Introduction to MoveIt!

Will Mandil

March 2, 2021

1 Workshop Plan

In this workshop you should gain some practical understanding of the ROS based moveit! package. The workshop is set into 6 sections moving from installation and start up all the way through to programming a robot to follow a set of trajectories.

There are fantastic resources online for any help or for people who want to learn more about MoveIt found **click here**

2 Installation

To install, follow the instructions found here for melodic, **click here** and for kinetic **click here**

You should finish with a catkin workspace called "ws_moveit" with two packages built in the src called panda_moveit_config and moveit_tutorials. For this workshop we will not be using the tutorials package however it is usual to have for reference and further learning.

3 Move Group and Moveit Setup Assistant

From a practical perspective there are two essential parts to the MoveIt package. First is MoveIt setup Assistant, which takes a URDF representation of any robot and converts it to whats called a "config" package, which contains all the files revolving around kinematics, ROS launch files, simulation files and even controllers for hardware such as motor drivers. In the installation section you will have already git cloned the config package for the Franka Emika Panda robot that we use in the robotics labs here at Lincoln, so this stage is already done.

Second is Move Group, which is a python/C++ interface for pragramming motion and interfacing with the robot (in simulation or in reality). In this workshop you will learn the basics of the Move Group interface.

4 Start Up

In this section we will ensure that the installation is correct and will introduce moving the simulated robot without any code.

First run the following command to launch a simulated version of the Franka Panda Robot along and moveit, an rviz window should start up similar to the one shown in 1.

```
roslaunch panda_moveit_config demo.launch
```
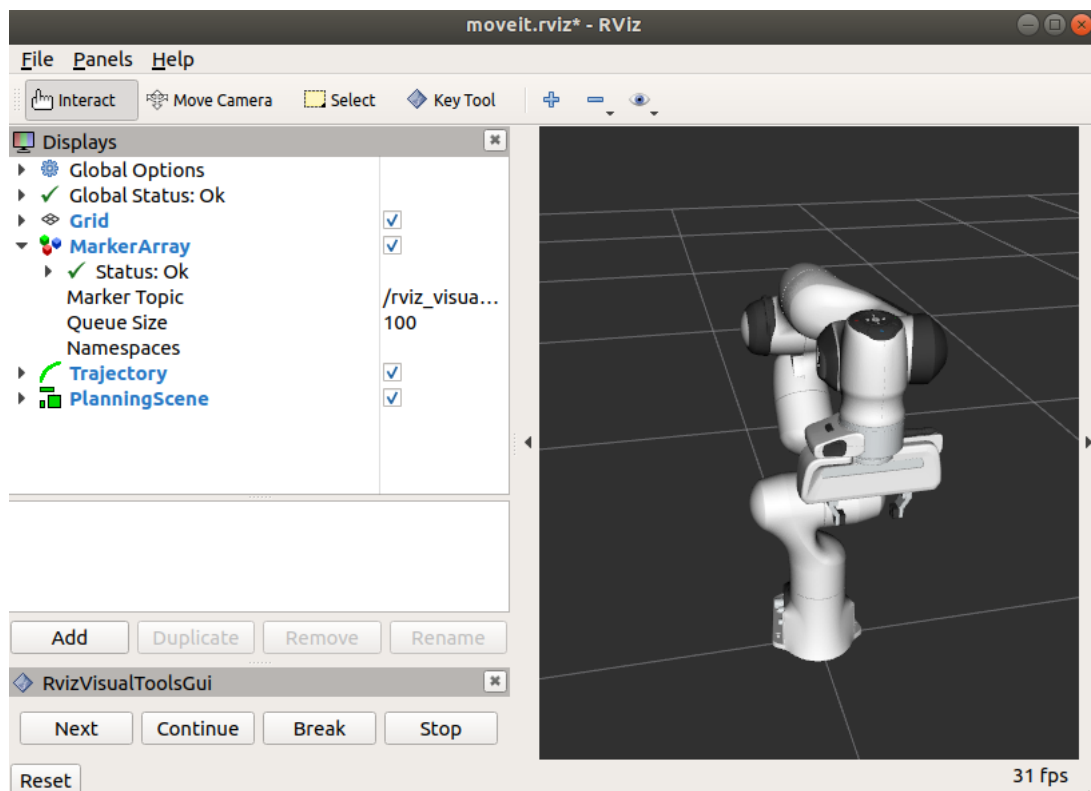

Figure 1: Rviz representation of robot

To double check and develop you understanding of whats going on in the back ground, feel free to have a look at the launch file and view a few of the ros topics being sent, for example you can observe the robots joint states by using the following command:

```
rostopic echo /joint_states
```

5   Waving at the World

Before touching code we will use the Rviz displays to move the robot. Of course this is all in simulation, however if a real robot was connected the following sequence would actually move the robot.

