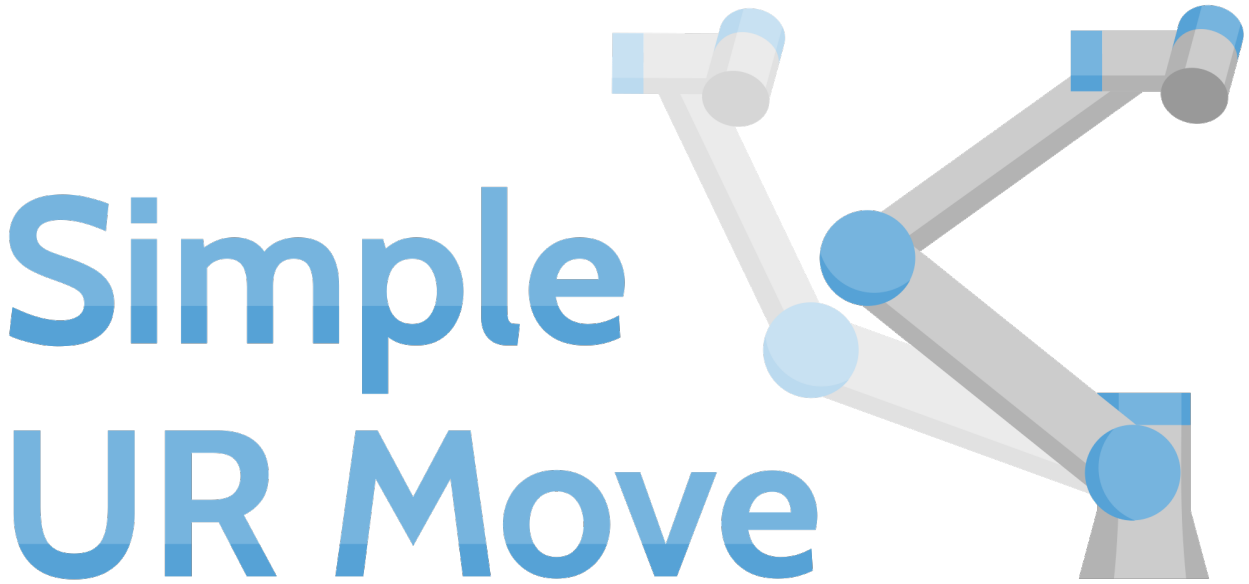
simple_ur_move

Clark B. Teeple, Harvard Microrobotics Lab

Jun 18, 2022

DOCUMENTATION



This ROS package provides a high-level python wrapper for easier control of Universal Robots using the built-in trajectory controllers in the Universal_Robots_ROS_Driver.

Table of Contents

CHAPTER

ONE

QUICKSTART GUIDE

(from " README.md " in github repo)

1.1 Installation

1. Clone the Universal Robot ROS Driver and associated dependencies
2. Clone this package to the *src* folder of your catkin workspace
3. **In the root folder of your workspace, install dependencies:**
    - `rosdep install --from-paths src --ignore-src -r -y`
4. Build your workspace (`catkin_make`)

1.2 Usage

This package has some useful python objects you can import into your own nodes to send trajectories to the robot.

- **`JointTrajectoryHandler`: Sends joint trajectories to the robot.**
    - Choose between several UR ROS controllers: `scaled_pos_joint_traj_controller`, `scaled_vel_joint_traj_controller`, `pos_joint_traj_controller`, `vel_joint_traj_controller`, and `forward_joint_traj_controller`
    - You can also go to specific joint configurations via the `go_to_point()` function.
- **`CartesianTrajectoryHandler`: Sends end effector pose trajectories to the robot.**
    - Choose between several UR ROS controllers: `pose_based_cartesian_traj_controller`, `joint_based_cartesian_traj_controller`, and `forward_cartesian_traj_controller`
    - You can also go to specific cartesian poses via the `go_to_point()` function.
- **`TwistHandler`: Sends cartesian end effector velocities to the robot.**
    - This uses the `twist_controller` from the UR ROS controllers.

