MoveIt! Documentation

By Aike van Alkemade

Introduction

MoveIt! is a ROS extension that makes motion planning munch more convenient. It can do forward and inverse kinematics with build in solvers. The package is able to do obstacle and self-collision avoidance and trajectory execution. To configure MoveIt! for a new robot a setup assistant is used. When a working URDF/XARCO model is supplied, the setup assistant creates a structured way of setting up the software.

Setup assistant

To start the setup the next command is used:

```
roslaunch moveit_setup_assistant setup_assistant.launch
```

The first step is adding the a URDF/XACRO file. This filed was created by Levi and Salah and it is file explaining the robot in computer language. An extra joint for a rotating end-efector and joint limits were later added, to make it more comparable to the physical prototype. During this time the final design was not finished and a rough model was used.

Generate Self-Collision Matrix

Because our arm isn’t able to self collide this step can be skipped.

Add Virtual Joints

In the URDF file a connection to the world was already made, so this option could be skipped aswell.

Add Planning Groups

Motion planning in MoveIt! is done by grouping joints. Afterwards you can select one of the supplied kinematic solvers or make your own custom solver for the robot. There are also options for setting search resolution, search timeout and solver attempts. With the software you can also include your own OMPL based path planner. At this time this parameter is left on default.

3D Perception

This option is used for creating parameters for configuring 3D sensors. For the first simulation this option was generated, but still not implemented because of a lack of prototype.

ROS Control

This tab is used for generating ROS control for the robot. It auto generate simulated controllers to actuate the joints of the robot. The joint position controller is used in our case.

MoveIt! RViz Plugin.

To do visualization and command of the simulated robot, RViz with special MoveIt! plugins is used. Four plugins are available: MotionPlanning, PlanningScene, RobotState and Trajectory. MotionPlanning is a simple gui tool for planning, generating scenes and gathering specific pose data. Planning Scene is used for planning a scene for checking for collision and checking the path before execution. RobotState makes extracting data of the state of the robot more convenient. Trajectory is used for visualizing the trajectory generated by path planning. A trajectory slider can be used to display all the way-points of the trajectory. A demo can be launched by typing the next line in the terminal.

```
roslaunch <name_of_moveit_package> demo.launch
```

Controlling with Python

When controlling the robot with python the moveit_commander package is used. This is included in MoveIt! and full documentation can be found at:

http://docs.ros.org/kinetic/api/moveit_commander/html/annotated.html

Smart wrist specific script

The moveit_commander has to be setup in the python code with the following line:

```
moveit_commander.roscpp_initialize(sys.argv)
```

Now you need to need add the the correct group of joints to MoveGroupCommander. This is set in during setup and is named smart_arm in our case. In the rest of the script self.group is used to give commands to the robot.

```
self.group = moveit_commander.MoveGroupCommander("smart_arm")
```

The simplest way of movement is by setting joint goal values. First the correct message type is pulled from the current joint values. Afterwards ether one or more joints can be given value which corresponds with the joint rotation of the goal. A plan can now be generated with plan() function. If using joint states, the joint states of the goal a required as an argument.

```
# Set joint goal message type
self.joint_goal = self.group.get_current_joint_values()
# Set each joint to the correct value
self.joint_goal[0] = theta.theta0
self.joint_goal[1] = theta.theta1
self.joint_goal[2] = theta.theta2
self.joint_goal[3] = theta.theta3
self.joint_goal[4] = theta.theta4
# Make a plan to move to the next goal
self.plan_of_wrist = self.group.plan(self.joint_goal)
```

Now the path can be executed with the go() function. A goal and if waiting for confirmation on reaching the goal will be used arguments. To ensure the robot is not continuing to move, the stop() function is used.

```
# Execute moving to the joint until movement is reached
self.group.go(self.joint_goal, wait=True)
# Stop movement to ensure that there is no residual movement
self.group.stop()
```

Another way to give movement commands is by giving pose goal for the end-effector. This goal is build up out of position values for x,y and z and quaternion angles for x, y, z and w. It starts by setting the pose goal with the set_pose_target() function. With the pose goal as an argument.

```
self.group.set_pose_target(pose_goal)
```

The plan can now be generated and extracted in the same way as before. Although no argument is now required.

```
self.plan_of_wrist = self.group.plan()
```

Also going to the goal is started with the go() function. Using it this ways only needs a boolean for needing to wait for reaching the goal.

```
self.plan = self.group.go(wait=True)
```

Again stopping and clearing the pose targets is good practice while working with MoveIt!.

```
self.group.stop()
self.group.clear_pose_targets()
```

The final demonstration is at the time of writing still being discussed. To make that implementation further on the road more friendly, setting a goal can be done by publishing it on the “/goal_smart_wirst” topic. This script is listing to this topic by subscribing to it with next command. It requires a message of the type Pose and starts the callback every time it receives a message.

```
self.goal_listen = rospy.Subscriber('/goal_smart_wrist', geometry_msgs.msg.Pose,
self.goal_recieved)
```

The goal _received is a callback that only stores the data in a variable and starts the go to pose goal mentioned earlier.

```
self.data_of_goal = goal_data
self.go_to_pose_goal(self.data_of_goal)
```