1. In the Rviz window, add the MotionPlanning display ("Add" on the bottom left)

2. Under MotionPlanning/Planned Request/Planning Group, change the value from "hand" to "panda_arm"

3. Under MotionPlanning/Planned Request/Planning Path, tick the value "loop animation". This will allow you to see the planned path before executing it.

4. Use the marker to drag the goal state of the robot around to a new location.

5. Under the "Motion Planning Window" (bottom left) under the "Planning" tab are a set of 4 commands, click "Plan" to plan from the robots current state to the goal state that you declared. This should show a looping animation for the robots planned path.

6. Then by clicking "Execute" the robot should then execute the panned path and move the robot to its goal state.

6 Move Group Start Up

In this section we will create a fresh package with a python script that will be built upon in the upcoming excersizes. To do so, under ws_moveit/src run the following command:

```
catkin_create_package moveit_workshop std_msgs rospy roscpp
```

Within the src of this newly created ROS package create and build (chmod +x filename.py) a python file. Within this python file add the following lines of code and run the script just to make sure all the packages and messages that we will need are installed. The code bellow imports the "moveit_commander" which will give us access to the classes that we need.

```python
#!/usr/bin/env python

import sys
import copy
import rospy
import moveit_commander
import moveit_msgs.msg
import geometry_msgs.msg
from geometry_msgs.msg import PoseStamped
from math import pi
from std_msgs.msg import String
from moveit_commander.conversions import pose_to_list
```

Following this we will initiate the FrankaRobot class (you can call it whateve you like) that will contain all the necessary objects for manipulating a real robot with moveit:

```python
class FrankaRobot(object):
    def __init__(self):
        super(FrankaRobot, self).__init__()
        moveit_commander.roscpp_initialize(sys.argv)
        rospy.init_node('FrankaRobotWorkshop', anonymous=True)
        self.robot = moveit_commander.RobotCommander()
        self.scene = moveit_commander.PlanningSceneInterface()
        self.move_group = moveit_commander.MoveGroupCommander("panda_arm")
        self.display_trajectory_publisher = rospy.Publisher('/move_group/display_planned_path',
                                                            moveit_msgs.msg.DisplayTrajectory,
                                                            queue_size=20)
        self.planning_frame = self.move_group.get_planning_frame()
        self.eef_link = self.move_group.get_end_effector_link()
        self.group_names = self.robot.get_group_names()

if __name__ == '__main__':
    robot = FrankaRobot()
```

We created several essential objects in this class with the following uses:

self.robot = moveit_commander.RobotCommander() provides the class with information such as the robot's kinematic model and the robots current joint states.

self.scene = moveit_commander.PlanningSceneInterface() provides the class with a remote interface for getting, setting and updating the robots internal understanding of the robots surrounding environment.

self.move_group = moveit_commander.MoveGroupCommander("panda_arm") this object is an interface for the planning group (set of robot joints), for this workshop we will only work with the robots arm joints and ignore the robots end effector joints (fingers).

7   Reading Robot States

There are two representations of a robots position that are useful when manipulating a robot, its joint state (the rotation at each joint) and the end effector state (task space), which is the position and orientation of the robots end effector with respect to a base frame.