Check out the *Examples*, the *full API reference*, or run any of the launch files in the `launch` folder.

CHAPTER

TWO

EXAMPLES

Some basic examples of how armstron can be used will be include here. You can find them in the armstron/launch folder in the github repo.

2.1 Start the Arm

1. **Set up the arm**
    a. *(Teach Pendant)* Turn on the robot, get into _manual_ mode, then load the "EXTERNAL_CONTROL.urp" program.
    b. *(Teach Pendant)* Load the desired installation (Load >> Installation)
    c. *(Teach Pendant)* Start the robot (tap the small red dot on the bottom left corner)
2. **Bringup the arm (ROS)**
    a. ***(Host Computer)* In a new terminal:**
        - If your arm is calibrated (for example, as in this package): `roslaunch ur_user_calibration bringup_armando.launch`
        - If your arm is not calibrated, use the standard bringup command : `roslaunch ur_robot_driver <robot_type>_bringup.launch robot_ip:=<robot_ip>`
    b. *(Teach Pendant)* Move the arm around manually to set things up.
    c. *(Teach Pendant)* Once you are ready to test, run the "EXTERNAL_CONTROL.urp" program. (press "play" in the bottom bar)

2.2 Run Joint Trajectories

Lets walk through how to run joint trajectories using a *JointTrajectoryHandler*. We will also see how the existing "run_joint_trajectory.py" script works (specify trajectory in a yaml file, then use *roslaunch* to run it on the robot.)

Contents:

- *Run Joint Trajectories (with python)*

2.2.1 Overview

`JointTrajectoryHandler` sends joint trajectories to the robot.

We can choose between several UR ROS controllers :

- `scaled_pos_joint_traj_controller` (default)
- `scaled_vel_joint_traj_controller`
- `pos_joint_traj_controller`
- `vel_joint_traj_controller`
- `forward_joint_traj_controller`

You can also go directly to specific joint configurations via the `go_to_point()` function.

2.2.2 Build a Trajectory

Configuration

We have several settings to choose for our trajectory configuration. These tell the package how to build trajectories into "ROS" format.

Config files are stored in the config folder. Config files can contain a trajectory (i.e. config and trajectory stored all in the same place), but typically we just want to set up the `settings` part, then dynamically set trajectories in Python instead.

Configuration settings:

- **units**
  - **orientation**: Units to use for joint angles. Options are either `degrees` or `radians`
- **joint_names**: Names of the joints of your arm.
- **path_tolerance**: A set of tolerances for the trajectory based on CartesianTolerance. *Typically the default will work fine.*
- **goal_tolerance**: A set of tolerances for the trajectory goal point based on CartesianTolerance. *Typically the default will work fine.*
- **goal_time_tolerance**: A tolerance (in sec) for the trajectory goal time. *Typically the default will work fine.*

Trajectory

A trajectory is just a list of waypoints in time. For joint trajectories, each waypoint has five parts:

- **time**: The time point (in sec). The trajectory implicitly starts at *0 sec* even if no point exists.
- **positions**: Joint positions at this time point (in [orientation_units]). These are required.
- **velocities**: Joint velocities at this time point (in [orientation_units]/s). If left out, zero velocity is used
- **accelerations**: Joint accelerations at this time point (in [orientation_units]/s^2). If left out, zero acceleration is used

- **effort**: Joint "efforts" (in arbitrary units). If left out, these are computed by the robot on-the-fly.

