

Parameters **joint_name** (*str*) – If it's None, it will return joint torques of all the actuated joints. Otherwise, it will return the joint torque of the specified joint

Returns

One of the following

- float: joint torque given joint_name
- list: joint torques if joint_name is None

get_jvel(*joint_name=None*)
Return the joint velocity(ies).

Parameters **joint_name** (*str*) – If it's None, it will return joint velocities of all the actuated joints. Otherwise, it will return the joint velocity of the specified joint

Returns

One of the following

- float: joint velocity given joint_name
- list: joint velocities if joint_name is None

go_home()
Move the robot arm to a pre-defined home pose.

move_ee_xyz(*delta_xyz*, *eef_step=0.005*, **args*, ***kwargs*)
Move the end-effector in a straight line without changing the orientation.

Parameters

- **delta_xyz** (*list or np.ndarray*) – movement in x, y, z directions
- **eef_step** (*float*) – interpolation interval along delta_xyz. Interpolate a point every eef_step distance between the two end points

Returns *bool* – A boolean variable representing if the action is successful at the moment when the function exits

set_ee_pose(*pos=None*, *ori=None*, **args*, ***kwargs*)
Move the end effector to the specifed pose.

Parameters

- **pos** (*list or np.ndarray*) – Desired x, y, z positions in the robot's base frame to move to (shape: $[3,]$)
- **ori** (*list or np.ndarray, optional*) – It can be euler angles ([roll, pitch, yaw], shape: $[4,]$), or quaternion ([qx, qy, qz, qw], shape: $[4,]$), or rotation matrix (shape: $[3, 3]$). If it's None, the solver will use the current end effector orientation as the target orientation

Returns *bool* – A boolean variable representing if the action is successful at the moment when the function exits

set_jpos(*position*, *joint_name=None*, **args*, ***kwargs*)
Move the arm to the specified joint position(s).

Parameters

- **position** (*float or list or flattened np.ndarray*) – desired joint position(s)
- **joint_name** (*str*) – If not provided, position should be a list and all the actuated joints will be moved to the specified positions. If provided, only the specified joint will be moved to the desired joint position
- **wait** (*bool*) – whether to block the code and wait for the action to complete

Returns *bool* – A boolean variable representing if the action is successful at the moment when the function exits

set_jtorq(*torque*, *joint_name=None*, **args*, ***kwargs*)
Apply torque(s) to the joint(s).

Parameters

- **torque** (*float or list or flattened np.ndarray*) – torque value(s) for the joint(s)
- **joint_name** (*str*) – specify the joint on which the torque is applied. If it's not provided(None), it will apply the torques on the actuated joints on the arm. Otherwise, only the specified joint will be applied with the given torque.
- **wait** (*bool*) – whether to block the code and wait for the action to complete

Returns *bool* – A boolean variable representing if the action is successful at the moment when the function exits

set_jvel(*velocity*, *joint_name=None*, **args*, ***kwargs*)
Move the arm with the specified joint velocity(ies).

Parameters

- **velocity** (*float or list or flattened np.ndarray*) – desired joint velocity(ies)
- **joint_name** (*str*) – If not provided, velocity should be a list and all the actuated joints will be moved in the specified velocities. If provided, only the specified joint will be moved in the desired joint velocity

- **wait** (*bool*) – whether to block the code and wait for the action to complete

Returns *bool* – A boolean variable representing if the action is successful at the moment when the function exits

2.1.2 airobot.arm.single_arm_pybullet

class airobot.arm.single_arm_pybullet.**SingleArmPybullet**(*cfgs*, *pb_client*, *seed=None*, *self_collision=False*, *eetool_cfg=None*)

Bases: *airobot.arm.arm.ARM*

Base class for a single arm simulated in pybullet.

Parameters

- **cfgs** (*YACS CfgNode*) – configurations for the arm.
- **pb_client** (BulletClient) – pybullet client.
- **seed** (*int*) – random seed.
- **self_collision** (*bool*) – enable self_collision or not whiling loading URDF.
- **eetool_cfg** (*dict*) – arguments to pass in the constructor of the end effector tool class.

Variables

- **cfgs** (*YACS CfgNode*) – configurations for the arm.
- **robot_id** (*int*) – pybullet body unique id of the robot.
- **arm_jnt_names** (*list*) – names of the arm joints.
- **arm_dof** (*int*) – degrees of freedom of the arm.
- **arm_jnt_ids** (*list*) – pybullet joint ids of the arm joints.
- **arm_jnt_ik_ids** (*list*) – joint indices of the non-fixed arm joints.
- **ee_link_jnt** (*str*) – joint name of the end-effector link.
- **ee_link_id** (*int*) – joint id of the end-effector link.
- **jnt_to_id** (*dict*) – dictionary with [joint name : pybullet joint] id [key : value] pairs.
- **non_fixed_jnt_names** (*list*) – names of non-fixed joints in the arms, used for returning the correct inverse kinematics solution.

compute_ik(*pos*, *ori=None*, *ns=False*, **args*, ***kwargs*)

Compute the inverse kinematics solution given the position and orientation of the end effector.

Parameters

- **pos** (*list or np.ndarray*) – position (shape: $[3,]$).
- **ori** (*list or np.ndarray*) – orientation. It can be euler angles ([roll, pitch, yaw], shape: $[3,]$), or quaternion ([qx, qy, qz, qw], shape: $[4,]$), or rotation matrix (shape: $[3, 3]$).
- **ns** (*bool*) – whether to use the nullspace options in pybullet, True if nullspace should be used. Defaults to False.

Returns *list* – solution to inverse kinematics, joint angles which achieve the specified EE pose (shape: $[DOF]$).

disable_torque_control(*joint_name=None*)
Disable the torque control mode in Pybullet.

Parameters **joint_name** (*str*) – If it's none, then all the six joints on the UR robot are disabled with torque control. Otherwise, only the specified joint is disabled with torque control. The joint(s) will enter velocity control mode.

enable_torque_control(*joint_name=None*)
Enable the torque control mode in Pybullet.

Parameters **joint_name** (*str*) – If it's none, then all the six joints on the UR robot are enabled in torque control mode. Otherwise, only the specified joint is enabled in torque control mode.

get_ee_pose()
Return the end effector pose.

Returns

4-element tuple containing

- np.ndarray: x, y, z position of the EE (shape: $[3,]$).
- np.ndarray: quaternion representation of the EE orientation (shape: $[4,]$).
- np.ndarray: rotation matrix representation of the EE orientation (shape: $[3, 3]$).
- np.ndarray: euler angle representation of the EE orientation (roll, pitch, yaw with static reference frame) (shape: $[3,]$).

get_ee_vel()
Return the end effector's velocity.

Returns

2-element tuple containing

- np.ndarray: translational velocity (shape: $[3,]$).
- np.ndarray: rotational velocity (shape: $[3,]$).

get_jpos(*joint_name=None*)
Return the joint position(s) of the arm.

Parameters **joint_name** (*str, optional*) – If it's None, it will return joint positions of all the actuated joints. Otherwise, it will return the joint position of the specified joint.

Returns

One of the following

- float: joint position given joint_name.
- list: joint positions if joint_name is None (shape: $[DOF]$).

get_jtorq(*joint_name=None*)
If the robot is operated in VELOCITY_CONTROL or POSITION_CONTROL mode, return the joint torque(s) applied during the last simulation step. In TORQUE_CONTROL, the applied joint motor torque is exactly what you provide, so there is no need to report it separately. So don't use this method to get the joint torque values when the robot is in TORQUE_CONTROL mode.

Parameters **joint_name** (*str, optional*) – If it's None, it will return joint torques of all the actuated joints. Otherwise, it will return the joint torque of the specified joint.

Returns

One of the following

- float: joint torque given joint_name
- list: joint torques if joint_name is None (shape: $[DOF]$).

get_jvel (*joint_name=None*)
Return the joint velocity(ies) of the arm.

Parameters **joint_name** (*str, optional*) – If it's None, it will return joint velocities of all the actuated joints. Otherwise, it will return the joint velocity of the specified joint.

Returns

One of the following

- float: joint velocity given joint_name.
- list: joint velocities if joint_name is None (shape: $[DOF]$).

go_home (*ignore_physics=False*)
Move the robot to a pre-defined home pose

move_ee_xyz (*delta_xyz*, *eef_step=0.005*, **args*, ***kwargs*)
Move the end-effector in a straight line without changing the orientation.

Parameters

- **delta_xyz** (*list or np.ndarray*) – movement in x, y, z directions (shape: $[3,]$).
- **eef_step** (*float*) – interpolation interval along delta_xyz. Interpolate a point every eef_step distance between the two end points.

Returns *bool* – A boolean variable representing if the action is successful at the moment when the function exits.

reset ()
Reset the simulation environment.

reset_joint_state (*jnt_name*, *jpos*, *jvel=0*)
Reset the state of the joint. It's best only to do this at the start, while not running the simulation. It will overrides all physics simulation.

Parameters

- **jnt_name** (*str*) – joint name.
- **jpos** (*float*) – target joint position.
- **jvel** (*float*) – optional, target joint velocity.

rot_ee_xyz (*angle*, *axis='x'*, *N=50*, **args*, ***kwargs*)
Rotate the end-effector about one of the end-effector axes, without changing the position

Parameters

- **angle** (*float*) – Angle by which to rotate in radians.
- **axis** (*str*) – Which end-effector frame axis to rotate about. Must be in ['x', 'y', 'z'].
- **N** (*int*) – Number of waypoints along the rotation trajectory (larger N means motion will be more smooth but potentially slower).

Returns *bool* – A boolean variable representing if the action is successful at the moment when the function exits.

set_ee_pose(*pos=None*, *ori=None*, *wait=True*, *ignore_physics=False*, **args*, ***kwargs*)
Move the end effector to the specifed pose.

Parameters

- **pos** (*list or np.ndarray*) – Desired x, y, z positions in the robot's base frame to move to (shape: $[3,]$).
- **ori** (*list or np.ndarray, optional*) – It can be euler angles ([roll, pitch, yaw], shape: $[4,]$), or quaternion ([qx, qy, qz, qw], shape: $[4,]$), or rotation matrix (shape: $[3, 3]$). If it's None, the solver will use the current end effector orientation as the target orientation.
- **wait** (*bool*) – whether to block the code and wait for the action to complete.
- **ignore_physics** (*bool*) – hard reset the joints to the target joint positions. It's best only to do this at the start, while not running the simulation. It will overrides all physics simulation.

Returns *bool* – A boolean variable representing if the action is successful at the moment when the function exits.

set_jpos(*position*, *joint_name=None*, *wait=True*, *ignore_physics=False*, **args*, ***kwargs*)
Move the arm to the specified joint position(s).

Parameters

- **position** (*float or list*) – desired joint position(s).
- **joint_name** (*str*) – If not provided, position should be a list and all the actuated joints will be moved to the specified positions. If provided, only the specified joint will be moved to the desired joint position.
- **wait** (*bool*) – whether to block the code and wait for the action to complete.
- **ignore_physics** (*bool*) – hard reset the joints to the target joint positions. It's best only to do this at the start, while not running the simulation. It will overrides all physics simulation.

Returns *bool* – A boolean variable representing if the action is successful at the moment when the function exits.

set_jtorq(*torque*, *joint_name=None*, *wait=False*, **args*, ***kwargs*)
Apply torque(s) to the joint(s), call enable_torque_control() or enable_torque_control(joint_name) before doing torque control.