Try to create two functions within the class that will read the robots state in joint and task space and print the essential values (list of joints and the EE [x,y,z, x,y,z,w]). To do so you will need the following commands for joint and task space respectively:

```
self.group.get_current_joint_values()
self.move_group.get_current_pose()
```
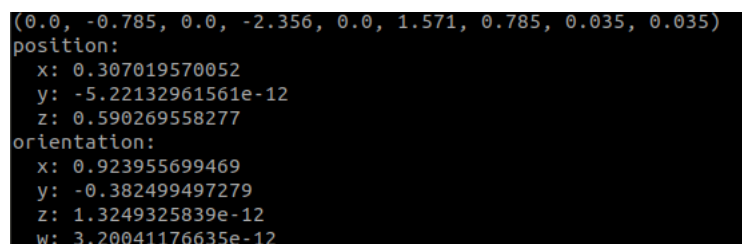
Solutions in the next section

8   Reading Robot States solutions

As you may have discovered the return of the provided functions are ROS messages. And so to access the required values you need to use .pose or .joint_state.position.

```
def get_robot_joint_state(self):
        robot_joint_state = self.robot.get_current_state().joint_state.position
        print(robot_joint_state)


def get_robot_task_state(self):
        robot_ee_pose = self.move_group.get_current_pose().pose
        print(robot_ee_pose)
```

```
(0.0, -0.785, 0.0, -2.356, 0.0, 1.571, 0.785, 0.035, 0.035)
position:
  x: 0.307019570052
  y: -5.22132961561e-12
  z: 0.590269558277
orientation:
  x: 0.923955699469
  y: -0.382499497279
  z: 1.3249325839e-12
  w: 3.20041176635e-12
```

Figure 2: Output of the two functions above

9   Moving the Robot

In order to move the robot we can provide target/goal joint states or EE states for the robot to move to. In order to do so, in task space the following function is required, where "joint_goal"

is a list of joint goal rotations in radians:

```
self.move_group.go(joint_goal, wait=True)
```

For task space, we need a few more steps:

1. ```
   target = self.move_group.set_pose_target(pose)
   ```

   Where "pose" should be in the form of the ROS geometry.msg "PoseStamped", you need to set the **base frame** for the pose (used by the /tf tree to calculate the kinematics), for the franka panda this is "/panda_link0". [TIP: you do not need to populate the seq and time stamp]

2. ```
   self.move_group.go(target)
   ```

To help understand the message requirements for geometry msg PoseStamped, you can use the following command:

```
rosmsg show geometry_msgs/PoseStamped
```

Solutions in the next section

10   Moving the Robot

The two functions below will move the robot to the set locations, you can observe the motions in the Rviz window. For fun, you can try to combine the read states function with the move functions to enable motions based on current state instead of movements relative to a world frame.

```
def go_to_joint_state(self):
        joint_goal = [0, -pi/4, 0, -pi/2, 0, pi/3, 0]
        self.move_group.go(joint_goal, wait=True)
        self.move_group.stop() # ensures there are no residual movements

def go_to_task_state(self):
        p = PoseStamped()
        p.header.frame_id = '/panda_link0'
        p.pose.position.x = 0.45
        p.pose.position.y = -0.25
        p.pose.position.z = 0.45
        p.pose.orientation.x = 1
        p.pose.orientation.y = 0
        p.pose.orientation.z = 0
        p.pose.orientation.w = 0
        target = self.move_group.set_pose_target(p)
        self.move_group.go(target)
```

11   Creating a Cartesian Trajectory

You can plan a Cartesian path directly, by specifying a list of waypoints for the end-effector to go through, try creating your own function in the robot class to perfom the following sequence

of motions:

```python
waypoints = []
wpose = move_group.get_current_pose().pose
wpose.position.z -= scale * 0.1   # First move up (z)
wpose.position.y += scale * 0.2   # and sideways (y)
waypoints.append(copy.deepcopy(wpose))

wpose.position.x += scale * 0.1   # Second move forward/backwards in (x)
waypoints.append(copy.deepcopy(wpose))

wpose.position.y -= scale * 0.1   # Third move sideways (y)
waypoints.append(copy.deepcopy(wpose))
(plan, fraction) = move_group.compute_cartesian_path(waypoints,
# waypoints to follow
                                                     0.01,
# eef_step
                                                     0.0)
```

To have Rviz display the planned trajectory we can publish a trajectory with the message DisplayTrajectory() which we give the start state of the robot and the trajectory plan:

```python
display_trajectory = moveit_msgs.msg.DisplayTrajectory()
display_trajectory.trajectory_start = self.robot.get_current_state()
display_trajectory.trajectory.append(plan)
self.display_trajectory_publisher.publish(display_trajectory)
```

Finally we can execute the planned trajectory with the following command:

```python
raw_input(press enter to execute)
move_group.execute(plan, wait=True)
```

12   Workshop end