Example of a typical config file with settings and a joint trajectory:

```
settings:
  units:
    orientation: 'degrees' # degrees, radians

  joint_names: ['shoulder_pan_joint', 'shoulder_lift_joint', 'elbow_joint',
                'wrist_1_joint', 'wrist_2_joint', 'wrist_3_joint']

  # Path tolerance (set to null for default)
  path_tolerance: null
    #position_error: [0.001, 0.001 ,0.001] #[x,y,z]
    #orientation_error: [0.01, 0.01 ,0.01] #[x,y,z]
    #twist_error:
    #  linear:  [0.001, 0.001 ,0.001] #[x,y,z]
    #  angular:  [0.01, 0.01 ,0.01] #[x,y,z]
    #acceleration_error:
    #  linear:  [0.001, 0.001 ,0.001] #[x,y,z]
    #  angular:  [0.01, 0.01 ,0.01] #[x,y,z]

  # Goal tolerance (set to null for default)
  goal_tolerance: null
    #position_error: [0.001, 0.001 ,0.001] #[x,y,z]
    #orientation_error: [0.01, 0.01 ,0.01] #[x,y,z]
    #twist_error:
    #  linear:  [0.001, 0.001 ,0.001] #[x,y,z]
    #  angular:  [0.01, 0.01 ,0.01] #[x,y,z]
    #acceleration_error:
    #  linear:  [0.001, 0.001 ,0.001] #[x,y,z]
    #  angular:  [0.01, 0.01 ,0.01] #[x,y,z]

  # Time tolerance (set to null for default)
  goal_time_tolerance: null

trajectory:
  - time: 0
    positions: [-70.0, -115.0, -88.0, -65.0, 88.0, 20.0]
    velocities: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
    #accelerations: []
    #effort: []

  - time: 2.0
    positions: [-70.0, -105.0, -88.0, -65.0, 88.0, 20.0]
    velocities: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
    #accelerations: []
    #effort: []

  - time: 4.0
    positions: [-110.0, -105.0, -88.0, -65.0, 88.0, -20.0]
    velocities: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
    #accelerations: []
    #effort: []

  - time: 6.0
    positions: [-110.0, -115.0, -88.0, -65.0, 88.0, -20.0]
    velocities: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
```

(continues on next page)

(continued from previous page)

```
    #accelerations: []
    #effort: []
```

2.2.3 Run Joint Trajectories (the simple way)

This package has a simple script that loads a trajectory config file and runs it on your arm. We invoke it using run_joint_trajectory.launch, which itself runs a simple python script (run_joint_trajectory.py) with your given settings

1. *Startup the arm*
2. In a new terminal, `roslaunch simple_ur_move run_joint_trajectory.launch traj:=test_traj_joint.yaml` *(Check out these launch files for information on different settings.)*

The trajectory will be run one time. Tada!

2.2.4 Run Joint Trajectories (with python)

The big feature of this package is a very simple python interface that allows you to run trajectories directly via python (so you can focus on writing other, more-useful control software).

To run trajectories, do the following steps:

1. Import necessary packages
2. Create a trajectory handler
3. Load the configuration from a file *OR* set the configuration directly
4. Move directly to a point
5. Set up other settings
6. Run the trajectory

You can set and run trajectories on-the-fly, in a loop, etc.

```
import os
import rospkg
from simple_ur_move.joint_trajectory_handler import JointTrajectoryHandler

# Create a trajectory handler
traj_handler = JointTrajectoryHandler(
    name="",
    controller="scaled_pos_joint_traj_controller",
    debug=False)

# Load the configuration from a file
filepath_config = os.path.join(rospkg.RosPack().get_path('simple_ur_move'), 'config')
traj_file="test_traj_joint.yaml"
traj_handler.load_config(filename=traj_file, directory=filepath_config)

# OR set the configuration directly
# config = {CONFIG_DICT}
# traj_handler.set_config(config)

# Go directly to a point
point={'positions',[0,0,0,0,0,0]}    # Define the home configuration
```

(continues on next page)

(continued from previous page)

```
traj_handler.set_initialize_time(3.0) # Set the transition time here
traj_handler.go_to_point(point)       # Go to the point

# Set up other settings
traj_handler.set_trajectory(traj)
traj_handler.set_initialize_time(3.0) # Time the robot should take to get to the␣
↪starting position.
traj_handler.set_speed_factor(1.0) # Speed multiplier (smaller=slower, larger=faster)

# Run the trajectory
traj=[{'time':1.0, 'positions',[0,0,0,0,0,0]},
      {'time':3.0, 'positions',[1.57,0,0,0,0,0.3]}]

traj_handler.run_trajectory(trajectory=traj)
```

