moveit$_{t}utorials$

Release Indigo

November 10, 2016

Contents

These tutorials will run you through how to use MoveIt! with your robot. It is assumed that you have already configured MoveIt! for your robot - check the list of robots running MoveIt! to see whether MoveIt! is already available for your robot. Otherwise, skip to the tutorial on Setting up MoveIt! for your robot. If you just want to test MoveIt!, use the PR2 as your quick-start robot.

Note: All tutorials referencing the PR2 have only been tested with ROS Indigo but likely work with ROS Jade. See issue for more information.

Previous version of these tutorials: ROS Indigo

CHAPTER 1

Beginner

The primary user interface to MoveIt! is through the move_group_interface. You can use this interface both through C++ and Python. A GUI-based interface is available through the use of the MoveIt! Rviz Plugin. We will walk through each of these interfaces in detail:

1.1 MoveIt! RViz Plugin Tutorial

MoveIt! comes with a plugin for the ROS Visualizer (RViz). The plugin allows you to setup scenes in which the robot will work, generate plans, visualize the output and interact directly with a visualized robot. We will explore the plugin in this tutorial.

1.1.1 Pre-requisites

You should have completed the MoveIt! Setup Assistant tutorial and you should now have a MoveIt! configuration for the PR2 that you can use. This tutorial assumes the generated MoveIt! configuration package is called "pr2_moveit_config".

Alternately, you can just install the pre-made MoveIt! configuration for the PR2 in the pr2_moveit_config ROS package. To install it, run:

```
sudo apt-get install ros-indigo-moveit-pr2
```

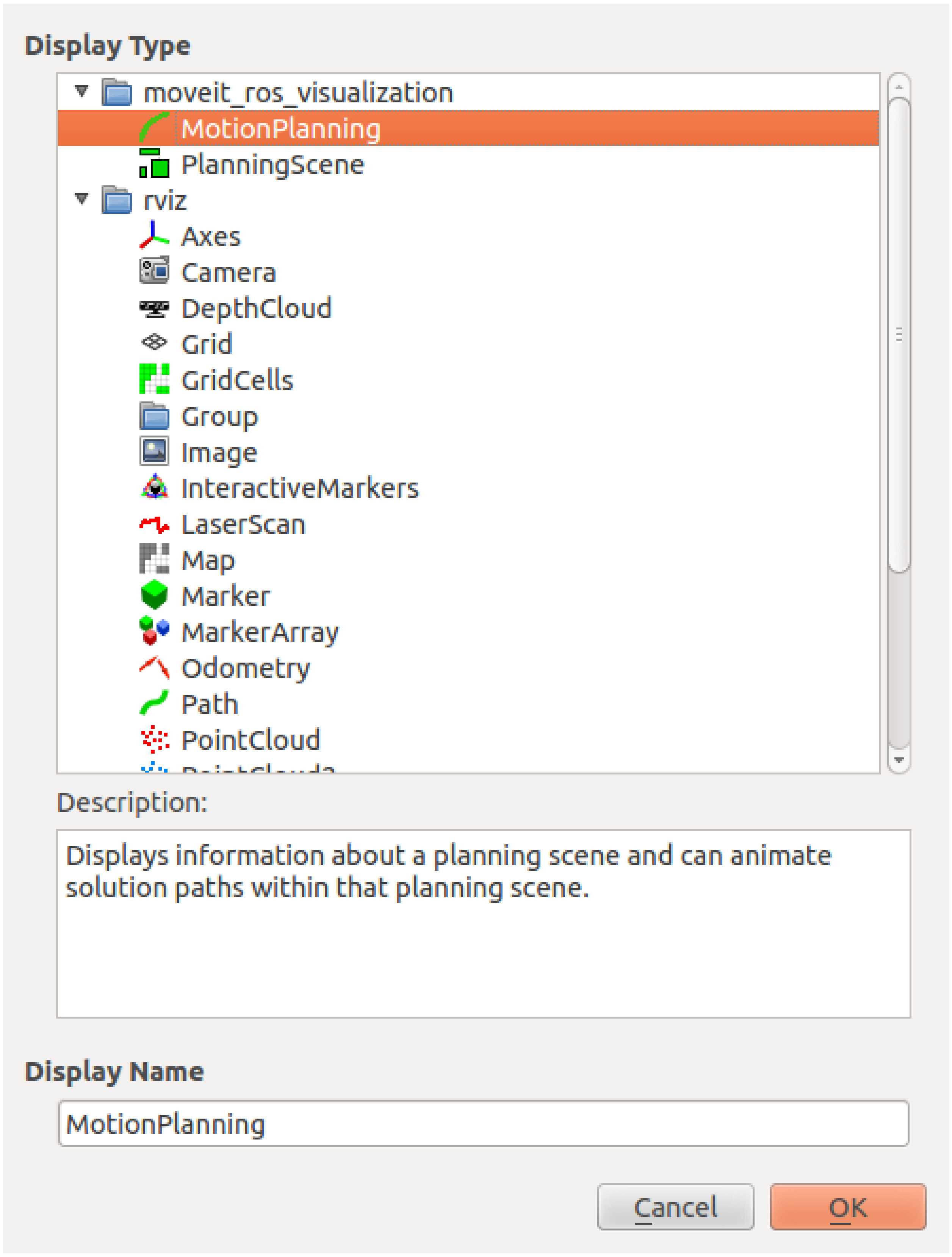
This tutorial does **not** require you to actually have a PR2 robot, it just needs a set of working robot model files.

