PyRobot Documentation

Release 0.0.1

Lerrel Pinto

Jun 21, 2019

Contents

Here lies API documentation for PyRobot. For tutorials on using PyRobot please refer to the PyRobot.

CHAPTER 1

Core API for PyRobot

PyRobot is built around the following core classes that encapsulate different components of a robot.

1.1 Robot

class pyrobot.core.**Robot**(*robot_name*, *use_arm=True*, *use_base=True*, *use_camera=True*, *use_gripper=True*, *arm_config={}*, *base_config={}*, *camera_config={}*, *gripper_config={}*)

This is the main interface class that is composed of key robot modules (base, arm, gripper, and camera). This class builds robot specific objects by reading a configuration and instantiating the necessary robot module objects.

__init__(*robot_name*, *use_arm=True*, *use_base=True*, *use_camera=True*, *use_gripper=True*, *arm_config={}*, *base_config={}*, *camera_config={}*, *gripper_config={}*)

Constructor for the Robot class

Parameters

- **robot_name** (*string*) – robot name
- **use_arm** (*bool*) – use arm or not
- **use_base** (*bool*) – use base or not
- **use_camera** (*bool*) – use camera or not
- **use_gripper** (*bool*) – use gripper or not
- **arm_config** (*dict*) – configurations for arm
- **base_config** (*dict*) – configurations for base
- **camera_config** (*dict*) – configurations for camera
- **gripper_config** (*dict*) – configurations for gripper

1.2 Base

class pyrobot.core.**Base**(*configs*)

Bases: object

This is a parent class on which the robot specific Base classes would be built.

__init__(*configs*)

The consturctor for Base class.

Parameters **configs** (*YACS CfgNode*) – configurations for base

get_state(*state_type*)

Returns the requested base pose in the (x,y, yaw) format.

Parameters **state_type** (*string*) – Requested state type. Ex: Odom, SLAM, etc

Returns pose: pose of the form [x, y, yaw]

Return type list

go_to_absolute(*xyt_position*, *use_map*, *close_loop*, *smooth*)

Moves the robot to the robot to given goal state in the world frame.

Parameters

- **xyt_position** (*list*) – The goal state of the form (x,y,t) in the world (map) frame.
- **use_map** (*bool*) – When set to "True", ensures that controler is using only free space on the map to move the robot.
- **close_loop** (*bool*) – When set to "True", ensures that controler is operating in open loop by taking account of odometry.
- **smooth** (*bool*) – When set to "True", ensures that the motion leading to the goal is a smooth one.

go_to_relative(*xyt_position*, *use_map*, *close_loop*, *smooth*)

Moves the robot to the robot to given goal state relative to its initial pose.

Parameters

- **xyt_position** (*list*) – The relative goal state of the form (x,y,t)
- **use_map** (*bool*) – When set to "True", ensures that controler is using only free space on the map to move the robot.
- **close_loop** (*bool*) – When set to "True", ensures that controler is operating in open loop by taking account of odometry.
- **smooth** (*bool*) – When set to "True", ensures that the motion leading to the goal is a smooth one.

set_vel(*fwd_speed*, *turn_speed*, *exe_time=1*)

Set the moving velocity of the base

Parameters

- **fwd_speed** – forward speed
- **turn_speed** – turning speed
- **exe_time** – execution time

stop()

Stop the base

track_trajectory (*states*, *controls*, *close_loop*)
State trajectory that the robot should track.

Parameters

- **states** (*list*) – sequence of (x,y,t) states that the robot should track.
- **controls** (*list*) – optionally specify control sequence as well.
- **close_loop** (*bool*) – whether to close loop on the computed control sequence or not.

1.3 Arm

class pyrobot.core.**Arm** (*configs*, *moveit_planner='ESTkConfigDefault'*, *analytical_ik=None*, *use_moveit=True*)
Bases: object

This is a parent class on which the robot specific Arm classes would be built.

__init__ (*configs*, *moveit_planner='ESTkConfigDefault'*, *analytical_ik=None*, *use_moveit=True*)
Constructor for Arm parent class.