2.3 Run Cartesian Trajectories

Lets walk through how to run cartesian trajectories using a *CartesianTrajectoryHandler*. We will also see how the existing "run_cartesian_trajectory.py" script works (specify trajectory in a yaml file, then use *roslaunch* to run it on the robot.)

Contents:

2.3.1 Overview

`CartesianTrajectoryHandler` sends cartesian trajectories to the robot.

We can choose between several UR ROS controllers :

- `pose_based_cartesian_traj_controller`,
- `joint_based_cartesian_traj_controller` (default)
- `forward_cartesian_traj_controller`

You can also go directly to specific tool poses via the `go_to_point()` function.

2.3.2 Build a Trajectory

Configuration

We have several settings to choose for our trajectory configuration. These tell the package how to build trajectories into "ROS" format.

Config files are stored in the config folder. Config files can contain a trajectory (i.e. config and trajectory stored all in the same place), but typically we just want to set up the `settings` part, then dynamically set trajectories in Python instead.

Configuration settings:

- **units**
  - **position**: Units to use for positions. Options are either `m` (meter) or `mm` (millimeter)
  - **orientation**: Units to use for orientations. Options are `euler_degrees`, `euler_radians`, or `quaternion`
- **controlled_frame**: Names of the tf frame you are controlling. Usually this is `tool0`.
- **interpolate**: Whether or not to interpolate the trajectory. Options are `smooth`, `linear`, or `False` (turn off interpolation).
- **interp_time**: Interpolation time (default is 0.005 sec)
- **path_tolerance**: A set of tolerances for the trajectory based on CartesianTolerance. *Typically the default will work fine.*
- **goal_tolerance**: A set of tolerances for the trajectory goal point based on CartesianTolerance. *Typically the default will work fine.*
- **goal_time_tolerance**: A tolerance (in sec) for the trajectory goal time. *Typically the default will work fine.*

Note: A note on trajectory interpolation:

The builtin inverse kinematic solvers for the UR's seem to sometimes jump in configuration space, causing large motions for seemingly close cartesian waypoints. To prevent this from happening, this package implements a very fine-grained interpolation between waypoints (every 0.005 sec). This leads to reliable results.

Warning: Known Issue:

If waypoint orientations are defined in euler angles, occasionally the robot can flip around to alternative configurations. This is because the conversion between euler angles and quaternions happens in ROS without knowledge of the current robot configuration. If waypoint orientations are defined in quaternions directly, this problem does not exist.

Trajectory

A trajectory is just a list of waypoints in time. For cartesian trajectories, each waypoint has five parts:

- **time**: The time point (in sec). The trajectory implicitly starts at *0 sec* even if no point exists.
- **position**: Tool position (in [position_units]). This is required.
- **orientation**: Tool orientation (in [orientation_units]). This is required.
- **linear_velocity**: Tool's linear velocity (in [position_units]/s). If left out, zero velocity is used.
- **angular_velocity**: Tool's angular velocity(in [orientation_units]/s). If left out, zero velocity is used.
- **posture: Joint posture to use.** ***(This is typically omitted).***
  - **posture_joint_names**: List of the robot's joint names
  - **posture_joint_values**: List of joint orientations (in rad)

