clopema_compliance Documentation

Release 0.0.1

Tadeas Lejsek

December 29, 2014

CONTENTS

CHAPTER

ONE

MANAGERS

1.1 Move Manager

The class `MoveManager` from the module `move_manager` provides the functionality connected with operating the *CloPeMa* robot.

1.1.1 Robot movement

There are two methods for moving the robot arms along a straight line: `Move()` for one arm only and `MoveBoth()` for both arms at the same time. Both methods can be executed asynchronously. Use `StopExecution()` to stop the execution of an asynchronous movement.

There are also two example methods for turning the arm in a specified joint: `TurnR2R()` for joint *r2_joint_r* and `TurnL2T()` for joint *r1_joint_t*. The speed of the movement can be specified directly.

There are methods for changing the speed of the whole robot: `SetRobotSpeed()` and `GetRobotSpeed()`. Note that the maximum is 0.2 (20% of the speed that can be achieved through the touch pendant).

The methods `Home()`, `RightHome()` and `LeftHome()` move both or one arm to its home position.

In case one does not need any advanced functionality concerning the grippers, one can use `OpenGripper()` and `CloseGripper()` to fully open or close the gripper.

1.1.2 Manipulation with the poses

The method `CreatePose()` is used to create a pose (*PoseStamped*). The method `RotatePose()` rotates the pose around a specified axis and angle. The method `TransformPoint()` transforms a point (*Point*) from one coordinate system to another.

The method `GetCurrentPose()` returns the pose of the specified coordinate system.

The method `GetXYDistanceBetweenFrames()` returns the distance between two coordinate systems from the top view.

1.2 Gripper Manager

The class `GripperManager` from the module `gripper_manager` provides an advanced functionality for the *CloPeMa* grippers.

The user can get/set the current opening/closing speed of the gripper through `GetGripperFrequency()` and `SetGripperFrequency()`. The maximum is 25000.

The gripper can be opened only partially using `MoveAbsolutePercentage()` on the percentage like basis. 0 is fully open and 100 fully close.

1.3 Force Manager

The class `ForceManager` from the module `force_manager` provides the functionality connected with the two force/torque sensors that are placed in each wrist of the *CloPeMa* robot.

The method `GetForce()` reads the force data from the force sensor.

The methods `CloseOnForce()` and `OpenOnForce()` are used to simplify the work of the user when inserting objects to the gripper. The gripper opens/closes when the user exerts a force on it. I use the `CloseOnForce()` every time I have to manually place something into the gripper.

The methods `WaitUntilForceX()` and `WaitUntilForceZ()` interrupt the execution of the program until the force exceeds a certain threshold. This is useful in situations when for instance the movement of the robotic arm should be stopped. These methods provide the main functionality for the **slingshot** (`shooter2.Shooter` class). They are also used during the **knot-tying** (`tyer2.Tyer` class) to stop the motion of the robotic arms when the knot is already tightened.

1.4 Camera Manager

The class `CameraManager` from the module `camera_manager` works with the *PointCloud2* message obtained from the *xtion1* or *xtion2* sensors.

One can get the raw PointCloud message using `GetPointCloud()`. The RGB image is obtained using `GetRgbImage()` and the depth image `GetDepthImage()`. If one wishes to work with both contained in one array, one can use `GetXYZ()`. This method returns a m x n x 6 numpy array. 1 to 3 are the x, y and z point coordinates and 4 to 6 the R, G and B.

The algorithm for finding connected components is implemented as the method `find_component2()`. The method `find_rope_end()` segments the rope and finds the rope end. This is crucial part in **knot-tying** (`tyer2.Tyer` class).

1.5 Proximity Manager

The class `ProximityManager` from the module `proximity_manager` provides the functionality connected with the tactile, light and proximity sensors that are placed in each gripper.

The method `GetProximity()` reads the current data from all the tactile, light and proximity sensors from both grippers. The data is returned in a big array.

The method `CloseOnProximity()` closes the gripper when the output of the light sensor in that gripper exceeds a certain threshold. This method is used when catching the swinging pole and ribbon in the robotic **gymnastics** (`pendulum2.Pendulum` class).

The method `WaitUntilProximityPeak()` interrupts the execution of the program until the maximum of a peak in the output of the proximity sensor has gone. This method is the core stone of stopping the movement of the robotic arm in the **regrasping** (`regrasper.Regrasper` class).

CHAPTER

TWO

SOURCE CODE DOCUMENTATION

2.1 Use Cases Reference

2.1.1 Slingshot

class `shooter2.`**`Shooter`**

Shoots a projectile with a slingshot.