Parameters

- **configs** (*YACS CfgNode*) – configurations for arm
- **moveit_planner** (*string*) – moveit planner type
- **analytical_ik** (*None or an Analytical ik class*) – customized analytical ik class if you have one, None otherwise
- **use_moveit** (*bool*) – use moveit or not, default is True

compute_fk_position (*joint_positions*, *des_frame*)
Given joint angles, compute the pose of desired_frame with respect to the base frame (self.configs.ARM.ARM_BASE_FRAME). The desired frame must be in self.arm_link_names

Parameters

- **joint_positions** (*np.ndarray*) – joint angles
- **des_frame** (*string*) – desired frame

Returns translational vector and rotational matrix

Return type np.ndarray, np.ndarray

compute_fk_velocity (*joint_positions*, *joint_velocities*, *des_frame*)
Given joint_positions and joint velocities, compute the velocities of des_frame with respect to the base frame

Parameters

- **joint_positions** (*np.ndarray*) – joint positions
- **joint_velocities** (*np.ndarray*) – joint velocities
- **des_frame** (*string*) – end frame

Returns translational and rotational velocities (vx, vy, vz, wx, wy, wz)

Return type np.ndarray

compute_ik (*position*, *orientation*, *qinit=None*, *numerical=True*)
Inverse kinematics

Parameters

- **position** (*np.ndarray*) – end effector position (shape: $[3,]$)
- **orientation** (*np.ndarray*) – end effector orientation. It can be quaternion ([x,y,z,w], shape: $[4,]$), euler angles (yaw, pitch, roll angles (shape: $[3,]$)), or rotation matrix (shape: $[3,3]$)
- **qinit** (*list*) – initial joint positions for numerical IK
- **numerical** (*bool*) – use numerical IK or analytical IK

Returns None or joint positions

Return type np.ndarray

get_ee_pose(*base_frame*)
Return the end effector pose with respect to the base_frame

Parameters **base_frame** (*string*) – reference frame

Returns

tuple (trans, rot_mat, quat)

trans: translational vector (shape: $[3,1]$)

rot_mat: rotational matrix (shape: $[3,3]$)

quat: rotational matrix in the form of quaternion (shape: $[4,]$)

Return type tuple or ROS PoseStamped

get_jacobian(*joint_angles*)
Return the geometric jacobian on the given joint angles. Refer to P112 in "Robotics: Modeling, Planning, and Control"

Parameters **joint_angles** (*list or flattened np.ndarray*) – joint_angles

Returns jacobian

Return type np.ndarray

get_joint_angle(*joint*)
Return the joint angle of the 'joint'

Parameters **joint** (*string*) – joint name

Returns joint angle

Return type float

get_joint_angles()
Return the joint angles

Returns joint_angles

Return type np.ndarray

get_joint_torque(*joint*)
Return the joint torque of the 'joint'

Parameters **joint** (*string*) – joint name

Returns joint torque

Return type float

get_joint_torques()
Return the joint torques

Returns joint_torques

Return type np.ndarray

get_joint_velocities()
Return the joint velocities

Returns joint_vels

Return type np.ndarray

get_joint_velocity(*joint*)
Return the joint velocity of the 'joint'

Parameters **joint** (*string*) – joint name

Returns joint velocity

Return type float

get_transform(*src_frame*, *dest_frame*)
Return the transform from the src_frame to dest_frame

Parameters

- **src_frame** (*string*) – source frame
- **dest_frame** (*basestring*) – destination frame

Returns

tuple (trans, rot_mat, quat)

trans: translational vector (shape: $[3, 1]$)

rot_mat: rotational matrix (shape: $[3, 3]$)

quat: rotational matrix in the form of quaternion (shape: $[4,]$)

Return type tuple or ROS PoseStamped

go_home()
Reset robot to default home position

move_ee_xyz(*displacement*, *eef_step=0.005*, *numerical=True*, *plan=True*, ***kwargs*)
Keep the current orientation fixed, move the end effector in {xyz} directions

Parameters

- **displacement** (*np.ndarray*) – (delta_x, delta_y, delta_z)
- **eef_step** (*float*) – resolution (m) of the interpolation on the cartesian path
- **numerical** (*bool*) – use numerical inverse kinematics solver or analytical inverse kinematics solver
- **plan** (*bool*) – use moveit the plan a path to move to the desired pose. If False, it will do linear interpolation along the path, and simply use IK solver to find the sequence of desired joint positions and then call *set_joint_positions*

Returns whether the movement is successful or not

Return type bool

pose_ee
Return the end effector pose w.r.t 'ARM_BASE_FRAME'

Returns

trans: translational vector (shape: $[3, 1]$)

rot_mat: rotational matrix (shape: $[3, 3]$)

quat: rotational matrix in the form of quaternion (shape: $[4,]$)

Return type tuple (trans, rot_mat, quat)

set_ee_pose(*position*, *orientation*, *plan=True*, *wait=True*, *numerical=True*, ***kwargs*)
Commands robot arm to desired end-effector pose (w.r.t. 'ARM_BASE_FRAME'). Computes IK solution in joint space and calls set_joint_positions. Will wait for command to complete if wait is set to True.