Example of a typical config file with settings and a cartesian trajectory:

```
settings:
  units:
    position: 'm' # m (meter), mm (millimeter)
    orientation: 'euler_degrees' # euler_degrees, euler_radians, quaternion

  controlled_frame: 'tool0'
  interpolate: 'smooth' # 'smooth' or 'linear'
  interp_time: 0.005

  # Path tolerance (set to null for default)
  path_tolerance: null
    #position_error: [0.001, 0.001 ,0.001] #[x,y,z]
    #orientation_error: [0.01, 0.01 ,0.01] #[x,y,z]
    #twist_error:
    #  linear:  [0.001, 0.001 ,0.001] #[x,y,z]
    #  angular:  [0.01, 0.01 ,0.01] #[x,y,z]
    #acceleration_error:
    #  linear:  [0.001, 0.001 ,0.001] #[x,y,z]
    #  angular:  [0.01, 0.01 ,0.01] #[x,y,z]

  # Goal tolerance (set to null for default)
  goal_tolerance: null
    #position_error: [0.001, 0.001 ,0.001] #[x,y,z]
    #orientation_error: [0.01, 0.01 ,0.01] #[x,y,z]
    #twist_error:
    #  linear:  [0.001, 0.001 ,0.001] #[x,y,z]
    #  angular:  [0.01, 0.01 ,0.01] #[x,y,z]
    #acceleration_error:
    #  linear:  [0.001, 0.001 ,0.001] #[x,y,z]
    #  angular:  [0.01, 0.01 ,0.01] #[x,y,z]

  # Time tolerance (set to null for default)
  goal_time_tolerance: null

trajectory:
  - time: 0
    position: [-0.100, -0.600, 0.100]
    orientation: [180, 0, 180]
    #linear_velocity:
    #angular_velocity:
  - time: 2.0
    position: [-0.100, -0.600, 0.200]
    orientation: [180, 0, 90]
    #linear_velocity:
    #angular_velocity:
  - time: 3.0
    position: [-0.100, -0.600, 0.100]
    orientation: [180, 0, 90]
    #linear_velocity:
    #angular_velocity:
```

2.3.3 Run Cartesian Trajectories (the simple way)

This package has a simple script that loads a trajectory config file and runs it on your arm. We invoke it using run_cartesian_trajectory.launch, which itself runs a simple python script (run_cartesian_trajectory.py) with your given

settings

1. *Startup the arm*
2. In a new terminal, `roslaunch simple_ur_move run_cartesian_trajectory.launch traj:=test_traj.yaml` *(Check out these launch files for information on different settings.)*

The trajectory will be run one time. Tada!

2.3.4 Run Cartesian Trajectories (with python)

The big feature of this package is a very simple python interface that allows you to run trajectories directly via python (so you can focus on writing other, more-useful control software).

To run trajectories, do the following steps:

1. Import necessary packages
2. Create a trajectory handler
3. Load the configuration from a file *OR* set the configuration directly
4. Move directly to a point
5. Set up other settings
6. Run the trajectory

You can set and run trajectories on-the-fly, in a loop, etc.

```
import os
import rospkg
from simple_ur_move.cartesian_trajectory_handler import CartesianTrajectoryHandler

# Create a trajectory handler
traj_handler = CartesianTrajectoryHandler(
    name="",
    controller="pose_based_cartesian_traj_controller",
    debug=False)

# Load the configuration from a file
filepath_config = os.path.join(rospkg.RosPack().get_path('simple_ur_move'), 'config')
traj_file="test_traj.yaml"
traj_handler.load_config(filename=traj_file, directory=filepath_config)

# OR set the configuration directly
# config = {CONFIG_DICT}
# traj_handler.set_config(config)

# Go directly to a point
point={'position',[0.5, 0.5, 0.2], 'orientation',[180, 0, 90]}    # Define the home␣
↪configuration
traj_handler.set_initialize_time(3.0) # Set the transition time here
traj_handler.go_to_point(point)       # Go to the point

# Set up other settings
traj_handler.set_trajectory(traj)
traj_handler.set_initialize_time(3.0) # Time the robot should take to get to the␣
↪starting position.
traj_handler.set_speed_factor(1.0) # Speed multiplier (smaller=slower, larger=faster)
```

(continues on next page)

(continued from previous page)

```
# Run the trajectory
traj=[{'time':1.0, 'position',[-0.5, -0.5, 0.2], 'orientation',[180, 0, 90]},
      {'time':3.0, 'position',[-0.3, -0.3, 0.15], 'orientation',[180, 0, 90]}]

traj_handler.run_trajectory(trajectory=traj)
```

CHAPTER

THREE

API REFERENCE

Each page contains details and full API reference for all the classes that can be accessed outside the simple_ur_move package. These are located in the `src/simple_ur_move` folder.