In the final application real rotation of the joint of the robot will be send back to ROS to determine the new plan of motion. These angles will send as list of integers that store the values of the encoders attached to the servomotors. The communication between CODESYS and ROS node will publish these values to the “/current_joint_values” topic. The script is subscribed to this and starts the set_current_pose() function.

```
self.real_listen = rospy.Subscriber('/current_joint_values', CurrentJointValues,
self.set_current_pos)
```

A conversion to radians is need before MoveIt! is able to use these values.

```
self.current_pos.position[0] = self.real_joint_angle.theta0 * pi / 3140
```

For this simulation which works with fake controllers the values will be published to the controller joint state.

```
self.give_current_pose =
rospy.Publisher('/move_group/fake_controller_joint_states', JointState,
queue_size=10)
```

This message has to be send in the follow matter:

Header header
    uint32 seq
    time stamp
    string frame_id
string[] name
float64[] position
float64[] velocity
float64[] effort

Sending the trajectory to CODESYS is the last part of controlling with Python. The self made function send_goal() is used. It requires trajectory information being stored in the self.plan_of_wrist variable. Which is done by calling the function plan(), which will calculate a plan in MoveIt!.

Sending the goal starts with gathering the amount of way-points in the trajectory. This can be achieved by using len(object).

```
self.amount_of_waypoints = len(self.plan_of_wrist.joint_trajectory.points)
```

Now the message type has to be set and prepared. This message consist of a integer with the amount of way points followed by lists of joint angles, written as the before mentioned encoder values. First lists have to created for each joint filled with zeros.

```
# Set variable to correct message type
self.servo_waypoints = codesys_joint()
# Set the amount of waypoints
self.servo_waypoints.waypoints = self.amount_of_waypoints
# Create empty lists to store the data
self.servo_waypoints.theta0 = [0] * self.amount_of_waypoints
...
```

Afterwards a for loop with the range from zero to the amount of way-points is used to store, translate and change type for the trajectory.

```
self.moveit_waypoint_theta0 =
self.plan_of_wrist.joint_trajectory.points[nb_of_waypoints].positions[0]
...
self.servo_waypoints.theta0[nb_of_waypoints] =
int((self.moveit_waypoint_theta0 / pi) * 3140)
...
```

These way-points are published to the “/codesys_waypoints” topic. Further on in the project a node can subscribe to the topic and send these values to CODESYS. An example of this message will be given.

```
waypoints: 52
theta0: [-660, -676, -693, -709, -726, -742, ...]
theta1: [1, -9, -19, -29, -38, -48, ...]
theta2: [930, 922, 913, 905, 896, 888, ...]
theta3: [13, -84, -182, -280, -378, -476, ...]
theta4: [0, 1, 1, 2, 2, 3, …]
```

To expand the usability of the commander it was also required to add additional path calculations. For example by giving different inputs on where the robot should go. If only the orientation of the end-effector matters, then the function set_orientation_target(). On the other hand, if only position matters the function set_position_target() is used. To make the make the position only goal work there had to be a variables set in the MoveIt! package. In the kinematic config file the line “position_only_ik: true” had to be added. The setback to this was the other means of setting targets couldn’t work and made it unusable.

For our robotic arm another goal was having a leveled mode, where the top platform needed to be perpendicular to the ground. Which was done by first determining the workspace of the robotic arm. If the goal remains within these boundaries, it is possible to use the xyz position and set the orientation the same a the ground. With this pose_goal the same function set_pose_target() can be used to calculate the trajectory.

All of the earlier mentioned functions were combined in a single script. This script is called “web_moveit_coordinate_node” and does all the communication between MoveIt! and the other ROS nodes. It can subscribe to goal publishers and execute them, for example being given by in the project included web application.

TRAC-IK

To improve the speed of the inverse kinematic calculations without losing precision, a couple of inverse kinematic solvers were tried. The best results were reached by using TRAC-IK kinematics solver, which became part of the MoveIt! at the ROS Indigo/Jade distribution. It lowered the calculation time to average around 20 ms. Also the the Cartesian error distance was with 10^-5 well within spec. It is also solved the problem of having to use the position_only_ik having to be set to true. Because with this kinematic solver all of the variations on setting a target work without change config files and relaunching the package. For ROS kinetic it is required to install the solver with the following command line:

```
sudo apt-get install ros-kinetic-trac-ik-kinematics-plugin
```

Conclusion

MoveIt! has been and still is a good choice for doing path planning and inverse kinematics for our robotic. The setup assistant makes loading in new models very convenient and visual. The menus are straight forward, but also includes countless settings and freedom of adjustment. Running it for the first time has been made easy by automatically creating a package with a demo launch file. Controlling it with Python was a bigger hurdle in this project, because the freedom of movement of this slanted wedge based wrist is rather limited. As a result of this the available poses of the end-effector are only reached with a unique set of joint angles. This requires the goals to be specified very precisely, otherwise the kinematic solver is unable to find a solution for the possible joint angles. The documentation on the functions that are used by move_group node is scares. The main solution on finding out how it works, was by searching the code and trying to decode what is required to reach the desired outcome. Eventually the result is a MoveIt! smart arm specific package and a python script that only requires simple orders to operate. It can generate trajectories for CODESYS and can set encoder data back to joint angles for MoveIt!.