Parameters

- **position** (*np.ndarray*) – position of the end effector (shape: $[3,]$)
- **orientation** (*np.ndarray*) – orientation of the end effector (can be rotation matrix, euler angles (yaw, pitch, roll), or quaternion) (shape: $[3, 3]$, $[3,]$ or $[4,]$) The convention of the Euler angles here is z-y'-x' (intrinsic rotations), which is one type of Tait-Bryan angles.
- **plan** (*bool*) – use moveit the plan a path to move to the desired pose
- **wait** (*bool*) – wait until the desired pose is achieved
- **numerical** (*bool*) – use numerical inverse kinematics solver or analytical inverse kinematics solver

Returns Returns True if command succeeded, False otherwise

Return type bool

set_joint_positions(*positions*, *plan=True*, *wait=True*, ***kwargs*)
Sets the desired joint angles for all arm joints

Parameters

- **positions** (*list*) – list of length #of joints, angles in radians
- **plan** (*bool*) – whether to use moveit to plan a path. Without planning, there is no collision checking and each joint will move to its target joint position directly.
- **wait** (*bool*) – if True, it will wait until the desired joint positions are achieved

Returns True if successful; False otherwise (timeout, etc.)

Return type bool

set_joint_torques(*torques*, ***kwargs*)
Sets the desired joint torques for all arm joints

Parameters **torques** (*list*) – target joint torques

set_joint_velocities(*velocities*, ***kwargs*)
Sets the desired joint velocities for all arm joints

Parameters **velocities** (*list*) – target joint velocities

1.4 Camera

class pyrobot.core.**Camera**(*configs*)
Bases: object

This is a parent class on which the robot specific Camera classes would be built.

__init__(*configs*)
Constructor for Camera parent class.

Parameters **configs** (*YACS CfgNode*) – configurations for camera

1.5 Gripper

class pyrobot.core.**Gripper**(*configs*)
Bases: object

This is a parent class on which the robot specific Gripper classes would be built.

__init__(*configs*)
Constructor for Gripper parent class.

Parameters **configs** (*YACS CfgNode*) – configurations for gripper

CHAPTER 2

PyRobot wrapper on LoCoBot

Here are the wrapper definitions for using PyRobot with the LoCoBot robot.

2.1 Base

class locobot.base.**LoCoBotBase**(*configs*, *map_img_dir=None*, *base_controller='ilqr'*, *base_planner=None*, *base=None*)

Bases: *pyrobot.core.Base*

This is a common base class for the locobot and locobot-lite base.

__init__(*configs*, *map_img_dir=None*, *base_controller='ilqr'*, *base_planner=None*, *base=None*)

The constructor for LoCoBotBase class.

Parameters

- **configs** (*YACS CfgNode*) – configurations read from config file
- **map_img_dir** (*string*) – parent directory of the saved RGB images and depth images

get_state(*state_type*)

Returns the requested base pose in the (x,y, yaw) format as computed either from Wheel encoder readings or Visual-SLAM

Parameters **state_type** (*string*) – Requested state type. Ex: Odom, SLAM, etc

Returns pose of the form [x, y, yaw]

Return type list

go_to_absolute(*xyt_position*, *use_map=False*, *close_loop=True*, *smooth=False*)

Moves the robot to the robot to given goal state in the world frame.

Parameters

- **xyt_position** (*list or np.ndarray*) – The goal state of the form (x,y,t) in the world (map) frame.

- **use_map** (*bool*) – When set to "True", ensures that controler is using only free space on the map to move the robot.
- **close_loop** (*bool*) – When set to "True", ensures that controler is operating in open loop by taking account of odometry.
- **smooth** (*bool*) – When set to "True", ensures that the motion leading to the goal is a smooth one.

go_to_relative (*xyt_position*, *use_map=False*, *close_loop=True*, *smooth=False*)
Moves the robot to the robot to given goal state relative to its initial pose.