1.1.2 Step 1: Launch the demo and Configure the Plugin

- Launch the demo:

```
    roslaunch pr2_moveit_config demo.launch
```

- If you are doing this for the first time, you will have to add the Motion Planning Plugin.
  - In the Rviz Displays Tab, press *Add*
  - From the moveit_ros_visualization folder, choose "MotionPlanning" as the DisplayType. Press "Ok".


Display Type

moveit_ros_visualization
MotionPlanning
PlanningScene
rviz
Axes
Camera
DepthCloud
Grid
GridCells
Group
Image
InteractiveMarkers
LaserScan
Map
Marker
MarkerArray
Odometry
Path
PointCloud

Description:

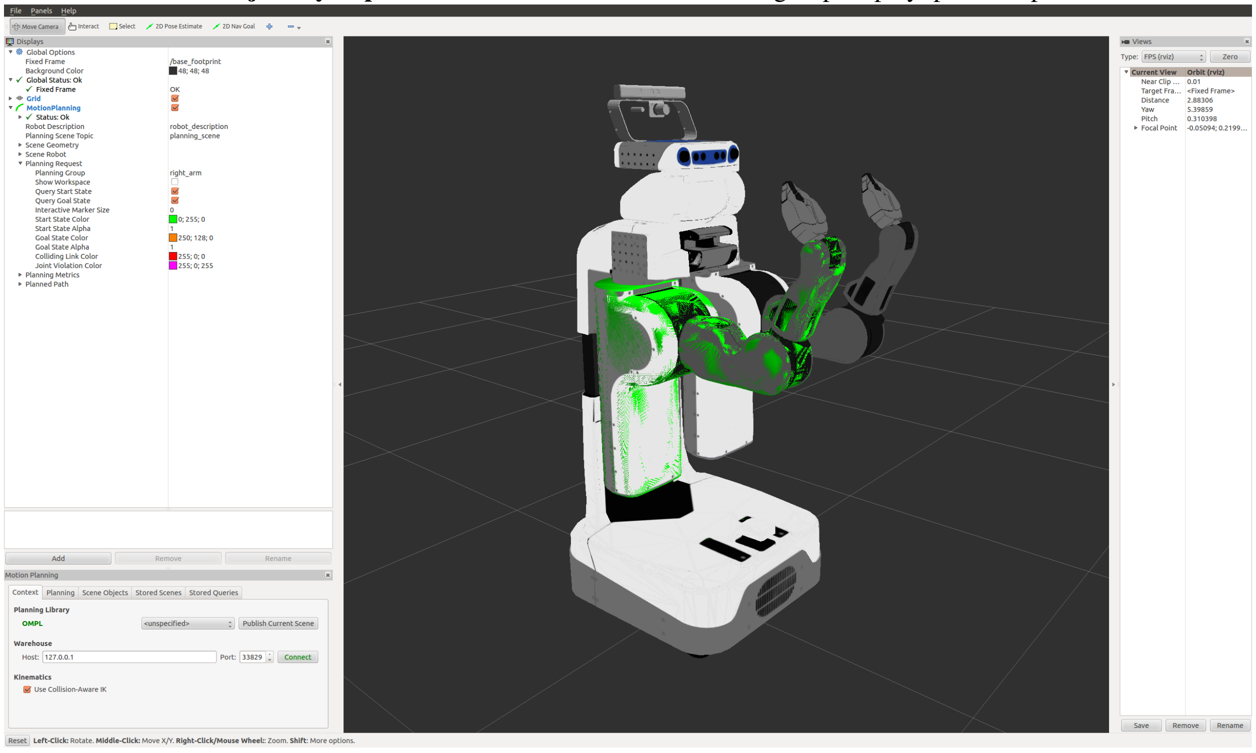
Displays information about a planning scene and can animate solution paths within that planning scene.