Attributes: alpha (float): Shooting angle.

thresh_load (float): Force threshold for detecting that the elastic string was touched.

thresh_fire0 (float): Force threshold used to detect that the elastic string starts to be stretched.

thresh_fire1 (float): If this force threshold is exceeded, the slingshot is ready to fire (the elastic string stops to be stretched).

slow_speed (float): How many times the speed of the robot should be slowed down during loading and firing.

`AdjustAlpha`(*alpha*)

Adjusts the shooting angle *alpha*.

Args: alpha (float): shooting angle in degrees <0, 40>. *alpha* = 0 means horizontal shooting. *alpha* > 0 means shooting upwards.

`Fire`()

Starts stretching the elastic string.

The position when a certain force (given by *thresh_fire0*) is exerted on the string is noted. When the measured force exceeds *thresh_fire1*, the movement is stopped. The *dx* is computed.

`InitPosition`()

Go to initial position.

`InsertBullet`()

Place the projectile to the opened gripper. When you exert a force on the gripper, it closes automatically.

`Load`()

Loads the projectile.

The projectile is moved towards the elastic string. When the string is touched, the movement of the robotic arm is stopped.

`ShootMultiple`()

Shoots a projectile multiple times.

After every shot the user can decide, whether he or she wants to fire a next one. Type *y* for the next shot, *n* to stop.

ShootOne()
Shoots a projectile once.

ShootingPosition()
Go to the shooting position.

2.1.2 Knot-tying

class tyer2.**Tyer**
Ties an overhand knot on a given 2m rope.

Attributes: width (int): A width of a gap that can be stepped over during finding connected components [pixels].

show (int): 0 for no images, 1 to show rope end, 2 to show rope end and all images from the segmentation process.

CatchX (float), CatchY (float), CatchZ (float): x, y and z offset for rope end catching [m].

catch(*catchX=None*, *catchY=None*, *catchZ=None*)
Catches the rope end and informs the uses whether the rope end was actually caught.

get_image()
Turn 'r2_arm', move the 'r1_arm' so that the rope end gets in sight of xtion1 and finds the rope end.

init()
Move to initial position: both grippers facing each other.

insert_rope()
Insert the rope to both grippers.

tie_a_knot()
The whole knot-tying procedure.

tighten()
Moves both arms to the tightening pose, helps the rope slide down the 'r2_arm' and tightens the knot by stretching both arms.

wrap()
Wraps the rope around the 'r2_arm'.

2.1.3 Gymnastics

class pendulum2.**Pendulum**(*catch_angle*, *y1_diff*)
Pendulum - ribbon catching.

Args:

y1_diff (float): Adjust the position of the 'r1_arm' gripper in x-z plane [m].

catch(*T*)
Swing the pole hanging on a ribbon and catch it.

Args: T (float): Swing period [s].

dance(*num*, *T*)
Swings the robot arm in joint *r1_joint_t* there and back.

Args: T (float): Period of the swings [s]. num (int): Number of swings.

dancing_pose()

Moves the 'r1_arm' to the dancing pose.

sequence()

Executes the whole sequence: swings the pole, catches it with the other arm and performs a few fast movements with it.

swing(*T*, *num*, *async=False*)

Swings the arm in *r2_joint_r* between trajectory points *rr_0* and *rr_2*.

Args: T (float): Swing period [s]. num (int): Number of swings. async (boolean, optional): *True* for asynchronous execution of the motion.

2.1.4 Regrasping

class `regrasper.`**Regrasper**(*h=0.3*, *dy=0.03*, *peak_thresh=20*)

Regrasps a hanging piece of rope or string.

Args: h (float, optional): distance between the two grippers [m].

dy (float, optional): adjusts the position of the gripper used for catching [m].

peak_thresh (int, optional): When this threshold is exceeded, a peak in the output of the sensor starts to be recorded.

grasp_left()

Moves the rope towards the 'r1_arm', stops the motion of 'r2_arm' upon the detection of the presence of the rope and finally closes the gripper of 'r1_arm'.

grasp_right()

Moves the rope towards the 'r1_arm', stops the motion of 'r2_arm' upon the detection of the presence of the rope and finally closes the gripper of 'r1_arm'.

init()

Go to initial position and open both grippers

regrasp()

Regrasp the rope from 'r1_arm' to 'r2_arm' and back.

switch_arms_left()

Switches the poses of both arms.

switch_arms_right()

Switches the poses of both arms.

2.2 Manager Reference

2.2.1 Move Manager

class `move_manager.`**MoveManager**(*init*, *frame='base_link'*, *eef_l='r1_ee'*, *eef_r='r2_ee'*)

Move manager is used to control the movements of the arms of the *CloPeMa* robot.

Args: init (int): If set to 1, a ros node 'move_manager' is initialized.

frame (string, optional): Sets the default world coordinate system. Used as a ClopemaRobotCommander reference frame.

eef_l (string, optional): End effector link for the left arm ('r1_arm').

eef_r (string, optional): End effector link for the right arm ('r2_arm').