Note: call to this function is only effective in this simulation step. you need to supply torque value for each simulation step to do the torque control. It's easier to use torque control in step_simulation mode instead of realtime_simulation mode. If you are using realtime_simulation mode, the time interval between two set_jtorq() calls must be small enough (like 0.0002s)

Parameters

- **torque** (*float or list*) – torque value(s) for the joint(s).
- **joint_name** (*str*) – specify the joint on which the torque is applied. If it's not provided(None), it will apply the torques on the six joints on the arm. Otherwise, only the specified joint will be applied with the given torque.
- **wait** (*bool*) – Not used in this method, just to keep the method signature consistent.

Returns *bool* – Always return True as the torque will be applied as specified in Pybullet.

set_jvel(*velocity*, *joint_name=None*, *wait=False*, **args*, ***kwargs*)
Move the arm with the specified joint velocity(ies).

Parameters

- **velocity** (*float or list*) – desired joint velocity(ies).
- **joint_name** (*str*) – If not provided, velocity should be a list and all the actuated joints will be moved in the specified velocities. If provided, only the specified joint will be moved in the desired joint velocity.
- **wait** (*bool*) – whether to block the code and wait for the action to complete.

Returns *bool* – A boolean variable representing if the action is successful at the moment when the function exits.

2.1.3 airobot.arm.single_arm_ros

2.1.4 airobot.arm.single_arm_real

2.1.5 airobot.arm.dual_arm_pybullet

class airobot.arm.dual_arm_pybullet.**DualArmPybullet**(*cfgs*, *pb_client*, *seed=None*, *self_collision=False*, *eetool_cfg=None*)

Bases: *airobot.arm.arm.ARM*

Class for the pybullet simulation environment of a dual arm robot.

Parameters

- **cfgs** (*YACS CfgNode*) – configurations for the arm.
- **pb_client** (BulletClient) – pybullet client.
- **seed** (*int*) – random seed.
- **self_collision** (*bool*) – enable self_collision or not whiling loading URDF.
- **eetool_cfg** (*dict*) – arguments to pass in the constructor of the end effector tool class.

Variables

- **cfgs** (*YACS CfgNode*) – configurations for the robot.
- **robot_id** (*int*) – pybullet body unique id of the robot.
- **arms** (*dict*) – internal dictionary keyed by the names of each single arm, with values as interfaces to the arms.
- **arm_jnt_names** (*list*) – names of the arm joints.
- **right_arm_jnt_names** (*list*) – names of the arm joints on the right arm.
- **left_arm_jnt_names** (*list*) – names of the arm joints on the left arm.
- **arm_jnt_ids** (*list*) – pybullet joint ids of the arm joints.
- **r_ee_link_jnt** (*str*) – name of the end effector link on the right arm.
- **l_ee_link_jnt** (*str*) – name of the end effector link on the left arm.

- **dual_arm_dof** (*int*) – total number of arm joints in the robot.
- **single_arm_dof** (*int*) – number of joints in a single arm of the robot.
- **jnt_to_id** (*dict*) – dictionary with [joint name : pybullet joint] id [key : value] pairs.
- **non_fixed_jnt_names** (*list*) – names of non-fixed joints in the arms, used for returning the correct inverse kinematics solution.

compute_ik(*pos*, *ori=None*, *arm=None*, *ns=False*, **args*, ***kwargs*)
Compute the inverse kinematics solution given the position and orientation of the end effector.

Parameters

- **pos** (*list or np.ndarray*) – position (shape: $[3,]$)
- **ori** (*list or np.ndarray*) – orientation. It can be euler angles ([roll, pitch, yaw], shape: $[3,]$), or quaternion ([qx, qy, qz, qw], shape: $[4,]$), or rotation matrix (shape: $[3, 3]$).
- **arm** (*str*) – Which arm EE pose corresponds to, must match arm names in cfg file
- **ns** (*bool*) – whether to use the nullspace options in pybullet, True if nullspace should be used. Defaults to False.

Returns *list* – solution to inverse kinematics, joint angles which achieve the specified EE pose (shape: $[DOF]$).

disable_torque_control(*joint_name=None*)
Disable the torque control mode in Pybullet.

Parameters **joint_name** (*str*) – If it's none, then all the six joints on the UR robot are disabled with torque control. Otherwise, only the specified joint is disabled with torque control. The joint(s) will enter velocity control mode.

enable_torque_control(*joint_name=None*)
Enable the torque control mode in Pybullet.

Parameters **joint_name** (*str*) – If it's none, then all the six joints on the UR robot are enabled in torque control mode. Otherwise, only the specified joint is enabled in torque control mode.

get_ee_pose(*arm=None*)
Return the end effector pose.

Parameters **arm** (*str*) – Returned pose will be for specified arm, must match arm names in cfg file.

Returns

4-element tuple containing

- np.ndarray: x, y, z position of the EE (shape: $[3,]$).
- np.ndarray: quaternion representation of the EE orientation (shape: $[4,]$).
- np.ndarray: rotation matrix representation of the EE orientation (shape: $[3, 3]$).
- np.ndarray: euler angle representation of the EE orientation (roll, pitch, yaw with static reference frame) (shape: $[3,]$).

get_ee_vel(*arm=None*)
Return the end effector's velocity.

Parameters **arm** (*str*) – Which arm to get velocity for, must match arm names in cfg file.

Returns

2-element tuple containing

- np.ndarray: translational velocity (shape: $[3,]$).
- np.ndarray: rotational velocity (shape: $[3,]$).

get_jpos(*joint_name=None*)
Return the joint position(s) of the arm.

Parameters **joint_name** (*str, optional*) – If it's None, it will return joint positions of all the actuated joints. Otherwise, it will return the joint position of the specified joint.

Returns

One of the following

- float: joint position given joint_name.
- list: joint positions if joint_name is None (shape: $[DOF]$).

get_jtorq(*joint_name=None*)
If the robot is operated in VELOCITY_CONTROL or POSITION_CONTROL mode, return the joint torque(s) applied during the last simulation step. In TORQUE_CONTROL, the applied joint motor torque is exactly what you provide, so there is no need to report it separately. So don't use this method to get the joint torque values when the robot is in TORQUE_CONTROL mode.

Parameters **joint_name** (*str, optional*) – If it's None, it will return joint torques of all the actuated joints. Otherwise, it will return the joint torque of the specified joint.

Returns

One of the following

- float: joint torque given joint_name.
- list: joint torques if joint_name is None (shape: $[DOF]$).

get_jvel(*joint_name=None*)
Return the joint velocity(ies) of the arm.

Parameters **joint_name** (*str, optional*) – If it's None, it will return joint velocities of all the actuated joints. Otherwise, it will return the joint velocity of the specified joint.

Returns

One of the following

- float: joint velocity given joint_name.
- list: joint velocities if joint_name is None (shape: $[DOF]$).

go_home(*arm=None*, *ignore_physics=False*)
Move the robot to a pre-defined home pose.

move_ee_xyz(*delta_xyz*, *eef_step=0.005*, *arm=None*, **args*, ***kwargs*)
Move the end-effector in a straight line without changing the orientation.

Parameters

- **delta_xyz** (*list or np.ndarray*) – movement in x, y, z directions (shape: $[3,]$).
- **eef_step** (*float*) – interpolation interval along delta_xyz. Interpolate a point every eef_step distance between the two end points.

- **arm** (*str*) – Which arm to move when setting cartesian command, must match arm names in cfg file.

Returns *bool* – A boolean variable representing if the action is successful at the moment when the function exits.

reset()
Reset the simulation environment.

reset_joint_state(*jnt_name*, *jpos*, *jvel=0*)
Reset the state of the joint. It's best only to do this at the start, while not running the simulation. It will overrides all physics simulation.

Parameters

- **jnt_name** (*str*) – joint name.
- **jpos** (*float*) – target joint position.
- **jvel** (*float*) – optional, target joint velocity.

set_ee_pose(*pos=None*, *ori=None*, *wait=True*, *arm=None*, **args*, ***kwargs*)
Move the end effector to the specifed pose.

Parameters

- **pos** (*list or np.ndarray*) – Desired x, y, z positions in the robot's base frame to move to (shape: $[3,]$).
- **ori** (*list or np.ndarray, optional*) – It can be euler angles ([roll, pitch, yaw], shape: $[4,]$), or quaternion ([qx, qy, qz, qw], shape: $[4,]$), or rotation matrix (shape: $[3, 3]$). If it's None, the solver will use the current end effector orientation as the target orientation.
- **wait** (*bool*) – Set to True if command should be blocking, otherwise the command can be overwritten before completion.
- **arm** (*str*) – Which arm to move when setting cartesian command, must match arm names in cfg file.

Returns *bool* – A boolean variable representing if the action is successful at the moment when the function exits.

set_jpos(*position*, *arm=None*, *joint_name=None*, *wait=True*, *ignore_physics=False*, **args*, ***kwargs*)
Move the arm to the specified joint position(s).

Parameters

- **position** (*float or list*) – desired joint position(s).
- **arm** (*str*) – If not provided, position should be a list and all actuated joints will be moved. If provided, only half the joints will move, corresponding to which arm was specified. Value should match arm names in cfg file.
- **joint_name** (*str*) – If not provided, position should be a list and all the actuated joints will be moved to the specified positions. If provided, only the specified joint will be moved to the desired joint position. If joint_name is provided, then arm argument should also be provided, and specified joint name must correspond to joint names for the specified arm.
- **wait** (*bool*) – whether to block the code and wait for the action to complete.

Returns *bool* – A boolean variable representing if the action is successful at the moment when the function exits.

set_jtorq(*torque*, *arm=None*, *joint_name=None*, *wait=False*, **args*, ***kwargs*)
Apply torque(s) to the joint(s), call enable_torque_control() or enable_torque_control(joint_name) before doing torque control.

Note: call to this function is only effective in this simulation step. you need to supply torque value for each simulation step to do the torque control. It's easier to use torque control in step_simulation mode instead of realtime_simulation mode. If you are using realtime_simulation mode, the time interval between two set_jtorq() calls must be small enough (like 0.0002s)

Parameters

- **torque** (*float or list*) – torque value(s) for the joint(s).
- **arm** (*str*) – If not provided, torque should be a list and all actuated joints will be moved. If provided, only half the joints will move, corresponding to which arm was specified. Value should match arm names in cfg file.
- **joint_name** (*str*) – specify the joint on which the torque is applied. If it's not provided(None), it will apply the torques on the six joints on the arm. Otherwise, only the specified joint will be applied with the given torque. If joint_name is provided, then arm argument should also be provided, and specified joint name must correspond to joint names for the specified arm.
- **wait** (*bool*) – Not used in this method, just to keep the method signature consistent.

Returns *bool* – Always return True as the torque will be applied as specified in Pybullet.

set_jvel(*velocity*, *arm=None*, *joint_name=None*, *wait=False*, **args*, ***kwargs*)
Move the arm with the specified joint velocity(ies).

Parameters

- **velocity** (*float or list*) – desired joint velocity(ies)
- **arm** (*str*) – If not provided, velocity should be a list and all actuated joints will be moved. If provided, only half the joints will move, corresponding to which arm was specified. Value should match arm names in cfg file.
- **joint_name** (*str*) – If not provided, velocity should be a list and all the actuated joints will be moved in the specified velocities. If provided, only the specified joint will be moved in the desired joint velocity. If joint_name is provided, then arm argument should also be provided, and specified joint name must correspond to joint names for the specified arm.
- **wait** (*bool*) – whether to block the code and wait for the action to complete

Returns *bool* – A boolean variable representing if the action is successful at the moment when the function exits

setup_single_arms(*right_arm*, *left_arm*)
Function to setup the single arm instances, and maintain an internal dictionary interface to them.

