Robot Manipulation with MoveIt!

Nicola Castaman
nicola.castaman@dei.unipd.it
What we are going to learn

• Integrate a robotic arm in MoveIt!

• Control a robotic arm with Move Group Interface

• Use the UR10 Manipulator
Integrating an arm in MoveIt!

$ roslaunch moveit_setup_assistant setup_assistant.launch
Integrating an arm in MoveIt!

- **Self-Collisions**: It is used to generate the self-collision matrix. This process is performed automatically by simply setting the sampling density. The collision matrix contains information about how and when links collide in order to improve the performance of the motion planner.

- **Virtual Joints**: It is used to assign virtual joints to the robot. A virtual joint is used to attach the robotic arm to the world. We need virtual joints when the manipulator is not fixed in one place.

- **Planning Groups**: Planning groups are sets of joints that need to be planned together in order to achieve a given goal on a specific link or end effector.

- **Robot Poses**: It gives the ability to define known robot poses in order to be able to reference them later.

- **End Effectors**: It is used to define the robotic arm's end effector.

- **Passive Joints**: It is an optional configuration step, which allows us to define joints that cannot be actuated.

- **Generate the configuration files**: Generate the configuration files.
Motion Planing with RViz

$ roslaunch ur10_moveit demo.launch
Move Group Interface

- Setup
- Planning to a Pose goal
- Planning to a joint-space goal
- Cartesian Paths
- Adding/Removing and Attaching/Detaching Objects
Move Group Interface

Setup

The MoveGroup class is setup using the name of the planning group you would like to control and plan for.

```cpp
moveit::planning_interface::MoveGroupInterface move_group(PLANNING_GROUP);
```

We will use the PlanningSceneInterface class to add and remove collision objects in our “virtual world” scene

```cpp
moveit::planning_interface::PlanningSceneInterface planning_scene_interface;
```

Raw pointers are frequently used to refer to the planning group for improved performance.

```cpp
const robot_state::JointModelGroup *joint_model_group =
    move_group.getCurrentState() -> getJointModelGroup(PLANNING_GROUP);
```
Move Group Interface

Planning to a Pose goal

We can plan a motion for this group to a desired pose for the end-effector.

```cpp
geometry_msgs::Pose target_pose1;
target_pose1.orientation.x = 0.707107;
target_pose1.orientation.y = 0.707107;
target_pose1.orientation.z = 0.00549799;
target_pose1.position.x = 0.37766;
target_pose1.position.y = -0.43118;
target_pose1.position.z = -0.00549799;
move_group.setPoseTarget(target_pose1);
```

Now, we call the planner to compute the plan and visualize it.

```cpp
moveit::planning_interface::MoveGroupInterface::Plan my_plan;
bool success = move_group.plan(my_plan);

ROS_INFO_NAMED("tutorial", "Visualizing plan 1 (pose goal) %s", success ? "" : "FAILED");
```

Note that we are just planning, not asking move_group to actually move the robot.
Move Group Interface

Moving to a pose goal

Moving to a pose goal is similar to the step above except we now use the `move()` function. Note that the pose goal we had set earlier is still active and so the robot will try to move to that goal.

```python
move_group.move();
```
Move Group Interface

Planning to a joint-space goal

We create a pointer that references the current robot’s state. RobotState is the object that contains all the current position/velocity/acceleration data.

```cpp
moveit::core::RobotStatePtr current_state = move_group.getCurrentState();
```

We get the current set of joint values for the group.

```cpp
std::vector<double> joint_group_positions;
current_state->copyJointGroupPositions(joint_model_group,
joint_group_positions);
```

Modify one of the joints and plan to the new joint space goal.

```cpp
joint_group_positions[0] = -1.0; // radians
move_group.setJointValueTarget(joint_group_positions);
success = move_group.plan(my_plan);
```
Move Group Interface

Cartesian Paths

We can plan a cartesian path directly by specifying a list of waypoints for the end-effector to go through. The initial pose (start state) does not need to be added to the waypoint list.

```cpp
std::vector<geometry_msgs::Pose> waypoints;
waypoints.push_back(start_pose2);  // Not needed

geometry_msgs::Pose target_pose3 = start_pose2;

target_pose3.position.z += 0.2;
waypoints.push_back(target_pose3);  // up

target_pose3.position.y -= 0.1;
waypoints.push_back(target_pose3);  // left

target_pose3.position.z -= 0.2;
target_pose3.position.y += 0.2;
target_pose3.position.x -= 0.2;
waypoints.push_back(target_pose3);  // down and right
```
Move Group Interface

Cartesian Paths

Cartesian motions are frequently needed to be slower for actions such as approach and retreat grasp motions. Here is demonstrated how to reduce the speed of the robot arm via a scaling factor of the maximum speed of each joint. **Note:** this is not the speed of the end effector point.

```cpp
move_group.setMaxVelocityScalingFactor(0.1);
```

We want the cartesian path to be interpolated at a resolution of 1 cm which is why we will specify 0.01 as the max step in cartesian translation. We will specify the jump threshold as 0.0, effectively disabling it.