Display Name

MotionPlanning

- In the "Global Options" tab of the "Displays" subwindow, set the **Fixed Frame** field to "/odom_combined"

- Now, you can start configuring the Plugin for your robot (the PR2 in this case). Click on “MotionPlanning” in “Displays”.
  - Make sure the **Robot Description** field is set to “robot_description”
  - Make sure the **Planning Scene Topic** field is set to “planning_scene”.
  - In **Planning Request**, change the **Planning Group** to “right_arm”.
  - Set the **Trajectory Topic** in the Planned Path tab to “/move_group/display_planned_path”.



```
▼ Global Options
    Fixed Frame                 /base_footprint
    Background Color            ■ 48; 48; 48
▼ ✓ Global Status: Ok
    ✓ Fixed Frame               OK
▶ Grid                          ✓
▼ MotionPlanning                ✓
  ▶ ✓ Status: Ok
    Robot Description           robot_description
    Planning Scene Topic        planning_scene
  ▶ Scene Geometry
  ▶ Scene Robot
  ▼ Planning Request
      Planning Group            right_arm
      Show Workspace            ☐
      Query Start State         ✓
      Query Goal State          ✓
      Interactive Marker Size   0
      Start State Color         🟩 0; 255; 0
      Start State Alpha         1
      Goal State Color          🟧 250; 128; 0
      Goal State Alpha          1
      Colliding Link Color      🟥 255; 0; 0
      Joint Violation Color     🟪 255; 0; 255
  ▶ Planning Metrics
  ▶ Planned Path
```

```
Planning Library
  OMPL                 <unspecified>      Publish Current Scene
Warehouse
  Host: 127.0.0.1      Port: 33829        Connect
Kinematics
  ✓ Use Collision-Aware IK
```

```
Type: FPS (rviz)
▼ Current View      Orbit (rviz)
    Near Clip ...   0.01
    Target Fra...   <Fixed Frame>
    Distance        2.88306
    Yaw             5.39859
    Pitch           0.310398
  ▶ Focal Point     -0.05094; 0.2199...
```

Left-Click: Rotate. **Middle-Click:** Move X/Y. **Right-Click/Mouse Wheel:**: Zoom. **Shift**: More options.