Parameters

- **right_arm** (SingleArmPybullet) – Right arm instance.
- **left_arm** (SingleArmPybullet) – Left arm instance.

2.1.6 airobot.arm.ur5e_pybullet

Pybullet simulation environment of a UR5e robot with a robotiq 2f140 gripper

class airobot.arm.ur5e_pybullet.**UR5ePybullet**(*cfgs*, *pb_client*, *seed=None*, *self_collision=False*, *eetool_cfg=None*)

Bases: *airobot.arm.single_arm_pybullet.SingleArmPybullet*

Class for the pybullet simulation environment of a UR5e robot with a robotiq 2f140 gripper.

Parameters

- **cfgs** (*YACS CfgNode*) – configurations for the arm
- **pb_client** (BulletClient) – pybullet client
- **seed** (*int*) – random seed
- **self_collision** (*bool*) – enable self_collision or not whiling loading URDF
- **eetool_cfg** (*dict*) – arguments to pass in the constructor of the end effector tool class

Variables

- **floor_id** (*int*) – pybullet body unique id of the floor
- **robot_id** (*int*) – pybullet body unique id of the robot
- **robot_base_pos** (*list*) – world frame position of the robot base link, shape: $[3,]$ ([x, y, z])
- **robot_base_ori** (*list*) – world frame orientation of the robot base link shape: $[4,]$ ([x, y, z, w])

reset(*force_reset=False*)

Reset the simulation environment.

set_visual_shape()

Set the color of the UR arm.

2.1.7 airobot.arm.ur5e_real

2.1.8 airobot.arm.yumi_pybullet

Pybullet simulation environment of an ABB Yumi robot.

class airobot.arm.yumi_pybullet.**YumiPybullet**(*cfgs*, *pb_client*, *seed=None*, *self_collision=False*, *eetool_cfg=None*)

Bases: *airobot.arm.dual_arm_pybullet.DualArmPybullet*

Class for pybullet simulation of ABB Yumi robot with separate functionality for both arms.

Parameters

- **cfgs** (*YACS CfgNode*) – configurations for the arm.
- **pb_client** (BulletClient) – pybullet client.
- **seed** (*int*) – random seed.
- **self_collision** (*bool*) – enable self_collision or not whiling loading URDF.
- **eetool_cfg** (*dict*) – arguments to pass in the constructor of the end effector tool class.

Variables

- **right_arm** (SingleArmPybullet) – Right arm instance
- **left_arm** (SingleArmPybullet) – Left arm instance
- **robot_id** (*int*) – PyBullet body unique id of the robot

reset (*force_reset=False*)
Reset the simulation environment.

2.1.9 airobot.arm.yumi_palms_pybullet

Pybullet simulation environment of an ABB Yumi robot with Gelslim palms and a compliant wrist

class airobot.arm.yumi_palms_pybullet.**CompliantYumiArm**(*cfgs*, *pb_client*, *seed=None*, *self_collision=False*, *eetool_cfg=None*)
Bases: *airobot.arm.single_arm_pybullet.SingleArmPybullet*

Class for the pybullet simulation of a single arm of the ABB Yumi robot, with additional joints specified to behave like springs.

Parameters

- **cfgs** (*YACS CfgNode*) – configurations for the arm.
- **pb_client** (BulletClient) – pybullet client.
- **seed** (*int*) – random seed.
- **self_collision** (*bool*) – enable self_collision or not whiling loading URDF.
- **eetool_cfg** (*dict*) – arguments to pass in the constructor of the end effector tool class.

Variables

- **comp_jnt_names** (*list*) – Names of the spring-like compliant joints.
- **comp_dof** (*list*) – Number of spring-like compliant joints.
- **comp_jnt_gains** (*list*) – Stiffness of spring-like compliant joints.
- **comp_jnt_ids** (*list*) – PyBullet joint ids of compliant joints.
- **max_force_comp** (*list*) – Maximum force that can be applied at the compliant joints.

set_compliant_jpos()
Regulate compliant/spring like joints about nominal position.

set_jpos(*position*, *joint_name=None*, *wait=True*, *ignore_physics=False*, **args*, ***kwargs*)
Move the arm to the specified joint position(s). Applies regulation position command to the compliant joints after sending driven joint commands.

Parameters

- **position** (*float or list*) – desired joint position(s).
- **joint_name** (*str*) – If not provided, position should be a list and all the actuated joints will be moved to the specified positions. If provided, only the specified joint will be moved to the desired joint position.
- **wait** (*bool*) – whether to block the code and wait for the action to complete.

Returns *bool* – A boolean variable representing if the action is successful at the moment when the function exits.

set_jtorq(*torque*, *joint_name=None*, *wait=False*, **args*, ***kwargs*)
Apply torque(s) to the joint(s), call enable_torque_control() or enable_torque_control(joint_name) before doing torque control. Applies regulation position command to the compliant joints after sending driven commands.

Note: call to this function is only effective in this simulation step. you need to supply torque value for each simulation step to do the torque control. It's easier to use torque control in step_simulation mode instead of realtime_simulation mode. If you are using realtime_simulation mode, the time interval between two set_jtorq() calls must be small enough (like 0.0002s).

Parameters

- **torque** (*float or list*) – torque value(s) for the joint(s).
- **joint_name** (*str*) – specify the joint on which the torque is applied. If it's not provided(None), it will apply the torques on the six joints on the arm. Otherwise, only the specified joint will be applied with the given torque.
- **wait** (*bool*) – Not used in this method, just to keep the method signature consistent.

Returns *bool* – Always return True as the torque will be applied as specified in Pybullet.

set_jvel(*velocity*, *joint_name=None*, *wait=False*, **args*, ***kwargs*)
Move the arm with the specified joint velocity(ies). Applies regulation position command to the compliant joints after sending driven commands.

Parameters

- **velocity** (*float or list*) – desired joint velocity(ies).
- **joint_name** (*str*) – If not provided, velocity should be a list and all the actuated joints will be moved in the specified velocities. If provided, only the specified joint will be moved in the desired joint velocity.
- **wait** (*bool*) – whether to block the code and wait for the action to complete.

Returns *bool* – A boolean variable representing if the action is successful at the moment when the function exits.

class airobot.arm.yumi_palms_pybullet.**YumiPalmsPybullet**(*cfgs*, *pb_client*, *seed=None*, *self_collision=False*, *eetool_cfg=None*)
Bases: *airobot.arm.dual_arm_pybullet.DualArmPybullet*

Class for pybullet simulation of ABB Yumi robot with separate functionality for both arms, with two Gelslim Palms attached as end effectors instead of parallel jaw grippers.

Parameters

- **cfgs** (*YACS CfgNode*) – configurations for the arm.
- **pb_client** (BulletClient) – pybullet client.
- **seed** (*int*) – random seed.
- **self_collision** (*bool*) – enable self_collision or not whiling loading URDF.
- **eetool_cfg** (*dict*) – arguments to pass in the constructor of the end effector tool class.

Variables

- **arms** (*dict*) – internal dictioanry keyed by the single arm names, values are interfaces to the single arm instances.
- **robot_id** (*int*) – pybullet unique body id of the robot.
- **left_arm** (CompliantYumiArm) – left arm interface.
- **right_arm** (CompliantYumiArm) – right arm interface.

reset (*force_reset=False*)
Reset the simulation environment.

setup_single_arms (*right_arm*, *left_arm*)
Function for setting up individual arms.

Parameters

- **right_arm** (CompliantYumiArm) – Instance of a single yumi arm with compliant joints.
- **left_arm** (CompliantYumiArm) – Instance of a single yumi arm with compliant joints.

2.2 airobot.base

2.3 airobot.ee_tool

2.3.1 airobot.ee_tool.ee

class airobot.ee_tool.ee.**EndEffectorTool** (*cfgs*)
Bases: object

Base class for end effector.

Parameters **cfgs** (*YACS CfgNode*) – configurations for the end effector.

Variables **cfgs** (*YACS CfgNode*) – configurations for the end effector.

close (***kwargs*)

open (***kwargs*)

2.3.2 airobot.ee_tool.robotiq2f140_pybullet

class airobot.ee_tool.robotiq2f140_pybullet.**Robotiq2F140Pybullet** (*cfgs*, *pb_client*)
Bases: airobot.ee_tool.simple_gripper_mimic_pybullet.SimpleGripperMimicPybullet

Class for interfacing with a Robotiq 2F140 gripper when it is attached to UR5e arm in pybullet.

Parameters

- **cfgs** (*YACS CfgNode*) – configurations for the gripper.
- **pb_client** (BulletClient) – pybullet client.

Variables

- **cfgs** (*YACS CfgNode*) – configurations for the gripper.

- **gripper_close_angle** (*float*) – position value corresponding to the fully closed position of the gripper.
- **gripper_open_angle** (*float*) – position value corresponding to the fully open position of the gripper.
- **jnt_names** (*list*) – names of the gripper joints.
- **gripper_jnt_ids** (*list*) – pybullet joint ids of the gripper joints.
- **robot_id** (*int*) – robot id in Pybullet.
- **jnt_to_id** (*dict*) – mapping from the joint name to joint id.

feed_robot_info(*robot_id*, *jnt_to_id*)
Setup the gripper, pass the robot info from the arm to the gripper.

Parameters

- **robot_id** (*int*) – robot id in Pybullet.
- **jnt_to_id** (*dict*) – mapping from the joint name to joint id.

2.3.3 airobot.ee_tool.robotiq2f140_real

2.3.4 airobot.ee_tool.yumi_parallel_jaw_pybullet

class airobot.ee_tool.yumi_parallel_jaw_pybullet.**YumiParallelJawPybullet**(*cfgs*, *pb_client*)
Bases: airobot.ee_tool.simple_gripper_mimic_pybullet.SimpleGripperMimicPybullet

Class for interfacing with the standard Yumi parallel jaw gripper.

Parameters

- **cfgs** (*YACS CfgNode*) – configurations for the gripper
- **pb_client** (BulletClient) – pybullet client.

Variables

- **cfgs** (*YACS CfgNode*) – configurations for the gripper.
- **gripper_close_angle** (*float*) – position value corresponding to the fully closed position of the gripper.
- **gripper_open_angle** (*float*) – position value corresponding to the fully open position of the gripper.
- **jnt_names** (*list*) – names of the gripper joints.
- **gripper_jnt_ids** (*list*) – pybullet joint ids of the gripper joints.
- **robot_id** (*int*) – robot id in Pybullet.
- **jnt_to_id** (*dict*) – mapping from the joint name to joint id.

2.4 airobot.sensor

2.4.1 airobot.sensor.camera

airobot.sensor.camera.camera

class airobot.sensor.camera.camera.**Camera**(*cfgs*)
Bases: object

Base class for cameras.

Parameters **cfgs** (*YACS CfgNode*) – configurations for the camera.

Variables **cfgs** (*YACS CfgNode*) – configurations for the camera.

get_images(*get_rgb=True*, *get_depth=True*, ***kwargs*)
Return rgb/depth images.

Parameters

- **get_rgb** (*bool*) – return rgb image if True, None otherwise.
- **get_depth** (*bool*) – return depth image if True, None otherwise.

Returns

2-element tuple containing

- np.ndarray: rgb image.
- np.ndarray: depth image.

airobot.sensor.camera.rgbdcam

class airobot.sensor.camera.rgbdcam.**RGBDCamera**(*cfgs*)
Bases: *airobot.sensor.camera.camera.Camera*

A RGBD camera.

Parameters **cfgs** (*YACS CfgNode*) – configurations for the camera.