For an explanation of how to use all of it together, see *Quickstart Guide*.

3.1 Joint Trajectory Handler

The joint trajectory interface, allowing you to easily send joint trajectories (handles all the extra metadata for you).

class joint_trajectory_handler.**JointTrajectoryHandler**(*name*, *controller=None*, *debug=False*)

The joint trajectory interface

Parameters

- **name** (*str*) – Name of the Action Server
- **controller** (*str*) – The joint controller to use. Options are: `scaled_pos_joint_traj_controller`, `scaled_vel_joint_traj_controller`, `pos_joint_traj_controller`, `vel_joint_traj_controller`, and `forward_joint_traj_controller`.
- **debug** (*bool*) – Turn on debug print statements

build_goal(*trajectory*)

Build the trajectory goal

Parameters **trajectory** (*dict or trajectory_msgs/JointTrajectory*) – Trajectory to parse

Returns **goal** – The goal built from the trajectory

Return type trajectory_msgs/FollowJointTrajectoryGoal

convert_units(*trajectory*, *direction='to_ros'*)

Convert units from the units defined in the trajectory config to ROS default units (or vice versa).

Parameters

- **trajectory** (*list*) – Trajectory to parse
- **direction** (*str*) – Which direction to convert. Options are `to_ros` or `from_ros`.

Returns **trajectory** – Converted trajectory

Return type list

go_to_point(*point*)
Move the arm from its current pose to a new pose.

Parameters **point** (*dict or trajectory_msgs/JointTrajectoryPoint*) – Trajectory point to go to

load_config(*filename*, *directory=None*)
Load a trajectory configuration from a file

Parameters

- **filename** (*str*) – Filename of configuration to load
- **directory** (*str*) – Directory where config files should be loaded from. Default is the config folder of this package

pack_trajectory(*trajectory*)
Pack a trajectory into a ROS `JointTrajectory`

Parameters **trajectory** (*list*) – Trajectory to parse

Returns **ros_traj** – A ROS trajectory

Return type trajectory_msgs/JointTrajectory

run_trajectory(*trajectory=None*, *blocking=True*, *perform_init=True*)
Run a trajectory.

Parameters

- **trajectory** (*dict, JointTrajectory, FollowJointTrajectoryGoal*) – Trajectory to run. If excluded, the currently loaded trajectory is run
- **blocking** (*bool*) – Whether to wait for the trajectory to finish.
- **perform_init** (*bool*) – Whether to move to the initial point (using `initialize_time`) or skip it.

set_config(*config*)
Set the trajectory configuration

Parameters **config** (*dict*) – Configuration to set

set_initialize_time(*time*)
Set the time the robot takes to get to its initial poisiton

Parameters **time** (*float*) – Initialization time in seconds. Must be greater than or equal to 0

set_speed_factor(*speed_factor*)
Set the speed multiplier

Parameters **speed_factor** (*float*) – Speed multiplier to use

set_trajectory(*trajectory*)
Set the current trajectory

Parameters **trajectory** (*list or trajectory_msgs/JointTrajectory*) – Trajectory to parse

Raises **ValueError** – If a trajectory of incorrect type is passed

shutdown()
Shut down gracefully

3.2 Cartesian Trajectory Handler

The cartesian trajectory interface, allowing you to easily send cartesian trajectories (handles all the extra metadata for you).

class cartesian_trajectory_handler.**CartesianTrajectoryHandler**(*name*, *controller=None*, *debug=False*)