Attributes: crc (ClopemaRobotCommander): ClopemaRobotCommander for both arms ('arms').

`CloseGripper`(*armName*)

Fully closes a gripper.

Args: armName (string): 'left' for r1_arm or 'right' for r2_arm

static `CreatePose`(*x*, *y*, *z*, *rx*, *ry*, *rz*)

Creates a pose in the 'base_link' coordinate system.

Args: x (float): x-coordinate [m]

y (float): y-coordinate [m]

z (float): z-coordinate [m]

rx (float): rotation around x-axis [degrees]

ry (float): rotation around y-axis [degrees]

rz (float): rotation around z-axis [degrees]

Returns: ps (PoseStamped)

`GetCurrentPose`(*link_name*)

Get the current pose of a specified link *link_name*.

Arsg: link_name (str): Name of the robot link e.g. 'r1_ee'.

Returns: (PoseStamped): Pose of the requested link

`GetRobotSpeed`()

Get the current speed of the robot.

Returns: speed (float): Current robot speed. <0, 0.2>

static `GetXYDistanceBetweenFrames`(*frame1='xtion1_rgb_optical_frame'*, *frame2='r2_ee'*)

Compute the distance in x-y plane (top view) between two frames.

Args: frame1 (str): First frame. frame2 (str): Second frame.

Returns: dist (float): distance in x-y plane between *frame1* and *frame2*

`Home`()

Move both arms to their home positions.

static `LeftHome`()

Move the 'r1_arm' to its home position.

`Move`(*ps*, *armName*, *params={'async': False*, *'step': 0.01*, *'jump_thresh': 1.2}*)

Cartesian move of one arm.

Args: ps (PoseStamped): Target pose.

armName (string): 'left' for r1_arm or 'right' for r2_arm

params (optional): step (float): Distance between the generated trajectory points.

jump_thresh: Max allowed jump.

async (Boolean): If set to *True*, the trajectory is executed asynchronously.

`MoveBoth`(*psl*, *psr*, *params={'async': False*, *'step': 0.01*, *'jump_thresh': 1.2}*)

Cartesian move of both arms.

Args: psl (PoseStamped): Target pose of the left arm.

psr (PoseStamped): Target pose of the right arm.

params (optional): step (float): Distance between the generated trajectory points.

jump_thresh: Max allowed jump.

async (Boolean): If set to *True*, the trajectory is executed asynchronously.

`OpenGripper`(*armName*)
Fully opens a gripper.

Args: armName (string): 'left' for r1_arm or 'right' for r2_arm

static **`RightHome`**()
Move the 'r2_arm' to its home position.

static **`RotatePose`**(*poseStamped*, *angle*, *axis*, *point*)
Rotates a given pose *poseStamped* around axis *axis* that goes through a point *point* around the angle *angle*.

Args: poseStamped (PoseStamped): Initial pose.

angle (float): Angle of rotation [rad].

axis (numpy float array 1x3): Axis of rotation.

point (numpy float array 1x4): Point lying on axis in homogenous coordinates.

Returns: ps (PoseStamped): Rotated pose.

`SetRobotSpeed`(*speed*)
Sets robot speed.

Args: speed (float): Robot speed. <0, 0.2> 0.2 is the maximum when operating the robot through ROS.

`StopExecution`()
Stops the execution of an asynchronous movement of the robotic arms.

static **`TransformPoint`**(*point*, *frameFrom*, *frameTo*)
Transforms a point *point* from one coordinate system to another.

Args: point (Point): Point to be transformed.

frameFrom (str): Name of the current frame.

frameTo (str): Name of the target frame.

Returns: (point): Point in the new coordinate system.

`TurnL2T`(*angle*, *time=1.0*)
Turn r1_joint_t to specified angle (degree)

`TurnR2R`(*angle*, *time=1.0*)
Turn r2_joint_r to specified angle (degree)

2.2.2 Force Manager

class `force_manager.`**`ForceManager`**(*moveManager*)
Force Manager provides the functionality connected with the force/torque sensor.

Args: moveManager (MoveManager): Move manager.

Attributes: force_left (str): Name of the topic connected with the force/torque sensor placed in 'r1_arm'.

force_right (str): Name of the topic connected with the force/torque sensor placed in 'r2_arm'.

CloseOnForce (*armName*)

Opens the gripper and closes it when the force exerted on it exceeds the *CLOSING_THRESH* threshold [Newton].

Args: armName (string): ‘left’ for r1_arm or ‘right’ for r2_arm.

GetForce (*armName*)

Gets the force output from the force/torque sensor.