Variables

- **cfgs** (*YACS CfgNode*) – configurations for the end effector.
- **img_height** (*int*) – height of the image.
- **img_width** (*int*) – width of the image.
- **cam_ext_mat** (*np.ndarray*) – extrinsic matrix (shape: $[4, 4]$) for the camera (source frame: base frame. target frame: camera frame).
- **cam_int_mat** (*np.ndarray*) – intrinsic matrix (shape: $[3, 3]$) for the camera.
- **cam_int_mat_inv** (*np.ndarray*) – inverse of the intrinsic matrix.
- **depth_scale** (*float*) – ratio of the depth image value to true depth value.
- **depth_min** (*float*) – minimum depth value considered in 3D reconstruction.
- **depth_max** (*float*) – maximum depth value considered in 3D reconstruction.

get_cam_ext()
Return the camera's extrinsic matrix.

Returns *np.ndarray* – extrinsic matrix (shape: $[4, 4]$) for the camera (source frame: base frame. target frame: camera frame).

get_cam_int()
Return the camera's intrinsic matrix.

Returns *np.ndarray* – intrinsic matrix (shape: $[3, 3]$) for the camera.

get_pcd(*in_world=True*, *filter_depth=True*, *depth_min=None*, *depth_max=None*, *cam_ext_mat=None*, *rgb_image=None*, *depth_image=None*)

Get the point cloud from the entire depth image in the camera frame or in the world frame.

Parameters

- **in_world** (*bool*) – return point cloud in the world frame, otherwise, return point cloud in the camera frame.
- **filter_depth** (*bool*) – only return the point cloud with depth values lying in [depth_min, depth_max].
- **depth_min** (*float*) – minimum depth value. If None, it will use the default minimum depth value defined in the config file.
- **depth_max** (*float*) – maximum depth value. If None, it will use the default maximum depth value defined in the config file.
- **cam_ext_mat** (*np.ndarray*) – camera extrinsic matrix (shape: $[4, 4]$). If provided, it will be used to compute the points in the world frame.
- **rgb_image** (*np.ndarray*) – externally captured RGB image, if we want to convert a depth image captured outside this function to a point cloud. (shape $[H, W, 3]$)
- **depth_image** (*np.ndarray*) – externally captured depth image, if we want to convert a depth image captured outside this function to a point cloud. (shape $[H, W]$)

Returns

2-element tuple containing

- np.ndarray: point coordinates (shape: $[N, 3]$).
- np.ndarray: rgb values (shape: $[N, 3]$).

get_pix_3dpt(*rs*, *cs*, *in_world=True*, *filter_depth=False*, *k=1*, *ktype='median'*, *depth_min=None*, *depth_max=None*, *cam_ext_mat=None*)

Calculate the 3D position of pixels in the RGB image.

Parameters

- **rs** (*int or list or np.ndarray*) – rows of interest. It can be a list or 1D numpy array which contains the row indices. The default value is None, which means all rows.
- **cs** (*int or list or np.ndarray*) – columns of interest. It can be a list or 1D numpy array which contains the column indices. The default value is None, which means all columns.
- **in_world** (*bool*) – if True, return the 3D position in the world frame, Otherwise, return the 3D position in the camera frame.
- **filter_depth** (*bool*) – if True, only pixels with depth values between [depth_min, depth_max] will remain.
- **k** (*int*) – kernel size. A kernel (slicing window) will be used to get the neighboring depth values of the pixels specified by rs and cs. And depending on the ktype, a corresponding method will be applied to use some statistical value (such as minimum, maximum, median, mean) of all the depth values in the slicing window as a more robust estimate of the depth value of the specified pixels.
- **ktype** (*str*) – what kind of statistical value of all the depth values in the sliced kernel to use as a proxy of the depth value at specified pixels. It can be *median*, *min*, *max*, *mean*.

- **depth_min** (*float*) – minimum depth value. If None, it will use the default minimum depth value defined in the config file.
- **depth_max** (*float*) – maximum depth value. If None, it will use the default maximum depth value defined in the config file.
- **cam_ext_mat** (*np.ndarray*) – camera extrinsic matrix (shape: $[4, 4]$). If provided, it will be used to compute the points in the world frame.

Returns *np.ndarray* – 3D point coordinates of the pixels in camera frame (shape: $[N, 3]$).

set_cam_ext(*pos=None*, *ori=None*, *cam_ext=None*)
Set the camera extrinsic matrix.

Parameters

- **pos** (*np.ndarray*) – position of the camera (shape: $[3,]$).
- **ori** (*np.ndarray*) – orientation. It can be rotation matrix (shape: $[3, 3]$) quaternion ([x, y, z, w], shape: $[4]$), or euler angles ([roll, pitch, yaw], shape: $[3]$).
- **cam_ext** (*np.ndarray*) – extrinsic matrix (shape: $[4, 4]$). If this is provided, pos and ori will be ignored.

airobot.sensor.camera.rgbdcam_pybullet

class airobot.sensor.camera.rgbdcam_pybullet.**RGBDCameraPybullet**(*cfgs*, *pb_client*)

Bases: *airobot.sensor.camera.rgbdcam.RGBDCamera*

RGBD Camera in Pybullet.

Parameters **cfgs** (*YACS CfgNode*) – configurations for the camera.

Variables

- **view_matrix** (*np.ndarray*) – view matrix of opengl camera (shape: $[4, 4]$).
- **proj_matrix** (*np.ndarray*) – projection matrix of opengl camera (shape: $[4, 4]$).

get_images(*get_rgb=True*, *get_depth=True*, *get_seg=False*, ***kwargs*)
Return rgb, depth, and segmentation images.

Parameters

- **get_rgb** (*bool*) – return rgb image if True, None otherwise.
- **get_depth** (*bool*) – return depth image if True, None otherwise.
- **get_seg** (*bool*) – return the segmentation mask if True, None otherwise.

Returns

2-element tuple (if *get_seg* is False) containing

- np.ndarray: rgb image (shape: [H, W, 3]).
- np.ndarray: depth image (shape: [H, W]).

3-element tuple (if *get_seg* is True) containing

- np.ndarray: rgb image (shape: [H, W, 3]).
- np.ndarray: depth image (shape: [H, W]).

- np.ndarray: segmentation mask image (shape: [H, W]), with pixel values corresponding to object id and link id. From the PyBullet documentation, the pixel value "combines the object unique id and link index as follows: value = objectUniqueId + (linkIndex+1)<<24 ... for a free floating body without joints/links, the segmentation mask is equal to its body unique id, since its link index is -1.".

set_cam_ext (*pos=None*, *ori=None*, *cam_ext=None*)
Set the camera extrinsic matrix.

Parameters

- **pos** (*np.ndarray*) – position of the camera (shape: $[3,]$).
- **ori** (*np.ndarray*) – orientation. It can be rotation matrix (shape: $[3, 3]$) quaternion ([x, y, z, w], shape: $[4]$), or euler angles ([roll, pitch, yaw], shape: $[3]$).
- **cam_ext** (*np.ndarray*) – extrinsic matrix (shape: $[4, 4]$). If this is provided, pos and ori will be ignored.

setup_camera (*focus_pt=None*, *dist=3*, *yaw=0*, *pitch=0*, *roll=0*, *height=None*, *width=None*)
Setup the camera view matrix and projection matrix. Must be called first before images are renderred.

Parameters

- **focus_pt** (*list*) – position of the target (focus) point, in Cartesian world coordinates.
- **dist** (*float*) – distance from eye (camera) to the focus point.
- **yaw** (*float*) – yaw angle in degrees, left/right around up-axis (z-axis).
- **pitch** (*float*) – pitch in degrees, up/down.
- **roll** (*float*) – roll in degrees around forward vector.
- **height** (*float*) – height of image. If None, it will use the default height from the config file.
- **width** (*float*) – width of image. If None, it will use the default width from the config file.

airobot.sensor.camera.rgbdcam_real

2.5 airobot.cfgs

2.5.1 airobot.cfgs.ur5e_cfg

airobot.cfgs.ur5e_cfg.**get_cfg**()

```
from airobot.cfgs.assets.default_configs import get_cfg_defaults
from airobot.cfgs.assets.pybullet_camera import get_sim_cam_cfg
from airobot.cfgs.assets.realsense_camera import get_realsense_cam_cfg
from airobot.cfgs.assets.ur5e_arm import get_ur5e_arm_cfg

_C = get_cfg_defaults()
# whether the robot has an arm or not
_C.HAS_ARM = True
# whether the robot has a camera or not
_C.HAS_CAMERA = True
# whether the robot has a end effector tool or not
```

(continues on next page)

(continued from previous page)

```
_C.HAS_EETOOL = False

_C.ROBOT_DESCRIPTION = '/robot_description'
_C.PYBULLET_URDF = 'ur5e_pybullet.urdf'

_C.ARM = get_ur5e_arm_cfg()

_C.CAM.SIM = get_sim_cam_cfg()
_C.CAM.REAL = get_realsense_cam_cfg()
_C.CAM.CLASS = 'RGBDCamera'


def get_cfg():
    return _C.clone()
```

2.5.2 airobot.cfgs.ur5e_2f140_cfg

airobot.cfgs.ur5e_2f140_cfg.**get_cfg**()

```
from airobot.cfgs.assets.default_configs import get_cfg_defaults
from airobot.cfgs.assets.pybullet_camera import get_sim_cam_cfg
from airobot.cfgs.assets.realsense_camera import get_realsense_cam_cfg
from airobot.cfgs.assets.robotiq2f140 import get_robotiq2f140_cfg
from airobot.cfgs.assets.ur5e_arm import get_ur5e_arm_cfg

_C = get_cfg_defaults()
# whether the robot has an arm or not
_C.HAS_ARM = True
# whether the robot has a camera or not
_C.HAS_CAMERA = True
# whether the robot has a end effector tool or not
_C.HAS_EETOOL = True

_C.ROBOT_DESCRIPTION = '/robot_description'
_C.PYBULLET_URDF = 'ur5e_2f140_pybullet.urdf'

_C.ARM = get_ur5e_arm_cfg()

_C.CAM.SIM = get_sim_cam_cfg()
_C.CAM.REAL = get_realsense_cam_cfg()
_C.CAM.CLASS = 'RGBDCamera'

_C.EETOOL = get_robotiq2f140_cfg()
_C.EETOOL.CLASS = 'Robotiq2F140'


def get_cfg():
    return _C.clone()
```

2.5.3 airobot.cfgs.ur5e_stick_cfg

airobot.cfgs.ur5e_stick_cfg.**get_cfg**()

```
from airobot.cfgs.assets.default_configs import get_cfg_defaults
from airobot.cfgs.assets.pybullet_camera import get_sim_cam_cfg
from airobot.cfgs.assets.realsense_camera import get_realsense_cam_cfg
from airobot.cfgs.assets.ur5e_arm import get_ur5e_arm_cfg

_C = get_cfg_defaults()
# whether the robot has an arm or not
_C.HAS_ARM = True
# whether the robot has a camera or not
_C.HAS_CAMERA = True
# whether the robot has a end effector tool or not
_C.HAS_EETOOL = False

_C.ROBOT_DESCRIPTION = '/robot_description'
_C.PYBULLET_URDF = 'ur5e_stick_pybullet.urdf'

_C.ARM = get_ur5e_arm_cfg()

# update end-effector frame of the arm
_C.ARM.ROBOT_EE_FRAME = 'ee_tip'
_C.ARM.ROBOT_EE_FRAME_JOINT = 'ee_tip_joint'

_C.CAM.SIM = get_sim_cam_cfg()
_C.CAM.REAL = get_realsense_cam_cfg()
_C.CAM.CLASS = 'RGBDCamera'


def get_cfg():
    return _C.clone()
```