The cartesian trajectory interface

Parameters

- **name** (*str*) – Name of the Action Server
- **controller** (*str*) – The cartesian controller to use. Options are: `forward_cartesian_traj_controller`, `pose_based_cartesian_traj_controller`, and `joint_based_cartesian_traj_controller`.
- **debug** (*bool*) – Turn on debug print statements

build_goal(*trajectory*)

Build the trajectory goal

Parameters **trajectory** (*dict or cartesian_control_msgs/CartesianTrajectory*) – Trajectory to parse

Returns **goal** – The goal built from the trajectory

Return type cartesian_control_msgs/FollowCartesianTrajectoryGoal

convert_units(*trajectory*, *direction='to_ros'*)

Convert units from the units defined in the trajectory config to ROS default units (or vice versa).

Parameters

- **trajectory** (*list*) – Trajectory to parse
- **direction** (*str*) – Which direction to convert. Options are `to_ros` or `from_ros`.

Returns **trajectory** – Converted trajectory

Return type list

go_to_point(*point*)

Move the arm from its current pose to a new pose.

Parameters **point** (*dict or cartesian_control_msgs/CartesianTrajectoryPoint*) – Trajectory point to go to

load_config(*filename*, *directory=None*)

Load a trajectory configuration from a file

Parameters

- **filename** (*str*) – Filename of configuration to load
- **directory** (*str*) – Directory where config files should be loaded from. Default is the config folder of this package

pack_trajectory(*trajectory*)

Pack a trajectory into a ROS `CartesianTrajectory`

Parameters **trajectory** (*list*) – Trajectory to parse

Returns **ros_traj** – A ROS trajectory

Return type cartesian_control_msgs/CartesianTrajectory

run_trajectory(*trajectory=None*, *blocking=True*, *perform_init=True*)
Run a trajectory.

Parameters

- **trajectory** (*dict, CartesianTrajectory, FollowCartesianTrajectoryGoal*) – Trajectory to run. If excluded, the currently loaded trajectory is run
- **blocking** (*bool*) – Whether to wait for the trajectory to finish.
- **perform_init** (*bool*) – Whether to move to the initial point (using `initialize_time`) or skip it.

set_config(*config*)
Set the trajectory configuration

Parameters **config** (*dict*) – Configuration to set

set_initialize_time(*time*)
Set the time the robot takes to get to its initial poisiton

Parameters **time** (*float*) – Initialization time in seconds. Must be greater than or equal to 0

set_speed_factor(*speed_factor*)
Set the speed multiplier

Parameters **speed_factor** (*float*) – Speed multiplier to use

set_trajectory(*trajectory*)
Set the current trajectory

Parameters **trajectory** (*list or cartesian_control_msgs/CartesianTrajectory*) – Trajectory to parse

Raises **ValueError** – If a trajectory of incorrect type is passed

shutdown()
Shut down gracefully

3.3 Twist Handler

The twist interface, allowing you to easily specify cartesian speeds.

class twist_handler.**TwistHandler**(*name*, *controller='twist_controller'*, *debug=False*, *self_contained=False*)
The twist interface

Parameters

- **name** (*str*) – Name of the Action Server
- **controller** (*str*) – The twist controller to use. Options are: `twist_controller`
- **debug** (*bool*) – Turn on debug print statements
- **self_contained** (*bool*) – Decide whether to set jog speed back to zero when object is deleted.

build_twist(*linear*, *angular*)
Build a twist message from vectors

Parameters

- **linear** (*list*) – The linear twist components [x,y,z]
- **angular** (*list*) – The angular twist components [x,y,z]

Returns **twist** – The resulting twist message

Return type geometry_msgs/Twist

convert_units(*twist*, *direction='to_ros'*)
Convert units from the units defined in the trajectory config to ROS default units (or vice versa).

Parameters

- **twist** (*dict*) – twist to parse
- **direction** (*str*) – Which direction to convert. Options are `to_ros` or `from_ros`.