Parameters

- **xyt_position** (*list or np.ndarray*) – The relative goal state of the form (x,y,t)
- **use_map** (*bool*) – When set to "True", ensures that controler is using only free space on the map to move the robot.
- **close_loop** (*bool*) – When set to "True", ensures that controler is operating in open loop by taking account of odometry.
- **smooth** (*bool*) – When set to "True", ensures that the motion leading to the goal is a smooth one.

track_trajectory (*states*, *controls=None*, *close_loop=True*)
State trajectory that the robot should track.

Parameters

- **states** (*list*) – sequence of (x,y,t) states that the robot should track.
- **controls** (*list*) – optionally specify control sequence as well.
- **close_loop** (*bool*) – whether to close loop on the computed control sequence or not.

2.2 Base Controllers

Here are the controller classes for the Base.

class locobot.base_controllers.**ProportionalControl** (*bot_base*, *ctrl_pub*, *configs*)
This class encapsulates and provides interface to a Proportional controller used to control the base

__init__ (*bot_base*, *ctrl_pub*, *configs*)
The constructor for ProportionalControl class.

Parameters

- **configs** (*dict*) – configurations read from config file
- **base_state** (*BaseState*) – an object consisting of an instance of BaseState.
- **ctrl_pub** (*rospy.Publisher*) – a ros publisher used to publish velocity commands to base of the robot.

go_to_absolute (*xyt_position*, *close_loop=True*, *smooth=False*)
Moves the robot to the robot to given goal state in the world frame using proportional control.

Parameters

- **xyt_position** (*list or np.ndarray*) – The goal state of the form (x,y,t) in the world (map) frame.

- **close_loop** (*bool*) – When set to "True", ensures that controler is operating in open loop by taking account of odometry.
- **smooth** (*bool*) – When set to "True", ensures that the motion leading to the goal is a smooth one.

goto (*xyt_position=None*)
Moves the robot to the robot to given goal state in the relative frame (base frame).

Parameters xyt_position (*list*) – The goal state of the form (x,y,t) in the relative (base) frame.

class locobot.base_controllers.**ILQRControl** (*bot_base*, *ctrl_pub*, *configs*)
Bases: *locobot.base_control_utils.TrajectoryTracker*

Class to implement LQR based feedback controllers on top of mobile bases.

__init__ (*bot_base*, *ctrl_pub*, *configs*)
Constructor for ILQR based Control.

Parameters

- **bot_base** – Object that has necessary variables that capture the robot state, collision checking, etc.
- **ctrl_pub** (*rospy.Publisher*) – Publisher topic to send commands to.
- **configs** – yacs configuration object.

go_to_absolute (*xyt_position*, *close_loop=True*, *smooth=True*)
Moves the robot to the robot to given goal state in the world frame using ILQR control.

Parameters

- **xyt_position** (*list or np.ndarray*) – The goal state of the form (x,y,t) in the world (map) frame.
- **close_loop** (*bool*) – When set to "True", ensures that controler is operating in open loop by taking account of odometry.
- **smooth** (*bool*) – When set to "True", ensures that the motion leading to the goal is a smooth one.

go_to_relative (*xyt_position*, *close_loop=True*, *smooth=True*)
Relative pose that the robot should go to.

track_trajectory (*states*, *controls=None*, *close_loop=True*)
State trajectory that the robot should track using ILQR control.

Parameters

- **states** (*list*) – sequence of (x,y,t) states that the robot should track.
- **controls** (*list*) – optionally specify control sequence as well.
- **close_loop** (*bool*) – whether to close loop on the computed control sequence or not.

class locobot.base_controllers.**MoveBaseControl** (*base_state*, *configs*)
Bases: object

This class encapsulates and provides interface to move_base controller used to control the base

__init__ (*base_state*, *configs*)
The constructor for MoveBaseControl class.

Parameters

- **configs** (*dict*) – configurations read from config file
- **base_state** (*BaseState*) – an object consisting of an instance of BaseState.

go_to_absolute(*xyt_position*, *close_loop=True*, *smooth=False*)
Moves the robot to the robot to given goal state in the world frame.