2.5.4 airobot.cfgs.yumi_cfg

airobot.cfgs.yumi_cfg.**get_cfg**()

```
from airobot.cfgs.assets.pybullet_camera import get_sim_cam_cfg
from airobot.cfgs.assets.realsense_camera import get_realsense_cam_cfg
from airobot.cfgs.assets.yumi_dual_arm import get_yumi_dual_arm_cfg

# _C = get_cfg_defaults()
_C = get_yumi_dual_arm_cfg()

_C.ROBOT_DESCRIPTION = '/robot_description'

# _C = get_yumi_dual_arm_cfg()
_C.PYBULLET_URDF = 'yumi.urdf'

_C.CAM.SIM = get_sim_cam_cfg()
_C.CAM.REAL = get_realsense_cam_cfg()
_C.CAM.CLASS = 'RGBDCamera'


def get_cfg():
    return _C.clone()
```

2.5.5 airobot.cfgs.yumi_grippers_cfg

airobot.cfgs.yumi_grippers_cfg.**get_cfg**()

```
from airobot.cfgs.assets.pybullet_camera import get_sim_cam_cfg
from airobot.cfgs.assets.realsense_camera import get_realsense_cam_cfg
from airobot.cfgs.assets.yumi_dual_arm import get_yumi_dual_arm_cfg
from airobot.cfgs.assets.yumi_parallel_jaw import get_yumi_parallel_jaw_cfg

_C = get_yumi_dual_arm_cfg()

_C.ROBOT_DESCRIPTION = '/robot_description'

_C.PYBULLET_URDF = 'yumi_grippers.urdf'

_C.CAM.SIM = get_sim_cam_cfg()
_C.CAM.REAL = get_realsense_cam_cfg()
_C.CAM.CLASS = 'RGBDCamera'

_C.ARM.RIGHT.HAS_EETOOL = True

_C.ARM.RIGHT.EETOOL = get_yumi_parallel_jaw_cfg()
_C.ARM.RIGHT.EETOOL.JOINT_NAMES = ['gripper_r_joint', 'gripper_r_joint_m']
_C.ARM.RIGHT.EETOOL.MIMIC_COEFF = [1, 1]

_C.ARM.LEFT.HAS_EETOOL = True

_C.ARM.LEFT.EETOOL = get_yumi_parallel_jaw_cfg()
_C.ARM.LEFT.EETOOL.JOINT_NAMES = ['gripper_l_joint', 'gripper_l_joint_m']
_C.ARM.LEFT.EETOOL.MIMIC_COEFF = [1, 1]


def get_cfg():
    return _C.clone()
```

2.5.6 airobot.cfgs.yumi_palms_cfg

airobot.cfgs.yumi_palms_cfg.**get_cfg**()

```
from airobot.cfgs.assets.pybullet_camera import get_sim_cam_cfg
from airobot.cfgs.assets.realsense_camera import get_realsense_cam_cfg
from airobot.cfgs.assets.yumi_dual_arm import get_yumi_dual_arm_cfg

_C = get_yumi_dual_arm_cfg()

_C.ROBOT_DESCRIPTION = '/robot_description'
_C.PYBULLET_URDF = 'yumi_gelslim_palm.urdf'

# prefix of the class name of the ARM
# if it's for pybullet simulation, the name will
# be augemented to be '<Prefix>Pybullet'
# if it's for the real robot, the name will be
# augmented to be '<Prefix>Real'
_C.ARM.CLASS = 'YumiPalms'

# yumi with palms has compliant joints at the wrist and in the gel
```

(continues on next page)

(continued from previous page)

```
_C.ARM.RIGHT.ARM.COMPLIANT_JOINT_NAMES = ['yumi_palm_r', 'yumi_gel_r']
_C.ARM.RIGHT.ARM.COMPLIANT_GAINS = [1, 0.1]
_C.ARM.RIGHT.ARM.COMPLIANT_MAX_FORCE = 1

# yumi with palms has compliant joints at the wrist and in the gel
_C.ARM.LEFT.ARM.COMPLIANT_JOINT_NAMES = ['yumi_palm_l', 'yumi_gel_l']
_C.ARM.LEFT.ARM.COMPLIANT_GAINS = [1, 0.1]
_C.ARM.LEFT.ARM.COMPLIANT_MAX_FORCE = 1

_C.CAM.SIM = get_sim_cam_cfg()
_C.CAM.REAL = get_realsense_cam_cfg()
_C.CAM.CLASS = 'RGBDCamera'


def get_cfg():
    return _C.clone()
```

2.5.7 airobot.cfgs.assets

airobot.cfgs.assets.default_configs

airobot.cfgs.assets.default_configs.**get_cfg_defaults**()

```
from yacs.config import CfgNode as CN

_C = CN()

# whether the robot has an arm or not
_C.HAS_ARM = False
# whether the robot has a mobile base or not
_C.HAS_BASE = False
# whether the robot has a camera or not
_C.HAS_CAMERA = False
# whether the robot has a end effector tool or not
_C.HAS_EETOOL = False

_C.ARM = CN()
_C.ARM.CLASS = ''

_C.CAM = CN()
_C.CAM.CLASS = ''

_C.BASE = CN()
_C.BASE.CLASS = ''

_C.EETOOL = CN()
_C.EETOOL.CLASS = ''


def get_cfg_defaults():
    return _C.clone()
```

airobot.cfgs.assets.pybullet_camera

airobot.cfgs.assets.pybullet_camera.**get_sim_cam_cfg**()

```
from yacs.config import CfgNode as CN

_C = CN()
_C.ZNEAR = 0.01
_C.ZFAR = 10
_C.WIDTH = 640
_C.HEIGHT = 480
_C.FOV = 60


def get_sim_cam_cfg():
    return _C.clone()
```

airobot.cfgs.assets.realsense_camera

airobot.cfgs.assets.realsense_camera.**get_realsense_cam_cfg**()

```
from yacs.config import CfgNode as CN

_C = CN()
# topic name of the camera info
_C.ROSTOPIC_CAMERA_INFO = 'camera/color/camera_info'
# topic name of the RGB images
_C.ROSTOPIC_CAMERA_RGB = 'camera/color/image_rect_color'
# topic name of the depth images
_C.ROSTOPIC_CAMERA_DEPTH = 'camera/aligned_depth_to_color/image_raw'
# minimum depth values to be considered as valid (m)
_C.DEPTH_MIN = 0.2
# maximum depth values to be considered as valid (m)
_C.DEPTH_MAX = 2
# scale factor to map depth image values to real depth values (m)
_C.DEPTH_SCALE = 0.001


def get_realsense_cam_cfg():
    return _C.clone()
```

airobot.cfgs.assets.robotiq2f140

airobot.cfgs.assets.robotiq2f140.**get_robotiq2f140_cfg**()

```
from yacs.config import CfgNode as CN

_C = CN()
# joint angle when the gripper is fully open
_C.OPEN_ANGLE = 0.0
# joint angle when the gripper is fully closed
_C.CLOSE_ANGLE = 0.7

# default host and port for socket used to
# communicate with the gripper through the
```

(continues on next page)

(continued from previous page)

```
# UR controller
_C.SOCKET_HOST = '127.0.0.1'
_C.SOCKET_PORT = 63352
_C.SOCKET_NAME = 'gripper_socket'

_C.DEFAULT_SPEED = 255
_C.DEFAULT_FORCE = 50

_C.COMMAND_TOPIC = '/ur_driver/URScript'
_C.GAZEBO_COMMAND_TOPIC = '/gripper/gripper_cmd/goal'
_C.JOINT_STATE_TOPIC = '/joint_states'
# Prefix of IP address of UR5 on local network
_C.IP_PREFIX = '192.168'

# Replace with the IP found on the UR5 pendant
_C.FULL_IP = '127.0.0.1'

# time in seconds to wait for new gripper state before exiting
_C.UPDATE_TIMEOUT = 5.0

# default maximum values for gripper state varibles
# minimum values are all 0
_C.POSITION_RANGE = 255
# scaling factor to convert from URScript range to Robotiq range
_C.POSITION_SCALING = (255 / 0.7)

# for the gripper in pybullet
_C.JOINT_NAMES = [
    'finger_joint', 'left_inner_knuckle_joint',
    'left_inner_finger_joint', 'right_outer_knuckle_joint',
    'right_inner_knuckle_joint', 'right_inner_finger_joint',
]
_C.MIMIC_COEFF = [1, -1, 1, -1, -1, 1]
_C.MAX_TORQUE = 25.0


def get_robotiq2f140_cfg():
    return _C.clone()
```

airobot.cfgs.assets.yumi_parallel_jaw

airobot.cfgs.assets.yumi_parallel_jaw.**get_yumi_parallel_jaw_cfg**()

```
from yacs.config import CfgNode as CN

_C = CN()
# joint angle when the gripper is fully open
_C.OPEN_ANGLE = 0.025
# joint angle when the gripper is fully closed
_C.CLOSE_ANGLE = 0.0

_C.MAX_TORQUE = 100.0

_C.CLASS = 'YumiParallelJawPybullet'
```

(continues on next page)

(continued from previous page)

```
def get_yumi_parallel_jaw_cfg():
    return _C.clone()
```

airobot.cfgs.assets.ur5e_arm

airobot.cfgs.assets.ur5e_arm.**get_ur5e_arm_cfg**()

```
from yacs.config import CfgNode as CN

_C = CN()

# prefix of the class name of the ARM
# if it's for pybullet simulation, the name will
# be augemented to be '<Prefix>Pybullet'
# if it's for the real robot, the name will be
# augmented to be '<Prefix>Real'
_C.CLASS = 'UR5e'
_C.MOVEGROUP_NAME = 'manipulator'
_C.ROSTOPIC_JOINT_STATES = '/joint_states'

# https://www.universal-robots.com/how-tos-and-faqs/faq/ur-faq/max-joint-torques-
↪17260/
_C.MAX_TORQUES = [150, 150, 150, 28, 28, 28]
_C.JOINT_NAMES = [
    'shoulder_pan_joint', 'shoulder_lift_joint', 'elbow_joint',
    'wrist_1_joint', 'wrist_2_joint', 'wrist_3_joint'
]
# base frame for the arm
_C.ROBOT_BASE_FRAME = 'base'
# end-effector frame of the arm
_C.ROBOT_EE_FRAME = 'ee_tip'
_C.ROBOT_EE_FRAME_JOINT = 'ee_tip_joint'
_C.JOINT_SPEED_TOPIC = '/joint_speed'
_C.URSCRIPT_TOPIC = '/ur_driver/URScript'
# inverse kinematics position tolerance (m)
_C.IK_POSITION_TOLERANCE = 0.01
# inverse kinematics orientation tolerance (rad)
_C.IK_ORIENTATION_TOLERANCE = 0.05
_C.HOME_POSITION = [0, -1.66, -1.92, -1.12, 1.57, 0]
_C.MAX_JOINT_ERROR = 0.01
_C.MAX_JOINT_VEL_ERROR = 0.05
_C.MAX_EE_POS_ERROR = 0.01
# real part of the quaternion difference should be
# greater than 1-error
_C.MAX_EE_ORI_ERROR = 0.02
_C.TIMEOUT_LIMIT = 10

# reset position for the robot in pybullet
_C.PYBULLET_RESET_POS = [0, 0, 1]
# reset orientation (euler angles) for the robot in pybullet
_C.PYBULLET_RESET_ORI = [0, 0, 0]
_C.PYBULLET_IK_DAMPING = 0.0005
```

(continues on next page)