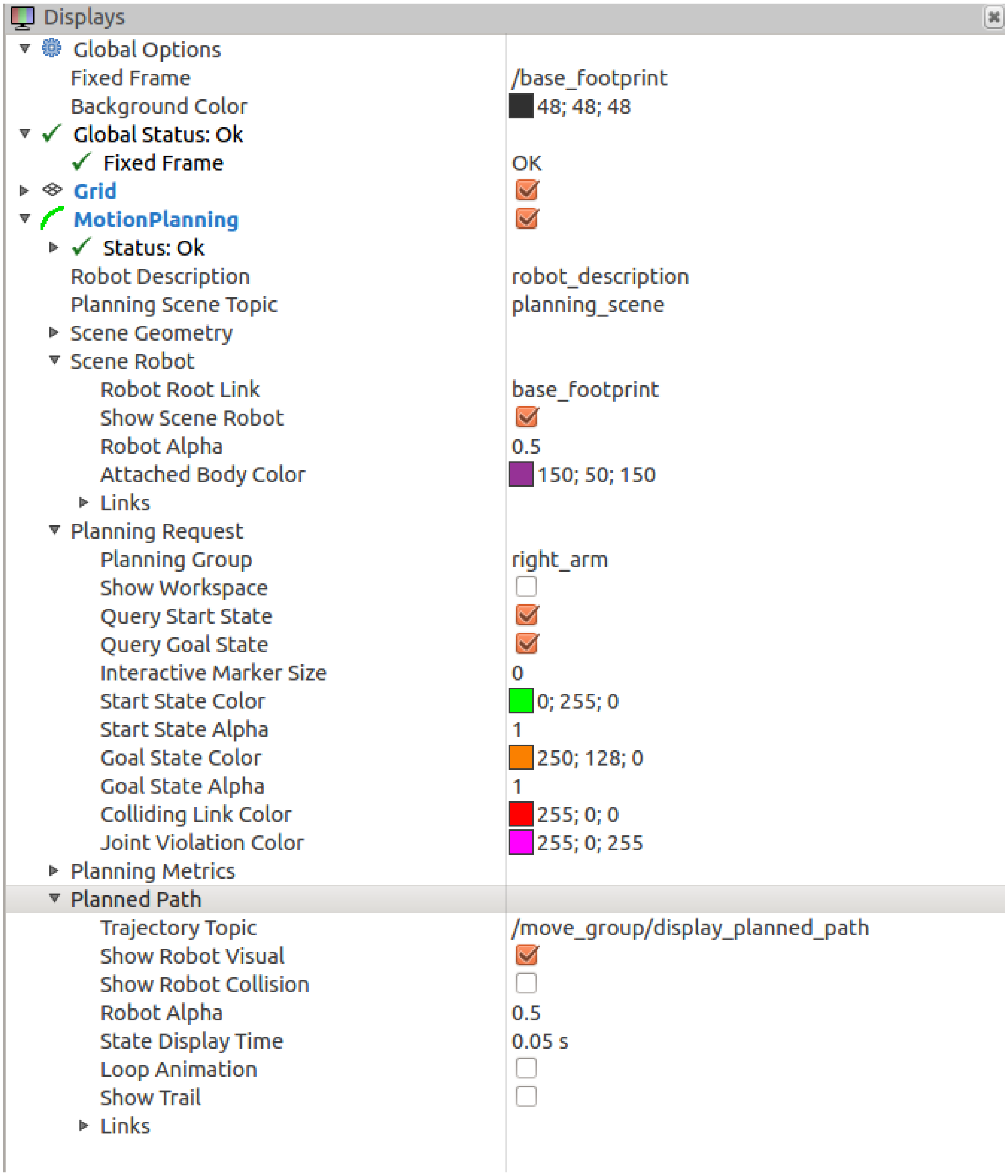
1.1.3 Step 2: Play with the visualized robots

There are four different visualizations active here currently:

1. The start state for motion planning (the planned group is represented in green).
2. The goal state for motion planning (the planned group is represented in orange).
3. The robot’s configuration in the planning scene/ planning environment
4. The planned path for the robot,

The display states for each of these visualizations can be toggled on and off using checkboxes.

1. The start state using the “Query Start State” checkbox in the “Planning Request” tab.
2. The goal state using the “Query Goal State” checkbox in the “Planning Request” tab.
3. The planning scene robot using the “Show Scene Robot” checkbox in the “Scene Robot” tab.
4. The planned path using the “Show Robot Visual” checkbox in the “Planned Path” tab.