Parameters

- **xyt_position** (*list or np.ndarray*) – The goal state of the form (x,y,t) in the world (map) frame.
- **close_loop** (*bool*) – When set to "True", ensures that controler is operating in open loop by taking account of odometry.
- **smooth** (*bool*) – When set to "True", ensures that the motion leading to the goal is a smooth one.

class locobot.base_control_utils.**TrajectoryTracker**(*system*)
Bases: object

Class to track a given trajectory. Uses LQR to generate a controller around the system that has been linearized around the trajectory.

__init__(*system*)
Provide system that should track the trajectory.

execute_plan(*plan*, *close_loop=True*)
Executes the plan, check for conditions like bumps, etc. Plan is object returned from generate_plan. Also stores the plan execution into variable self._trajectory_tracker_execution.

Parameters

- **plan** (*LQRSolver*) – Object returned from generate_plan function.
- **close_loop** (*bool*) – Whether to use feedback controller, or apply open-loop commands.

Returns Weather plan execution was successful or not.

Return type bool

generate_plan(*xs*, *us=None*)
Generates a feedback controller that tracks the state trajectory specified by xs. Optionally specify a control trajectory us, otherwise it is automatically inferred.

Parameters

- **xs** – List of states that define the trajectory.
- **us** – List of controls that define the trajectory. If None, controls are inferred from the state trajectory.

Returns LQR controller that can track the specified trajectory.

Return type *LQRSolver*

plot_plan_execution(*file_name=None*)
Plots the execution of the plan.

Parameters **file_name** (*string*) – Name of file to plot plan execution.

stop()
Stops the base.

class locobot.base_control_utils.**MoveBasePlanner**(*configs*)

This class enscapsulates the planning capabilities of the move_base and provides planning services and plan tracking services.

get_plan_absolute(*x*, *y*, *theta*)

Gets plan as a list of poses in the world frame for the given (x, y, theta

move_to_goal(*goal*, *go_to_relative*)

Moves the robot to the robot to given goal state in the absolute frame (world frame).

Parameters

- **goal** (*list*) – The goal state of the form (x,y,t) in the world (map) frame.
- **go_to_relative** – This is a method that moves the robot the appropiate relative goal while NOT taking map into account.

parse_plan(*plan*)

Parses the plan generated by move_base service

class locobot.base_control_utils.**LQRSolver**(*As=None*, *Bs=None*, *Cs=None*, *Qs=None*, *Rs=None*, *x_refs=None*, *u_refs=None*)

Bases: object

This class implements a solver for a time-varying Linear Quadratic Regulator System. A time-varying LQR system is defined via affine time varying transition functions, and time-varying quadratic cost functions.Such a system can be solved using dynamic programming to obtain time-varying feedback control matrices that can be used to compute the control command given the state of the system at any given time step.

__init__(*As=None*, *Bs=None*, *Cs=None*, *Qs=None*, *Rs=None*, *x_refs=None*, *u_refs=None*)

Constructor for LQR solver.

System definition: $x_{t+1} = A_t x_t + B_t u_t + C_t$

State cost: $x^t_t Q_t x_t + x^t_t q + q\_$

Control cost: $u^t_t R_t u_t$

Parameters

- **As** (*List of state_dim x state_dim numpy matrices*) – List of time-varying matrices A_t for dynamics
- **Bs** (*List of state_dim x control_dim numpy matrices*) – List of time-varying matrices B_t for dynamics
- **Cs** (*List of state_dim x 1 numpy matrices*) – List of time-varying matrices C_t for dynamics
- **Qs** (*List of 3 tuples with numpy array of size state_dim x state_dim, state_dim x 1, 1 x 1*) – List of time-varying matrices Q_t for state cost
- **Rs** (*List of control_dim x control_dim numpy matrices*) – List of time-varying matrices R_t for control cost

get_control(*x*, *i*)

Uses the stored solution to the system to output control cost if the system is in state x at time step i.

Parameters

- **x** (*numpy array (state_dim,)*) – state of the system
- **i** (*int*) – time step

Returns feedback control that should be applied

Return type numpy array (control_dim,)

get_control_ls (*x*, *alpha*, *i*)
Returns control but modulated via a step-size alpha.

Parameters

- **x** (*numpy array (state_dim,)*) – state of the system
- **alpha** (*float*) – step size
- **i** (*int*) – time step

Returns feedback control that should be applied

Return type numpy array (control_dim,)

get_cost_to_go (*x*, *i*)
Returns cost to go if the system is in state x at time step i.