(continued from previous page)

```
def get_ur5e_arm_cfg():
    return _C.clone()
```

airobot.cfgs.assets.yumi_arm

airobot.cfgs.assets.yumi_arm.**get_yumi_arm_cfg**()

```
from yacs.config import CfgNode as CN

_C = CN()
_C.ROBOT_DESCRIPTION = '/robot_description'

_C.ROSTOPIC_JOINT_STATES = '/joint_states'

# base frame for the arm
_C.ROBOT_BASE_FRAME = 'yumi_body'
_C.CLASS = 'Yumi'

_C.JOINT_SPEED_TOPIC = '/joint_speed'

# inverse kinematics position tolerance (m)
_C.IK_POSITION_TOLERANCE = 0.01
# inverse kinematics orientation tolerance (rad)
_C.IK_ORIENTATION_TOLERANCE = 0.1
_C.MAX_JOINT_ERROR = 0.01
_C.MAX_JOINT_VEL_ERROR = 0.1
_C.MAX_EE_POS_ERROR = 0.01
# real part of the quaternion difference should be
# greater than 1-error
_C.MAX_EE_ORI_ERROR = 0.02
_C.TIMEOUT_LIMIT = 10

# reset position for the robot in pybullet
_C.PYBULLET_RESET_POS = [0, 0, -0.1]
# reset orientation (euler angles) for the robot in pybullet
_C.PYBULLET_RESET_ORI = [0, 0, 0]
# damped inverse kinematics value
_C.PYBULLET_IK_DAMPING = 0.0005

# NOTE: Order of joints on yumi is [1, 2, 7, 3, 4, 5, 6]
# these torques are listed in that order
# _C.MAX_TORQUES = [14, 30, 13, 14, 1, 3.5, 0.2]
_C.MAX_TORQUES = [42, 90, 39, 42, 3, 12, 1]


def get_yumi_arm_cfg():
    return _C.clone()
```

airobot.cfgs.assets.yumi_dual_arm

airobot.cfgs.assets.yumi_dual_arm.**get_yumi_dual_arm_cfg**()

```
from yacs.config import CfgNode as CN

```

(continues on next page)

(continued from previous page)

```
from airobot.cfgs.assets.default_configs import get_cfg_defaults
from airobot.cfgs.assets.yumi_arm import get_yumi_arm_cfg

_C = get_cfg_defaults()
# whether the robot has an arm or not
_C.HAS_ARM = True
# whether the robot has a camera or not
_C.HAS_CAMERA = True
# whether the robot has a end effector tool or not
_C.HAS_EETOOL = False

_C.ROBOT_DESCRIPTION = '/robot_description'

# prefix of the class name of the ARM
# if it's for pybullet simulation, the name will
# be augemented to be '<Prefix>Pybullet'
# if it's for the real robot, the name will be
# augmented to be '<Prefix>Real'
_C.ARM = get_yumi_arm_cfg()
_C.ARM.CLASS = 'Yumi'

_C.ARM.RIGHT = CN()
_C.ARM.RIGHT.HAS_EETOOL = _C.HAS_EETOOL
_C.ARM.RIGHT.HAS_CAMERA = _C.HAS_CAMERA
_C.ARM.RIGHT.HAS_ARM = _C.HAS_ARM

_C.ARM.RIGHT.ARM = get_yumi_arm_cfg()

_C.ARM.RIGHT.ARM.NAME = 'right'
_C.ARM.RIGHT.ARM.JOINT_NAMES = [
    'yumi_joint_1_r', 'yumi_joint_2_r', 'yumi_joint_7_r',
    'yumi_joint_3_r', 'yumi_joint_4_r', 'yumi_joint_5_r',
    'yumi_joint_6_r'
]
# _C.ARM.RIGHT.ARM.MAX_TORQUES = [14, 30, 0.2, 13, 14, 1, 3.5]

_C.ARM.RIGHT.ARM.ROBOT_EE_FRAME = 'yumi_link_7_r'
_C.ARM.RIGHT.ARM.ROBOT_EE_FRAME_JOINT = 'yumi_joint_6_r'
_C.ARM.RIGHT.ARM.HOME_POSITION = [
    0.413, -1.325, -1.040, -0.053, -0.484, 0.841, -1.546]

_C.ARM.LEFT = CN()
_C.ARM.LEFT.HAS_EETOOL = _C.HAS_EETOOL
_C.ARM.LEFT.HAS_CAMERA = _C.HAS_CAMERA
_C.ARM.LEFT.HAS_ARM = _C.HAS_ARM

_C.ARM.LEFT.ARM = get_yumi_arm_cfg()

_C.ARM.LEFT.ARM.NAME = 'left'
_C.ARM.LEFT.ARM.JOINT_NAMES = [
    'yumi_joint_1_l', 'yumi_joint_2_l', 'yumi_joint_7_l',
    'yumi_joint_3_l', 'yumi_joint_4_l', 'yumi_joint_5_l',
    'yumi_joint_6_l'
]
# _C.ARM.LEFT.ARM.MAX_TORQUES = [14, 30, 0.2, 13, 14, 1, 3.5]

_C.ARM.LEFT.ARM.ROBOT_EE_FRAME = 'yumi_link_7_l'
```

(continues on next page)

(continued from previous page)

```
_C.ARM.LEFT.ARM.ROBOT_EE_FRAME_JOINT = 'yumi_joint_6_l'
_C.ARM.LEFT.ARM.HOME_POSITION = [
    -0.473, -1.450, 1.091, 0.031, 0.513, 0.77, -1.669]


def get_yumi_dual_arm_cfg():
    return _C.clone()
```

2.6 airobot.utils

2.6.1 airobot.utils.ai_logger

class airobot.utils.ai_logger.**Logger**(*log_level*)

Bases: object

A logger class.

Parameters **log_level** (*str*) – the following modes are supported: *debug*, *info*, *warn*, *error*, *critical*.

critical(*msg*)

Logging critical information

Parameters **msg** (*str*) – message to log

debug(*msg*)

Logging debug information

Parameters **msg** (*str*) – message to log

error(*msg*)

Logging error information

Parameters **msg** (*str*) – message to log

info(*msg*)

Logging info information

Parameters **msg** (*str*) – message to log

set_level(*log_level*)

Set logging level

Parameters **log_level** (*str*) – the following modes are supported: *debug*, *info*, *warn*, *error*, *critical*

warning(*msg*)

Logging warning information

Parameters **msg** (*str*) – message to log

2.6.2 airobot.utils.arm_util

airobot.utils.arm_util.**reach_ee_goal**(*pos*, *ori*, *get_func*, *pos_tol=0.01*, *ori_tol=0.02*)

Check if end effector reached goal or not. Returns true if both position and orientation goals have been reached within specified tolerance.

Parameters

- **pos** (*list np.ndarray*) – goal position.
- **ori** (*list or np.ndarray*) – goal orientation. It can be: **quaternion** ([qx, qy, qz, qw], shape: $[4]$) **rotation matrix** (shape: $[3, 3]$) **euler angles** ([roll, pitch, yaw], shape: $[3]$).
- **get_func** (*function*) – name of the function with which we can get the current end effector pose.
- **pos_tol** (*float*) – tolerance of position error.
- **ori_tol** (*float*) – tolerance of orientation error.

Returns *bool* – If goal pose is reached or not.

airobot.utils.arm_util.**reach_jnt_goal**(*goal*, *get_func*, *joint_name=None*, *max_error=0.01*)

Check if the joint reached the goal or not. The goal can be a desired velocity(s) or a desired position(s).

Parameters

- **goal** (*np.ndarray*) – goal positions or velocities.
- **get_func** (*function*) – name of the function with which we can get the current joint values.
- **joint_name** (*str*) – if it's none, all the actuated joints are compared. Otherwise, only the specified joint is compared.
- **max_error** (*float*) – tolerance of error.

Returns *bool* – if the goal is reached or not.

airobot.utils.arm_util.**wait_to_reach_ee_goal**(*pos*, *ori*, *get_func*, *get_func_derv=None*, *timeout=10.0*, *pos_tol=0.01*, *ori_tol=0.02*)

Block the code to wait for the end effector to reach its specified goal pose (must be below both position and orientation threshold).

Parameters

- **pos** (*list*) – goal position.
- **ori** (*list or np.ndarray*) – goal orientation. It can be: **quaternion** ([qx, qy, qz, qw], shape: $[4]$) **rotation matrix** (shape: $[3, 3]$) **euler angles** ([roll, pitch, yaw], shape: $[3]$).
- **get_func** (*function*) – name of the function with which we can get the end effector pose.
- **get_func_derv** (*function*) – the name of the function with which we can get end effector velocities.
- **timeout** (*float*) – maximum waiting time.
- **pos_tol** (*float*) – tolerance of position error.
- **ori_tol** (*float*) – tolerance of orientation error.

Returns *bool* – If end effector reached goal or not.

airobot.utils.arm_util.**wait_to_reach_jnt_goal**(*goal*, *get_func*, *joint_name=None*, *get_func_derv=None*, *timeout=10.0*, *max_error=0.01*)

Block the code to wait for the joint moving to the specified goal. The goal can be a desired velocity(s) or a desired position(s). if get_func returns the poisitions (velocities), then the get_func_derv should return the velocities (accelerations) (if provided). If the robot cannot reach the goal, providing get_func_derv can prevent the program from waiting until timeout. It uses the derivative information to make a faster judgement.

Parameters

- **goal** (*float or list*) – goal positions or velocities.
- **get_func** (*function*) – name of the function with which we can get the current joint values.
- **joint_name** (*str*) – if it's none, all the actuated joints are compared. Otherwise, only the specified joint is compared.W
- **get_func_derv** (*function*) – the name of the function with which we can get the derivative of the joint values.
- **timeout** (*float*) – maximum waiting time.
- **max_error** (*float*) – tolerance of error.

Returns *bool* – if the goal is reached or not.

2.6.3 airobot.utils.common

airobot.utils.common.**ang_in_mpi_ppi**(*angle*)
Convert the angle to the range [-pi, pi).

Parameters **angle** (*float*) – angle in radians.

Returns *float* – equivalent angle in [-pi, pi).

airobot.utils.common.**clamp**(*n*, *minn*, *maxn*)
Clamp the input value to be in [minn, maxn].

Parameters

- **n** (*float or int*) – input value.
- **minn** (*float or int*) – minimum value.
- **maxn** (*float or int*) – maximum value.

Returns *float or int* – clamped value.

airobot.utils.common.**create_folder**(*path*, *delete=True*)
Create a new folder.

Parameters

- **path** (*str*) – path of the folder.
- **delete** (*bool*) – if delete=True, then if the path already exists, the folder will be deleted and recreated.

airobot.utils.common.**create_se3**(*ori*, *trans=None*)

Parameters

- **ori** (*np.ndarray*) – orientation in any following form: rotation matrix (shape: $[3, 3]$) quaternion (shape: $[4]$) euler angles (shape: $[3]$).
- **trans** (*np.ndarray*) – translational vector (shape: $[3]$)

Returns *np.ndarray* – a transformation matrix (shape: $[4, 4]$)

airobot.utils.common.**euler2quat**(*euler*, *axes='xyz'*)
Convert euler angles to quaternion.

Parameters

- **euler** (*list or np.ndarray*) – euler angles (shape: $[3,]$).
- **axes** (*str*) – Specifies sequence of axes for rotations. 3 characters belonging to the set {'X', 'Y', 'Z'} for intrinsic rotations (rotation about the axes of a coordinate system XYZ attached to a moving body), or {'x', 'y', 'z'} for extrinsic rotations (rotation about the axes of the fixed coordinate system).

Returns *np.ndarray* – quaternion [x,y,z,w] (shape: $[4,]$).

airobot.utils.common.**euler2rot**(*euler*, *axes='xyz'*)
Convert euler angles to rotation matrix.