Returns **twist** – Converted twist

Return type dict

load_config(*filename*, *directory=None*)
Load a configuration from a file

Parameters

- **filename** (*str*) – Filename of configuration to load
- **directory** (*str*) – Directory where config files should be loaded from. Default is the config folder of this package

set_config(*config*)
Set the configuration

Parameters **config** (*dict*) – Configuration to set

set_speed_factor(*speed_factor*)
Set the speed multiplier

Parameters **speed_factor** (*float*) – Speed multiplier to use

set_twist(*twist*)
Run a trajectory.

Parameters **twist** (*dict or geometrty_msgs/Twist*) – Twist to set.

shutdown()
Shut down gracefully

3.4 Controller Handler

This handler allows you to easily turn on ROS controllers, and ensure other controllers get turned off when necessary.

class controller_handler.**ControllerHandler**(*robot_name*, *debug=False*)
Handle ROS controllers

Parameters **robot_name** (*str*) – Name of the robot

get_controller_list()
Get the list of currently loaded controllers

Returns **controllers** – List of controllers

Return type list

get_controllers_with_state(*states=None*)
Get a list of controllers that have a particular state

Parameters **states** (*list or str*) – List of states (or s single state) to check for (`uninitialized`, `initialized`, `running`, `stopped`, `waiting`, `aborted`, `unknown`)

Returns **controllers** – List of controller names matching the states

Return type list

load_controller(*controller*)
Load a ROS controller

Parameters **controller** (*str*) – Name of the controller to load

Returns **response** – Service response from the controller manager

Return type str

play_program()
Start the program on the teach pendant. This ony works if you are in remote control mode

Returns **result** – The result of the service call

Return type srv

set_controller(*controller*)
Set which ROS controller is started, and stop all others

Parameters **controller** (*str*) – Name of the controller to start

Returns **response** – Service response from the controller manager

Return type str

set_reserved_controllers(*controllers*)
Set which controllers are reserved (always running)

Parameters **controllers** (*list*) – List of controller names

set_speed_slider(*fraction*)
Set the speed slider fraction

Parameters **fraction** (*float*) – Slider fraction to set (0.02 to 1.00)

Returns **result** – The result of the service call

Return type srv

stop_program()
Stop the program on the teach pendant. This ony works if you are in remote control mode

Returns **result** – The result of the service call

Return type srv

switch_controller(*start_controllers*, *stop_controllers*, *strictness=1*, *start_asap=False*, *timeout=0*)
Switch ROS controllers

Parameters

- **start_controllers** (*list*) – Names of the controllers to start

- **stop_controllers** (*list*) – Names of the controllers to stop
- **strictness** (*int*) – Strictness of controller switching
- **start_asap** (*bool*) – Decide whether controllers should be started immediately
- **timeout** (*int*) – Timeout (in seconds)

Returns **response** – Service response from the controller manager

Return type str

unload_controller(*controller*)
Unload a ROS controller

Parameters **controller** (*str*) – Name of the controller to unload

Returns **response** – Service response from the controller manager

Return type str

update_controller_list()
Update the controller list via a ROS service call.

CHAPTER

FOUR

CONTRIBUTING

Contributing Checklist

- New code can only be added via a new branch and reviewed PR (no pushes to main!)
- Always bump the version of your branch by increasing the version number listed in _version.txt

CHAPTER

FIVE

INDEX

CHAPTER

SIX

INSTALL

Follow instructions in the *Quickstart Guide*.

CHAPTER

SEVEN

EXPLORE THE EXAMPLES

Check out the *Examples*, or run any of the launch files in the `launch` folder.

CHAPTER

EIGHT

LINKS

- **Documentation:** Read the Docs
- **Source code:** Github

CHAPTER

NINE

CONTACT

If you have questions, or if you've done something interesting with this package, get in touch with Clark Teeple, or the Harvard Microrobotics Lab!

If you find a problem or want something added to the library, open an issue on Github.

PYTHON MODULE INDEX

INDEX

T

U