Displays

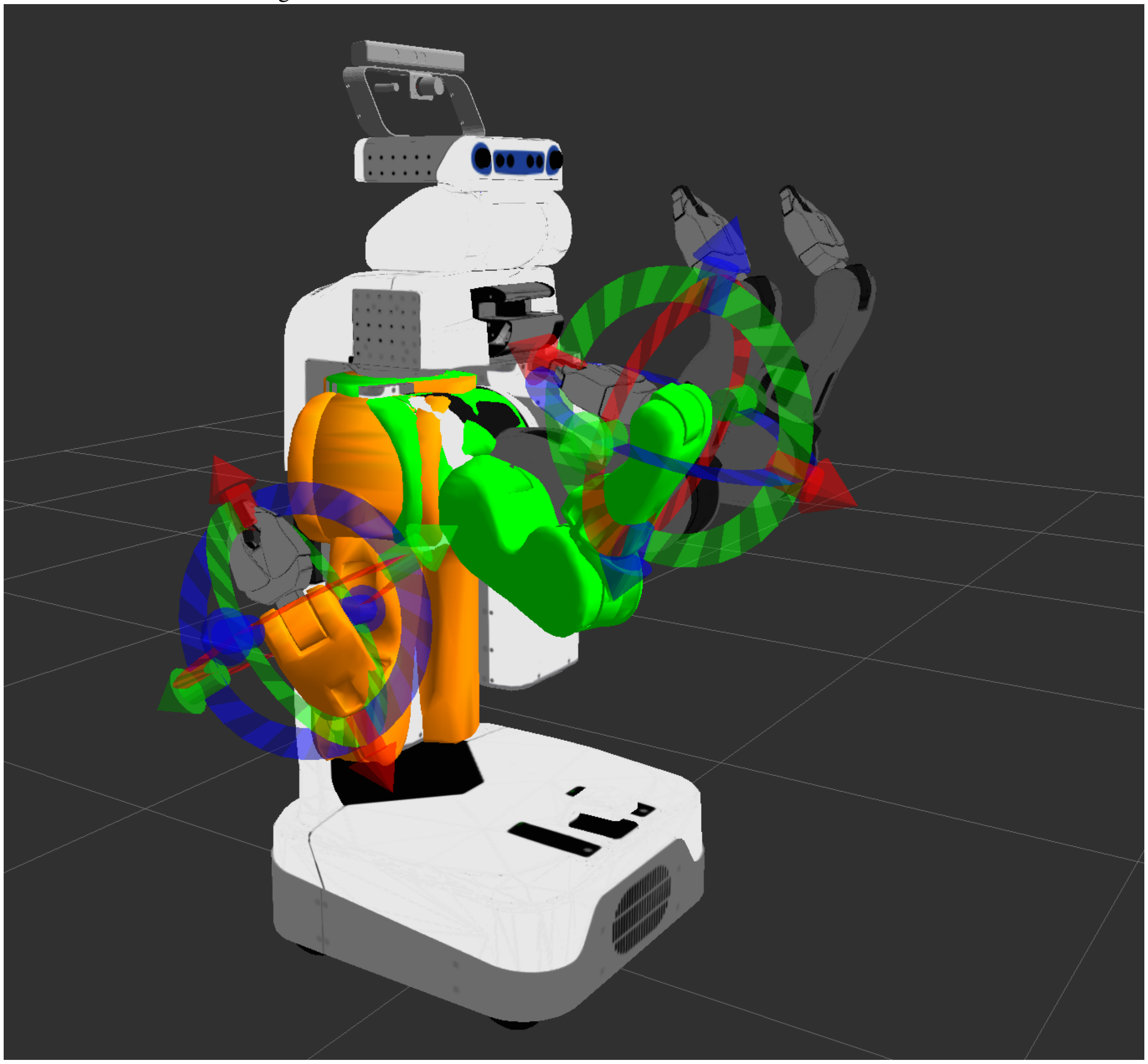
Global Options	
Fixed Frame	/base_footprint
Background Color	⬛ 48; 48; 48
✓ Global Status: Ok	
✓ Fixed Frame	OK
Grid	✓
MotionPlanning	✓
✓ Status: Ok	
Robot Description	robot_description
Planning Scene Topic	planning_scene
Scene Geometry	
Scene Robot	
Robot Root Link	base_footprint
Show Scene Robot	✓
Robot Alpha	0.5
Attached Body Color	🟪 150; 50; 150
Links	
Planning Request	
Planning Group	right_arm
Show Workspace	☐
Query Start State	✓
Query Goal State	✓
Interactive Marker Size	0
Start State Color	🟩 0; 255; 0
Start State Alpha	1
Goal State Color	🟧 250; 128; 0
Goal State Alpha	1
Colliding Link Color	🟥 255; 0; 0
Joint Violation Color	🟪 255; 0; 255
Planning Metrics	
Planned Path	
Trajectory Topic	/move_group/display_planned_path
Show Robot Visual	✓
Show Robot Collision	☐
Robot Alpha	0.5
State Display Time	0.05 s
Loop Animation	☐
Show Trail	☐
Links	

- Play with all these checkboxes to switch on and off different visualizations.