Parameters

- **euler** (*list or np.ndarray*) – euler angles (shape: $[3,]$).
- **axes** (*str*) – Specifies sequence of axes for rotations. 3 characters belonging to the set {'X', 'Y', 'Z'} for intrinsic rotations (rotation about the axes of a coordinate system XYZ attached to a moving body), or {'x', 'y', 'z'} for extrinsic rotations (rotation about the axes of the fixed coordinate system).

Returns *np.ndarray* – rotation matrix (shape: $[3, 3]$).

airobot.utils.common.**linear_interpolate_path**(*start_pos*, *delta_xyz*, *interval*)
Linear interpolation in a path.

Parameters

- **start_pos** (*list or np.ndarray*) – start position ([x, y, z], shape: $[3]$).
- **delta_xyz** (*list or np.ndarray*) – movement in x, y, z directions (shape: $[3,]$).
- **interval** (*float*) – interpolation interval along delta_xyz. Interpolate a point every *interval* distance between the two end points.

Returns *np.ndarray* – waypoints along the path (shape: $[N, 3]$).

airobot.utils.common.**list_class_names**(*dir_path*)
Return the mapping of class names in all files in dir_path to their file path.

Parameters **dir_path** (*str*) – absolute path of the folder.

Returns *dict* – mapping from the class names in all python files in the folder to their file path.

airobot.utils.common.**load_class_from_path**(*cls_name*, *path*)
Load a class from the file path.

Parameters

- **cls_name** (*str*) – class name.
- **path** (*str*) – python file path.

Returns *Python Class* – return the class A which is named as cls_name. You can call A() to create an instance of this class using the return value.

airobot.utils.common.**print_blue**(*skk*)
print the text in blue color.

Parameters **skk** (*str*) – text to be printed.

airobot.utils.common.**print_cyan**(*skk*)
print the text in cyan color.

Parameters **skk** (*str*) – text to be printed.

airobot.utils.common.**print_green**(*skk*)
print the text in green color.

Parameters **skk** (*str*) – text to be printed.

airobot.utils.common.**print_purple**(*skk*)
print the text in purple color.

Parameters **skk** (*str*) – text to be printed.

airobot.utils.common.**print_red**(*skk*)
print the text in red color.

Parameters **skk** (*str*) – text to be printed.

airobot.utils.common.**print_yellow**(*skk*)
print the text in yellow color.

Parameters **skk** (*str*) – text to be printed.

airobot.utils.common.**quat2euler**(*quat*, *axes='xyz'*)
Convert quaternion to euler angles.

Parameters

- **quat** (*list or np.ndarray*) – quaternion [x,y,z,w] (shape: $[4,]$).
- **axes** (*str*) – Specifies sequence of axes for rotations. 3 characters belonging to the set {'X', 'Y', 'Z'} for intrinsic rotations (rotation about the axes of a coordinate system XYZ attached to a moving body), or {'x', 'y', 'z'} for extrinsic rotations (rotation about the axes of the fixed coordinate system).

Returns *np.ndarray* – euler angles (shape: $[3,]$).

airobot.utils.common.**quat2rot**(*quat*)
Convert quaternion to rotation matrix.

Parameters **quat** (*list or np.ndarray*) – quaternion [x,y,z,w] (shape: $[4,]$).

Returns *np.ndarray* – rotation matrix (shape: $[3, 3]$).

airobot.utils.common.**quat2rotvec**(*quat*)
Convert quaternion to rotation vector.

Parameters **quat** (*list or np.ndarray*) – quaternion [x,y,z,w] (shape: $[4,]$).

Returns *np.ndarray* – rotation vector (shape: $[3,]$).

airobot.utils.common.**quat_inverse**(*quat*)
Return the quaternion inverse.

Parameters **quat** (*list or np.ndarray*) – quaternion [x,y,z,w] (shape: $[4,]$).

Returns *np.ndarray* – inverse quaternion (shape: $[4,]$).

airobot.utils.common.**quat_multiply**(*quat1*, *quat2*)
Quaternion mulitplication.

Parameters

- **quat1** (*list or np.ndarray*) – first quaternion [x,y,z,w] (shape: $[4,]$).
- **quat2** (*list or np.ndarray*) – second quaternion [x,y,z,w] (shape: $[4,]$).

Returns *np.ndarray* – quat1 * quat2 (shape: $[4,]$).

airobot.utils.common.**rot2euler**(*rot*, *axes='xyz'*)
Convert rotation matrix to euler angles.

Parameters

- **rot** (*np.ndarray*) – rotation matrix (shape: $[3, 3]$).
- **axes** (*str*) – Specifies sequence of axes for rotations. 3 characters belonging to the set {'X', 'Y', 'Z'} for intrinsic rotations (rotation about the axes of a coordinate system XYZ attached to a moving body), or {'x', 'y', 'z'} for extrinsic rotations (rotation about the axes of the fixed coordinate system).

Returns *np.ndarray* – euler angles (shape: $[3,]$).

airobot.utils.common.**rot2quat**(*rot*)
Convert rotation matrix to quaternion.

Parameters **rot** (*np.ndarray*) – rotation matrix (shape: $[3, 3]$).

Returns *np.ndarray* – quaternion [x,y,z,w] (shape: $[4,]$).

airobot.utils.common.**rot2rotvec**(*rot*)
Convert rotation matrix to quaternion.

Parameters **rot** (*np.ndarray*) – rotation matrix (shape: $[3, 3]$).

Returns *np.ndarray* – a rotation vector (shape: $[3,]$).

airobot.utils.common.**rotvec2euler**(*vec*, *axes='xyz'*)
A rotation vector is a 3 dimensional vector which is co-directional to the axis of rotation and whose norm gives the angle of rotation (in radians).

Parameters

- **vec** (*list or np.ndarray*) – a rotational vector. Its norm represents the angle of rotation.
- **axes** (*str*) – Specifies sequence of axes for rotations. 3 characters belonging to the set {'X', 'Y', 'Z'} for intrinsic rotations (rotation about the axes of a coordinate system XYZ attached to a moving body), or {'x', 'y', 'z'} for extrinsic rotations (rotation about the axes of the fixed coordinate system).

Returns *np.ndarray* – euler angles (shape: $[3,]$).

airobot.utils.common.**rotvec2quat**(*vec*)
A rotation vector is a 3 dimensional vector which is co-directional to the axis of rotation and whose norm gives the angle of rotation (in radians).

Parameters **vec** (*list or np.ndarray*) – a rotational vector. Its norm represents the angle of rotation.

Returns *np.ndarray* – quaternion [x,y,z,w] (shape: $[4,]$).

airobot.utils.common.**rotvec2rot**(*vec*)
A rotation vector is a 3 dimensional vector which is co-directional to the axis of rotation and whose norm gives the angle of rotation (in radians).

Parameters **vec** (*list or np.ndarray*) – a rotational vector. Its norm represents the angle of rotation.

Returns *np.ndarray* – rotation matrix (shape: $[3, 3]$).

airobot.utils.common.**se3_to_trans_ori**(*se3*, *ori='quat'*, *axes='xyz'*)

Parameters

- **se3** (*np.ndarray*) – a SE(3) matrix (shape: $[4, 4]$)
- **ori** (*str*) – orientation format, can be one of ['quat', 'euler', 'matrix', 'rotvec']
- **axes** (*str*) – only used when ori == 'euler'

Returns

2-element tuple containing

- np.ndarray: translational vector (shape: $[3,]$).
- np.ndarray: rotational vector/matrix.

airobot.utils.common.**to_euler_angles**(*ori*, *axes='xyz'*)

Convert orientation in any form (rotation matrix, quaternion, or euler angles) to euler angles (roll, pitch, yaw).

Parameters

- **ori** (*list or np.ndarray*) – orientation in any following form: rotation matrix (shape: $[3, 3]$) quaternion (shape: $[4]$) euler angles (shape: $[3]$).
- **axes** (*str*) – Specifies sequence of axes for rotations. 3 characters belonging to the set {'X', 'Y', 'Z'} for intrinsic rotations (rotation about the axes of a coordinate system XYZ attached to a moving body), or {'x', 'y', 'z'} for extrinsic rotations (rotation about the axes of the fixed coordinate system).

Returns

np.ndarray –

euler angles (shape: $[3,]$**).** By default, it's [roll, pitch, yaw]

airobot.utils.common.**to_quat**(*ori*)

Convert orientation in any form (rotation matrix, quaternion, or euler angles) to quaternion.

Parameters **ori** (*list or np.ndarray*) – orientation in any following form: rotation matrix (shape: $[3, 3]$) quaternion (shape: $[4]$) euler angles (shape: $[3]$).

Returns *np.ndarray* – quaternion [x, y, z, w](shape: $[4,]$).

airobot.utils.common.**to_rot_mat**(*ori*)

Convert orientation in any form (rotation matrix, quaternion, or euler angles) to rotation matrix.

Parameters **ori** (*list or np.ndarray*) – orientation in any following form: rotation matrix (shape: $[3, 3]$) quaternion (shape: $[4]$) euler angles (shape: $[3]$).

Returns *np.ndarray* – orientation matrix (shape: $[3, 3]$).

2.6.4 airobot.utils.moveit_util

2.6.5 airobot.utils.ros_util

2.6.6 airobot.utils.urscript_util

Tools for creating URScript messages, for communicating with the real UR robots over TCP/IP by creating URScript programs and sending them for execution on the robot's system

built off of urscript.py, part of python-urx library (https://github.com/SintefManufacturing/python-urx)

class airobot.utils.urscript_util.**Robotiq2F140URScript**(*socket_host*, *socket_port*, *socket_name*)

Bases: *airobot.utils.urscript_util.URScript*

Class for creating Robotiq 2F140 specific URScript messages to send to the UR robot, for setting gripper related variables

Parameters

- **socket_host** (*str*) – gripper IP address used by the UR controller
- **socket_port** (*int*) – gripper communication port used by the UR controller
- **socket_name** (*str*) – name of the socket connection

set_activate()
Activate the gripper, by setting some internal variables on the UR controller to 1

set_gripper_force(*force*)
Set maximum gripper force

Parameters **force** (*int*) – Desired maximum gripper force, ranges from 0-255

set_gripper_position(*position*)
Control the gripper position by setting internal position variable to desired position value on UR controller

Parameters **position** (*int*) – Position value, ranges from 0-255

set_gripper_speed(*speed*)
Set what speed the gripper should move

Parameters **speed** (*int*) – Desired gripper speed, ranges from 0-255

class airobot.utils.urscript_util.**URScript**
Bases: object

Class for creating urscript programs to send to the UR5 controller

__init__()
Constructor, each urscript has a header and a program that runs on the contoller

constrain_unsigned_char(*value*)
Ensure that unsigned char values are constrained to between 0 and 255.

Parameters **value** (*int*) – Value to ensure is between 0 and 255

reset()
Reset the urscript to empty

sleep(*value*)
Add a sleep command to urscript program, sleep for a specified amount of time

Parameters **value** (*float*) – Amount of time in seconds for program to sleep

socket_close(*socket_name*)
Add a close socket command to urscript program.