Parameters

- **x** (*numpy array (state_dim,)*) – state of the system
- **i** (*int*) – time step

Returns cost if system were to follow optimal f eedback control from now

Return type scalar

solve ()
Solves the LQR system and stores the solution, such that it can be accessed using get_control() function.

class locobot.base_control_utils.**ILQRSolver** (*dyn_fn*, *Q_fn*, *R_fn*, *start_state*, *goal_state*)
Bases: object

Iterative LQR solver for non-linear systems. Computes a linearization of the system at the current trajectory, and solves that linear system using LQR. This process is repeated.

__init__ (*dyn_fn*, *Q_fn*, *R_fn*, *start_state*, *goal_state*)
Constructor that sets up functions describing the system and costs.

Parameters

- **Q_fn** (*python function*) – State cost function that takes as input x_goal, x_ref, time_step, lqr_iteration, and returns quadratic approximations of state cost around x_ref.
- **R_fn** (*python function*) – Control cost function that takes as input u_ref and returns quadtratic approximation around it.
- **dyn_fn** (*python function*) – Dynamics function thattakes in state, controls, return_only_state flag, and returns the linearization and the next state.
- **start_state** (*numpy vector*) – Starting state of the system.
- **goal_state** (*numpy vector*) – Goal state of the system.

get_step_size (*lqr_*, *ref_controls*, *ref_cost*, *ilqr_iter*)
Search for the step size that improves the cost function over LQR iterations.

Parameters

- **ilqr_iter** (*int*) – Which ILQR iteration are we doing so as to compute the cost function which may depend on the ilqr_iteration for a log barrier method.
- **ref_controls** (*numpy array*) – Reference controls, we are starting iterations from.
- **ref_cost** (*float*) – Refernce cost that we want to improve over.

Returns step size, updated controls, updated cost

Return type float, numpy array, float

solve (*init_controls*, *ilqr_iters*)
Solve the non-linear system.

Parameters

- **init_controls** (*numpy array*) – Initial control sequence to start optimization from.
- **ilqr_iters** – Number of iterations of linearizations and LQR solutions.
- **ilqr_iters** – int

Returns LQR Tracker, step_size, cost of solution

Return type *LQRSolver*, int, cost

unroll (*dyn_fn*, *start_state*, *controls*)
Obtain state trajectory by applying controls on the system defined by the dynamics function dyn_fn.

Parameters

- **Q_fn** (*python function*) – State cost function that takes as input x_goal, x_ref, time_step, lqr_iteration, and returns quadratic approximations of state cost around x_ref.
- **start_state** (*numpy vector*) – Starting state of the system.
- **controls** (*numpy array*) – Sequence of controls to apply to the system.

Returns Sequence of states

Return type numpy array

2.3 Arm

class locobot.arm.**LoCoBotArm** (*configs*, *control_mode='position'*, *moveit_planner='ESTkConfigDefault'*, *use_moveit=True*)
Bases: *pyrobot.core.Arm*

This class has functionality to control a robotic arm in joint and task space (with and without any motion planning), for position/velocity/torque control, etc.

__init__ (*configs*, *control_mode='position'*, *moveit_planner='ESTkConfigDefault'*, *use_moveit=True*)
The constructor for LoCoBotArm class.

Parameters

- **configs** (*YACS CfgNode*) – configurations read from config file
- **control_mode** (*string*) – Choose between 'position', 'velocity' and 'torque' control
- **moveit_planner** (*string*) – Planner name for moveit, only used if planning_mode = 'moveit'.
- **use_moveit** (*bool*) – use moveit or not, default is True

go_home (*plan=False*)
Commands robot to home position

Parameters plan (*bool*) – use moveit to plan the path or not

set_ee_pose_pitch_roll(*position*, *pitch*, *roll=None*, *plan=True*, *wait=True*, *numerical=True*, ***kwargs*)
Commands robot arm to desired end-effector pose (w.r.t. 'ARM_BASE_FRAME'). Computes IK solution in joint space and calls set_joint_positions. Will wait for command to complete if wait is set to True.