**Warning:** disabling the jump threshold while operating real hardware can cause large unpredictable motions of redundant joints and could be a safety issue.

```cpp
moveit_msgs::RobotTrajectory trajectory;
const double jump_threshold = 0.0;
const double eef_step = 0.01;
double fraction = move_group.computeCartesianPath(waypoints, eef_step, jump_threshold, trajectory);
```
Move Group Interface

Cartesian Paths

Execute a Cartesian Path

The trajectory needs to be modified so it will include velocities as well.

First to create a RobotTrajectory object

```cpp
robot_trajectory::RobotTrajectory rt(group.getCurrentState()->getRobotModel(),
                                      PLANNING_GROUP);
```

Second get a RobotTrajectory from trajectory

```cpp
rt.setRobotTrajectoryMsg(*group.getCurrentState(), trajectory);
```

Third create a IterativeParabolicTimeParameterization object

```cpp
trajectory_processing::IterativeParabolicTimeParameterization iptp;
```

Fourth compute computeTimeStamps

```cpp
bool time_success = iptp.computeTimeStamps(rt);
```

Get RobotTrajectory_msg from RobotTrajectory

```cpp
rt.getRobotTrajectoryMsg(trajectory);
```

Finally plan and execute the trajectory

```cpp
moveit::planning_interface::MoveGroupInterface::Plan my_plan;
my_plan.trajectory_ = trajectory_msg;
move_group.execute(my_plan);
```
Planning Scene Interface

Adding/Removing Objects

Define a collision object ROS message.

```cpp
moveit_msgs::CollisionObject collision_object;
collision_object.header.frame_id = move_group.getPlanningFrame();

collision_object.id = "box1";  // The id of the object is used to identify it

shape_msgs::SolidPrimitive primitive;
primitive.type = primitive.BOX;
primitive.dimensions.resize(3);
primitive.dimensions[0] = 0.4;
primitive.dimensions[1] = 0.1;
primitive.dimensions[2] = 0.4;

geometry_msgs::Pose box_pose;
box_pose.orientation.w = 1.0;
box_pose.position.x = 0.6;
box_pose.position.y = -0.4;
box_pose.position.z = 1.2;

collision_object.primitives.push_back(primitive);
collision_object.primitive_poses.push_back(box_pose);

collision_object.operation = collision_object.ADD;

std::vector<moveit_msgs::CollisionObject> collision_objects;
collision_objects.push_back(collision_object);
```
Planning Scene Interface

Adding/Removing Objects

Add the collision object into the world

```python
planning_scene_interface.addCollisionObjects(collision_objects);
```

Remove the collision object from the world.

```python
planning_scene_interface.removeCollisionObjects(object_ids);
```

Attaching/Detaching Objects

Attach the collision object to the robot.

```python
move_group.attachObject(collision_object.id);
```

Detach the collision object to the robot.

```python
move_group.detachObject(collision_object.id);
```
Working with the UR10

Usage with Gazebo Simulation

Bring up the simulated robot in Gazebo

```
roslaunch ur_gazebo ur10.launch
```

Setup the MoveIt! nodes to allow motion planning

```
roslaunch ur10_moveit_config ur10_moveit_planning_execution.launch sim:=true
```

Start RViz with a configuration including the MoveIt! Motion Planning plugin

```
roslaunch ur10_moveit_config moveit_rviz.launch config:=true
```
Working with the UR10

Usage with Real Robot

Bring up the real robot

```bash
roslaunch ur_bringup ur10_bringup.launch robot_ip:=IP_OF_THE_ROBOT
```

Setup the MoveIt! nodes to allow motion planning

```bash
roslaunch ur10_moveit_config ur10_moveit_planning_execution.launch
```

Start RViz with a configuration including the MoveIt! Motion Planning plugin

```bash
roslaunch ur10_moveit_config moveit_rviz.launch config:=true
```
Working with the UR10

Note

As MoveIt! seems to have difficulties with finding plans for the UR with full joint limits [-2pi, 2pi], there is a joint_limited version using joint limits restricted to [-pi, pi].

In order to use this joint limited version, simply use the launch file arguments 'limited', i.e.:

```
roslaunch ur_bringup ur10_bringup.launch limited:=true
```

```
roslaunch ur10_moveit_config ur10_moveit_planning_execution.launch limited:=true
```

```
roslaunch ur10_moveit_config moveit_rviz.launch config:=true
```
Credits

• MoveIt! Tutorials (http://docs.ros.org/kinetic/api/moveit_tutorials/html/)