Parameters **socket_name** (*str*) – Name of socket to close (must be same as name of socket that was opened previously)

socket_get_var(*var*, *socket_name*)
Add a command to the program with communicates over a socket connection to get the value of a variable

Parameters

- **var** (*str*) – Name of the variable to obtain the value of
- **socket_name** (*str*) – Which socket connection to use for getting the value (must be same as name of socket that was opened previously)

socket_open(*socket_host*, *socket_port*, *socket_name*)
Add a open socket command to urscript program with specified host, port, and name

Parameters

- **socket_host** (*str*) – Host address
- **socket_port** (*int*) – Port to open
- **socket_name** (*str*) – Name of the socket to use when interacting with the socket after it is opened

socket_set_var(*var*, *value*, *socket_name*)
Add a command to the program with communicates over a socket connection to set the value of a variable

Parameters

- **var** (*str*) – Name of the variable to obtain the value of
- **value** (*int*) – Value to set the variable to
- **socket_name** (*str*) – Which socket connection to use for getting the value (must be same as name of socket that was opened previously)

sync()
Add a sync command to the myProg() urscript program

2.6.7 airobot.utils.pb_util

class airobot.utils.pb_util.**BulletClient**(*connection_mode=None*, *realtime=False*, *opengl_render=True*, *options=''*)

Bases: object

A wrapper for pybullet to manage different clients.

Parameters

- **connection_mode** (*pybullet mode*) – *None* connects to an existing simulation or, if fails, creates a new headless simulation, *pybullet.GUI* creates a new simulation with a GUI, *pybullet.DIRECT* creates a headless simulation, *pybullet.SHARED_MEMORY* connects to an existing simulation.
- **realtime** (*bool*) – whether to use realtime mode or not.
- **opengl_render** (*bool*) – use OpenGL (hardware renderer) to render RGB images.

get_body_state(*body_id*)
Get the body state.

Parameters **body_id** (*int*) – body index.

Returns

4-element tuple containing

- np.ndarray: x, y, z position of the body (shape: $[3,]$).
- np.ndarray: quaternion representation ([qx, qy, qz, qw]) of the body orientation (shape: $[4,]$).
- np.ndarray: linear velocity of the body (shape: $[3,]$).
- np.ndarray: angular velocity of the body (shape: $[3,]$).

get_client_id()
Return the pybullet client id.

Returns *int* – pybullet client id.

in_realtime_mode()
Check if the pybullet simulation is in step simulation mode or realtime simulation mode.

Returns *bool* – whether the pybullet simulation is in step simulation mode or realtime simulation mode.

load_geom(*shape_type*, *size=None*, *mass=0.5*, *visualfile=None*, *collifile=None*, *mesh_scale=None*, *rgba=None*, *specular=None*, *shift_pos=None*, *shift_ori=None*, *base_pos=None*, *base_ori=None*, *no_collision=False*, ***kwargs*)
Load a regular geometry (*sphere*, *box*, *capsule*, *cylinder*, *mesh*).

Note: Please do not call **load_geom('capsule')** when you are using **robotiq gripper**. The capsule generated will be in wrong size if the mimicing thread (_th_mimic_gripper) in the robotiq gripper class starts running. This might be a PyBullet Bug (current version is 2.5.6). Note that other geometries(box, sphere, cylinder, etc.) are not affected by the threading in the robotiq gripper.

Parameters

- **shape_type** (*str*) – one of [*sphere*, *box*, *capsule*, *cylinder*, *mesh*].
- **size** (*float or list*) – Defaults to None.

  If shape_type is sphere: size should be a float (radius).

  If shape_type is capsule or cylinder: size should be a 2-element list (radius, length).

  If shape_type is box: size can be a float (same half edge length for 3 dims) or a 3-element list containing the half size of 3 edges

  size doesn't take effect if shape_type is mesh.
- **mass** (*float*) – mass of the object in kg. If mass=0, then the object is static.
- **visualfile** (*str*) – path to the visual mesh file. only needed when the shape_type is mesh. If it's None, same collision mesh file will be used as the visual mesh file.
- **collifile** (*str*) – path to the collision mesh file. only needed when the shape_type is mesh. If it's None, same viusal mesh file will be used as the collision mesh file.
- **mesh_scale** (*float or list*) – scale the mesh. If it's a float number, the mesh will be scaled in same ratio along 3 dimensions. If it's a list, then it should contain 3 elements (scales along 3 dimensions).
- **rgba** (*list*) – color components for red, green, blue and alpha, each in range [0, 1] (shape: $[4,]$).
- **specular** (*list*) – specular reflection color components for red, green, blue and alpha, each in range [0, 1] (shape: $[4,]$).
- **shift_pos** (*list*) – translational offset of collision shape, visual shape, and inertial frame (shape: $[3,]$).
- **shift_ori** (*list*) – rotational offset (quaternion [x, y, z, w]) of collision shape, visual shape, and inertial frame (shape: $[4,]$).
- **base_pos** (*list*) – cartesian world position of the base (shape: $[3,]$).

- **base_ori** (*list*) – cartesian world orientation of the base as quaternion [x, y, z, w] (shape: $[4,]$).

Returns *int* – a body unique id, a non-negative integer value or -1 for failure.

load_mjcf(*filename*, ***kwargs*)
Load SDF into the pybullet client.

Parameters **filename** (*str*) – a relative or absolute path to the MJCF file on the file system of the physics server.

Returns *int* – a body unique id, a non-negative integer value. If the MJCF file cannot be loaded, this integer will be negative and not a valid body unique id.

load_sdf(*filename*, *scaling=1.0*, ***kwargs*)
Load SDF into the pybullet client.

Parameters

- **filename** (*str*) – a relative or absolute path to the SDF file on the file system of the physics server.
- **scaling** (*float*) – apply a scale factor to the SDF model.

Returns *int* – a body unique id, a non-negative integer value. If the SDF file cannot be loaded, this integer will be negative and not a valid body unique id.

load_urdf(*filename*, *base_pos=None*, *base_ori=None*, *scaling=1.0*, ***kwargs*)
Load URDF into the pybullet client.

Parameters

- **filename** (*str*) – a relative or absolute path to the URDF file on the file system of the physics server.
- **base_pos** (*list or np.ndarray*) – create the base of the object at the specified position in world space coordinates [X,Y,Z]. Note that this position is of the URDF link position.
- **base_ori** (*list or np.ndarray*) – create the base of the object at the specified orientation as world space quaternion [X,Y,Z,W].
- **scaling** (*float*) – apply a scale factor to the URDF model.

Returns *int* – a body unique id, a non-negative integer value. If the URDF file cannot be loaded, this integer will be negative and not a valid body unique id.

remove_body(*body_id*)
Delete body from the simulation.

Parameters **body_id** (*int*) – body index.

Returns *bool* – whether the body is removed.

reset_body(*body_id*, *base_pos*, *base_quat=None*, *lin_vel=None*, *ang_vel=None*)
Reset body to the specified pose and specified initial velocity.

Parameters

- **body_id** (*int*) – body index.
- **base_pos** (*list or np.ndarray*) – position [x,y,z] of the body base.
- **base_ori** (*list or np.ndarray*) – quaternion [qx, qy, qz, qw] of the body base.
- **lin_vel** (*list or np.ndarray*) – initial linear velocity if provided.

- **ang_vel** (*list or np.ndarray*) – initial angular velocity if provided.

Returns:

set_step_sim(*step_mode=True*)
Turn on/off the realtime simulation mode.

Parameters **step_mode** (*bool*) – run the simulation in step mode if it is True. Otherwise, the simulation will be in realtime.

class airobot.utils.pb_util.**TextureModder**(*pb_client_id*)
Bases: object

Modify textures in model.

Parameters **pb_client_id** (*int*) – pybullet client id.

Variables

- **texture_dict** (*dict*) – a dictionary that tells the texture of a link on a body.
- **texture_files** (*list*) – a list of texture files (usuallly images).

rand_all(*body_id*, *link_id*)
Randomize color, gradient, noise, texture of the specified link.

Parameters

- **body_id** (*int*) – body index.
- **link_id** (*int*) – link index in the body.

rand_gradient(*body_id*, *link_id*)
Randomize the gradient of the color of the link.

Parameters

- **body_id** (*int*) – body index.
- **link_id** (*int*) – link index in the body.

rand_noise(*body_id*, *link_id*)
Randomly add noise to the foreground.

Parameters

- **body_id** (*int*) – body index.
- **link_id** (*int*) – link index in the body.

rand_rgb(*body_id*, *link_id*)
Randomize the color of the link.

Parameters

- **body_id** (*int*) – body index.
- **link_id** (*int*) – link index in the body.

rand_texture(*body_id*, *link_id*)
Randomly apply a texture to the link. Call *set_texture_path* first to set the root path to the texture files.

Parameters

- **body_id** (*int*) – body index.
- **link_id** (*int*) – link index in the body.

randomize(*mode='all'*, *exclude=None*)
Randomize all the links in the scene.

Parameters

- **mode** (*str*) – one of *all*, *rgb*, *noise*, *gradient*.
- **exclude** (*dict*) – exclude bodies or links from randomization. *exclude* is a dict with body_id as the key, and a list of link ids as the value. If the value (link ids) is an empty list, then all links on the body will be excluded.

set_gradient(*body_id*, *link_id*, *rgb1*, *rgb2*, *vertical=True*)
Creates a linear gradient from rgb1 to rgb2.

Parameters

- **body_id** (*int*) – body index.
- **link_id** (*int*) – link index in the body.
- **rgb1** (*list or np.ndarray*) – first rgb color (shape: $[3,]$).
- **rgb2** (*list or np.ndarray*) – second rgb color (shape: $[3,]$).
- **vertical** (*bool*) – if True, the gradient in the vertical direction, if False it's in the horizontal direction.

set_noise(*body_id*, *link_id*, *rgb1*, *rgb2*, *fraction=0.9*)
Apply noise to the texture.

Parameters

- **body_id** (*int*) – body index.
- **link_id** (*int*) – link index in the body.
- **rgb1** (*list or np.ndarray*) – background rgb color (shape: $[3,]$).
- **rgb2** (*list or np.ndarray*) – color of random noise foreground color (shape: $[3,]$).
- **fraction** (*float*) – fraction of pixels with foreground color.

set_rgba(*body_id*, *link_id*, *rgba*)
Set color to the link.

Parameters

- **body_id** (*int*) – body index.
- **link_id** (*int*) – link index in the body.
- **rgba** (*list or np.ndarray*) – red, green, blue, alpha channel (opacity of the color), (shape: $[4,]$).

set_texture(*body_id*, *link_id*, *texture_file*)
Apply texture to a link. You can download texture files from: https://www.robots.ox.ac.uk/~vgg/data/dtd/index.html.

Parameters

- **body_id** (*int*) – body index.
- **link_id** (*int*) – link index in the body.
- **texture_file** (*str*) – path to the texture files (image, supported format: *jpg*, *png*, *jpeg*, *tga*, or *gif* etc.).

set_texture_path(*path*)
Set the root path to the texture files. It will search all the files in the folder (including subfolders), and find all files that end with *.png*, *jpg*, *jpeg*, *tga*, or *gif*.

Parameters **path** (*str*) – root path to the texture files.

whiten_materials(*body_id=None*, *link_id=None*)
Helper method for setting all material colors to white.

Parameters

- **body_id** (*int*) – unique body id when you load the body. If body_id is not provided, all the bodies will be whitened.
- **link_id** (*int*) – the index of the link on the body. If link_id is not provided and body_id is provided, all the links of the body will be whitened.

airobot.utils.pb_util.**create_pybullet_client**(*gui=True*, *realtime=True*, *opengl_render=True*)
Create a pybullet simulation client.

Parameters

- **gui** (*bool*) – use GUI mode or non-GUI mode.
- **realtime** – use realtime simulation or step simuation.
- **opengl_render** (*bool*) – use OpenGL (hardware renderer) to render RGB images.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

a

Index

Symbols

A

B

C

D

E

F

G

I

L

M

O

P

Q

R

S

T

U

W

Y