Args: armName (string): ‘left’ for r1_arm or ‘right’ for r2_arm.

Returns: (point): x, y and z coordinate with force [N].

OpenOnForce (*armName*)

Opens the gripper when the force exerted on it exceeds the *CLOSING_THRESH* threshold [Newton].

Args: armName (string): ‘left’ for r1_arm or ‘right’ for r2_arm.

WaitUntilForceX (*armName*, *thresh*)

Waits until the x-coordinate of the force exerted on the robot arm exceed the threshold *thresh*.

Args: armName (string): ‘left’ for r1_arm or ‘right’ for r2_arm.

thresh (float): Force treshold [N].

WaitUntilForceZ (*armName*, *thresh*, *max_it=1000*)

Waits until the z-coordinate of the force exerted on the robot arm exceed the threshold *thresh*.

Args: armName (string): ‘left’ for r1_arm or ‘right’ for r2_arm.

thresh (float): Force treshold [N].

max_it (int, optional): Maximum amount of iterations. After this number is exceeded, the waiting stops.

2.2.3 Camera Manager

class camera_manager.**CameraManager** (*init=1*)

Camera manager provides a functionality connected with the xtion sensor.

static GetDepthImage (*numpyPointCloud*)

Gets a depth image from the PointCloud.

static GetPointCloud (*topicName='/xtion1/depth_registered/points'*)

Get a PointCloud message from the xtion sensor.

Args: topicName (str): Name of the topic for the corresponding xtion sensor (*xtion1* or *xtion2*).

Returns: numpyPointCloud (numpy array): Point cloud. header (str): Name of the camera coordinate system.

static GetRgbImage (*numpyPointCloud*)

Gets a RGB image from the PointCloud.

static GetXYZ (*numpyPointCloud*)

Transforms the PointCloud obtained from the xtion sensor into numpy array.

Returns: xyz (numpy array): m x n x 6 (x, y and z-coordinate, r, g, b)

static find_component2 (*mask*, *width*)

Algorithm that finds connected component in the given image.

Args: mask (numpy array): 1 is foreground, 0 is background. width (int): Algorithm will step over a gap < *width* [pixels].

Returns: mask_ref (numpy array): 1 is foreground of the newly found connected components, the rest is 0 (background).

static **find_rope_end**(*numpyPointCloud*, *rope_dist*, *width*, *show=0*)
Finds the coordinates of the rope end.

Args: numpyPointCloud (numpy array): Point cloud.

rope_dist (float): Estimated distance between the rope and the camera coordinate system.

width (int): Parameter of the algorithm used to find connected components.

show (int, optional): 0 for no images, 1 to show rope end image, 2 to show all images (rope end, segmentation).

Returns: Xc (point): Rope end in the camera coordinate system.

2.2.4 Gripper Manager

class gripper_manager.**GripperManager**(*armName*)
Gripper Manager

Provides a fine control of the grippers. One instance per gripper is needed.

Args: armName(str): 'left' or 'right'

static **GetGripperFrequency**(*armName*)
Gets the gripper opening/closing speed.

Args: armName (string): 'left' for r1_arm or 'right' for r2_arm

Returns: Gripper frequency (int): <0, 25000>, 0 is min, 25000 is max.

MoveAbsolutePercentage(*perc*)
Opens/Closes the gripper to a specified degree

Args: perc: From 0 (full open) to 100 (full close)

static **SetGripperFrequency**(*armName*, *freq*)
Sets the gripper opening/closing speed.

Args: armName (string): 'left' for r1_arm or 'right' for r2_arm

freq (int): Frequency of the gripper. <0, 25000>; 0 is minimum, 25000 is maximum.

2.2.5 Proximity Manager

class proximity_manager.**ProximityManager**(*moveManager*)
Proximity manager is used to provide functionality connected with the light and proximity sensors that are placed in the grippers.

CloseOnProximity(*arm_name*, *thresh=70*)
The gripper closes when the output of the light sensor in exceeds the threshold *thresh*.

Args: arm_name (str): 'left' for r1_arm or 'right' for r2_arm.

thresh (float): Threshold for the output of the light sensor.

GetProximity()
Get the data from the tactile, light and proximity sensors from both grippers.

Returns: sensor_responses (array)

`WaitUntilProximityPeak`(*thresh*, *ind*, *max_it*)

Measures the output of the particular sensor in the gripper and waits until a peak passes.

Args: thresh (int): If the sensor output exceeds *thresh*, the beginning of the peak was detected.

ind (int): Index to the *sensor_responses* array. 35 for 'r1_arm' proximity, 34 for 'r2_arm' proximity.

max_it (int): Maximum amount of iterations. After this number is exceeded, the waiting stops.

CHAPTER

THREE

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

PYTHON MODULE INDEX

INDEX