Parameters

- **position** (*np.ndarray*) – position of the end effector (shape: [3,])
- **pitch** (*float*) – pitch angle
- **roll** (*float*) – roll angle
- **plan** (*bool*) – use moveit the plan a path to move to the desired pose
- **wait** (*bool*) – wait until the desired pose is achieved
- **numerical** (*bool*) – use numerical inverse kinematics solver or analytical inverse kinematics solver

Return result Returns True if command succeeded, False otherwise

Return type bool

set_joint_torque(*joint_name*, *value*)

Parameters

- **joint_name** (*string*) – joint name (['joint_1', 'joint_2', 'joint_3', 'joint_4'‘])
- **value** (*float*) – torque value in Nm

Returns sucessful or not

Return type bool

set_joint_torques(*torques*, ***kwargs*)
Sets the desired joint torques for all arm joints.

Parameters torques (*list*) – target joint torques, list of len 4 populated with torque to be applied on first 4 joints of arm in Nm

set_joint_velocities(*velocities*, ***kwargs*)
Sets the desired joint velocities for all arm joints

Parameters velocities (*list*) – target joint velocities

2.4 Camera

class locobot.camera.**SimpleCamera**(*configs*)
Bases: *pyrobot.core.Camera*

This is camera class that interfaces with the Realsense camera on the locobot and locobot-lite. This class does not have the pan and tilt actuation capabilities for the camera.

__init__(*configs*)
Constructor of the SimpleCamera class.

Parameters configs (*YACS CfgNode*) – Camera specific configuration object

get_current_pcd(*in_cam=True*)
Return the point cloud at current time step (one frame only)

Parameters **in_cam** (*bool*) – return points in camera frame, otherwise, return points in base frame

Returns

tuple (pts, colors)

pts: point coordinates in world frame (shape: $[N, 3]$)

colors: rgb values for pts_in_cam (shape: $[N, 3]$)

Return type tuple(np.ndarray, np.ndarray)

get_depth()

This function returns the depth image perceived by the camera.

Return type np.ndarray or None

get_intrinsics()

This function returns the camera intrinsics.

Return type np.ndarray

get_link_transform(*src*, *tgt*)

Returns the latest transformation from the target_frame to the source frame, i.e., the transform of source frame w.r.t target frame. If the returned transform is applied to data, it will transform data in the source_frame into the target_frame

For more information, please refer to http://wiki.ros.org/tf/Overview/Using%20Published%20Transforms

Parameters

- **src** (*string*) – source frame
- **tgt** (*string*) – target frame

Returns

tuple(trans, rot, T)

trans: translational vector (shape: $[3,]$)

rot: rotation matrix (shape: $[3, 3]$)

T: transofrmation matrix (shape: $[4, 4]$)

Return type tuple(np.ndarray, np.ndarray, np.ndarray)

get_rgb()

This function returns the RGB image perceived by the camera.

Return type np.ndarray or None

get_rgb_depth()

This function returns both the RGB and depth images perceived by the camera.

Return type np.ndarray or None

pix_to_3dpt(*rs*, *cs*, *in_cam=False*)

Get the 3D points of the pixels in RGB images.

Parameters

- **rs** (*list or np.ndarray*) – rows of interest in the RGB image. It can be a list or 1D numpy array which contains the row indices. The default value is None, which means all rows.

- **cs** (*list or np.ndarray*) – columns of interest in the RGB image. It can be a list or 1D numpy array which contains the column indices. The default value is None, which means all columns.
- **in_cam** (*bool*) – return points in camera frame, otherwise, return points in base frame

Returns

tuple (pts, colors)

pts: point coordinates in world frame (shape: $[N, 3]$)

colors: rgb values for pts_in_cam (shape: $[N, 3]$)

Return type tuple(np.ndarray, np.ndarray)

class locobot.camera.**LoCoBotCamera**(*configs*)

Bases: *locobot.camera.SimpleCamera*

This is camera class that interfaces with the Realsense camera and the pan and tilt joints on the robot.

__init__(*configs*)

Constructor of the LoCoBotCamera class.

Parameters configs (*YACS CfgNode*) – Object containing configurations for camera, pan joint and tilt joint.

get_pan()

Return the current pan joint angle of the robot camera.

Returns pan: Pan joint angle

Return type float

get_state()

Return the current pan and tilt joint angles of the robot camera.

Returns pan_tilt: A list the form [pan angle, tilt angle]

Return type list

get_tilt()

Return the current tilt joint angle of the robot camera.

Returns tilt: Tilt joint angle

Return type float

reset()

This function resets the pan and tilt joints by actuating them to their home configuration.

set_pan(*pan*, *wait=True*)

Sets the pan joint angle to the specified value.

Parameters

- **pan** (*float*) – value to be set for pan joint
- **wait** (*bool*) – wait until the pan angle is set to the target angle.

set_pan_tilt(*pan*, *tilt*, *wait=True*)

Sets both the pan and tilt joint angles to the specified values.

Parameters

- **pan** (*float*) – value to be set for pan joint
- **tilt** (*float*) – value to be set for the tilt joint

- **wait** (*bool*) – wait until the pan and tilt angles are set to the target angles.

set_tilt (*tilt*, *wait=True*)
Sets the tilt joint angle to the specified value.

Parameters

- **tilt** (*float*) – value to be set for the tilt joint
- **wait** (*bool*) – wait until the tilt angle is set to the target angle.

state
Return the current pan and tilt joint angles of the robot camera.

Returns pan_tilt: A list the form [pan angle, tilt angle]

Return type list

2.5 Gripper

class locobot.gripper.**LoCoBotGripper** (*configs*, *wait_time=3*)
Bases: *pyrobot.core.Gripper*

Interface for gripper

__init__ (*configs*, *wait_time=3*)
The constructor for LoCoBotGripper class.

Parameters

- **configs** (*YACS CfgNode*) – configurations for gripper
- **wait_time** (*float*) – waiting time for opening/closing gripper

close (*wait=True*)
Commands gripper to close fully

Parameters **wait** (*bool*) – True if blocking call and will return after target_joint is achieved, False otherwise

get_gripper_state ()
Return the gripper state.

Returns

state

state = -1: unknown gripper state

state = 0: gripper is fully open

state = 1: gripper is closing

state = 2: there is an object in the gripper

state = 3: gripper is fully closed

Return type int

open (*wait=True*)
Commands gripper to open fully

Parameters **wait** (*bool*) – True if blocking call and will return after target_joint is achieved, False otherwise

reset(*wait=True*)
Utility function to reset gripper if it is stuck

Parameters **wait** (*bool*) – True if blocking call and will return after target_joint is achieved, False otherwise

CHAPTER 3

PyRobot wrapper on Sawyer robot

Here are the wrapper definitions for using PyRobot with the Sawyer robot.

3.1 Arm

class sawyer.arm.**SawyerArm**(*configs*, *moveit_planner='ESTkConfigDefault'*)

Bases: *pyrobot.core.Arm*

This class has functionality to control a Sawyer manipulator.

__init__(*configs*, *moveit_planner='ESTkConfigDefault'*)

The constructor for LoCoBotArm class.

Parameters

- **configs** (*YACS CfgNode*) – configurations read from config file
- **moveit_planner** (*string*) – Planner name for moveit, only used if planning_mode = 'moveit'.

get_collision_state()

Return the collision state

Returns collision or not

Return type bool

go_home()

Commands robot to home position

move_to_neutral()

Move the robot to a pre-defined neutral pose

3.2 Gripper

class sawyer.gripper.**SawyerGripper**(*configs*, *ee_name='right_gripper'*, *calibrate=True*, *wait_time=3*)

Bases: *pyrobot.core.Gripper*

Interface for gripper.

__init__(*configs*, *ee_name='right_gripper'*, *calibrate=True*, *wait_time=3*)

Parameters

- **configs** (*YACS CfgNode*) – configurations for the robot
- **ee_name** (*str*) – robot gripper name (default: "right_gripper")
- **calibrate** (*bool*) – Attempts to calibrate the gripper when initializing class (defaults True)
- **wait_time** (*float*) – waiting time for opening/closing gripper

close(*position=None*, *wait=True*)

Commands gripper to close fully

Parameters

- **position** (*float*) – gripper position
- **wait** (*bool*) – True if blocking call and will return after target_joint is achieved, False otherwise

open(*position=None*, *wait=True*)

Commands gripper to open fully

Parameters

- **position** (*float*) – gripper position
- **wait** (*bool*) – True if blocking call and will return after target_joint is achieved, False otherwise

reset(*wait=True*)

Utility function to reset gripper if it is stuck

Parameters **wait** (*bool*) – True if blocking call and will return after target_joint is achieved, False otherwise

CHAPTER 4

search

Index

Symbols

A

B

C

E

G

I

L

M

O

P

R

S

T

U