1.1.4 Step 3: Interact with the PR2

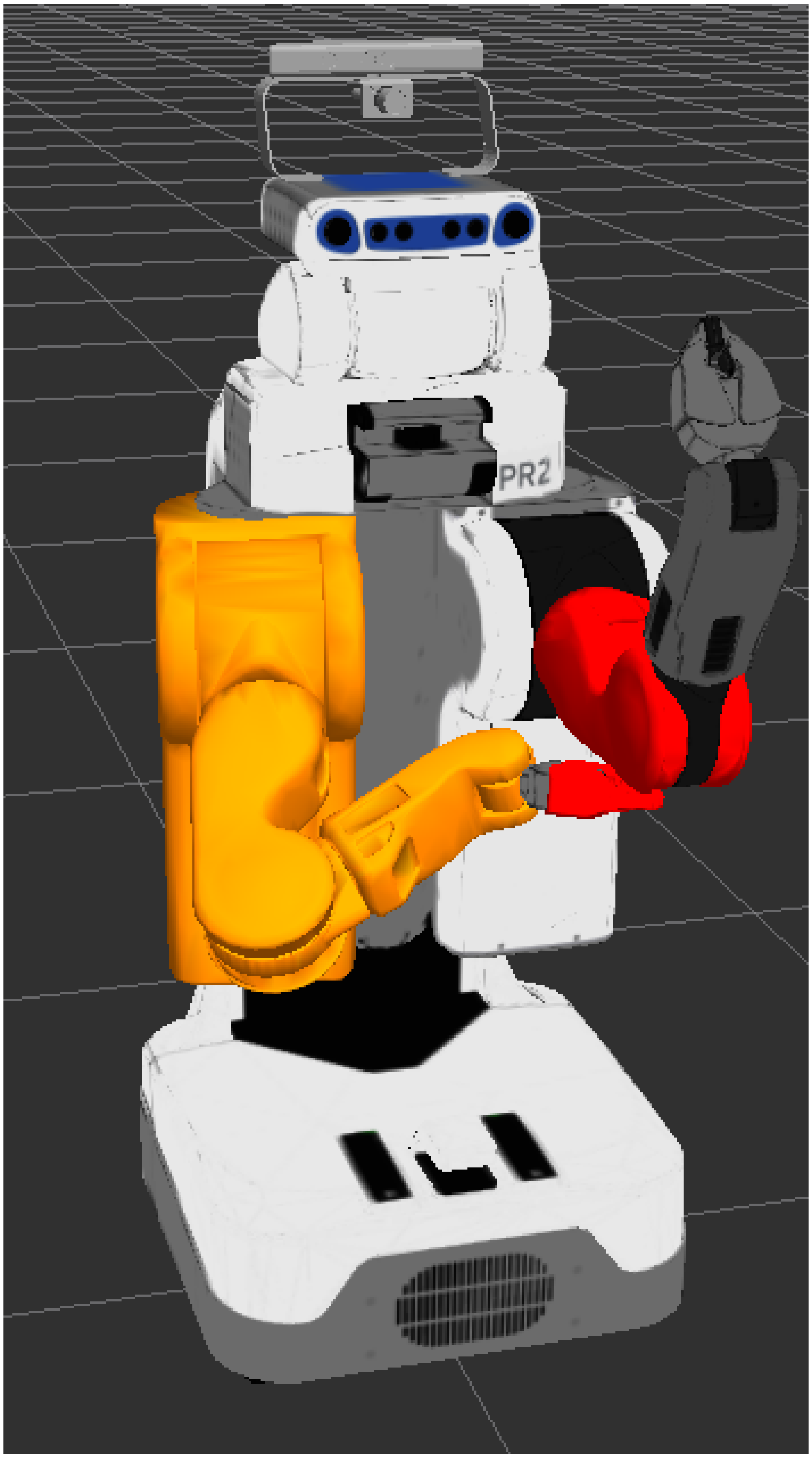
- Press **Interact** in the top menu of rviz. You should see a couple of interactive markers appear for the right arm of the PR2.

- One marker (corresponding to the orange colored right arm) will be used to set the "Goal State" for motion planning. Another marker corresponding to a green colored representation of the right arm will be used to set the "Start State" for motion planning.
- You will be able to use these markers (which are attached to the tip link of each arm) to drag the arm around and change its orientation.

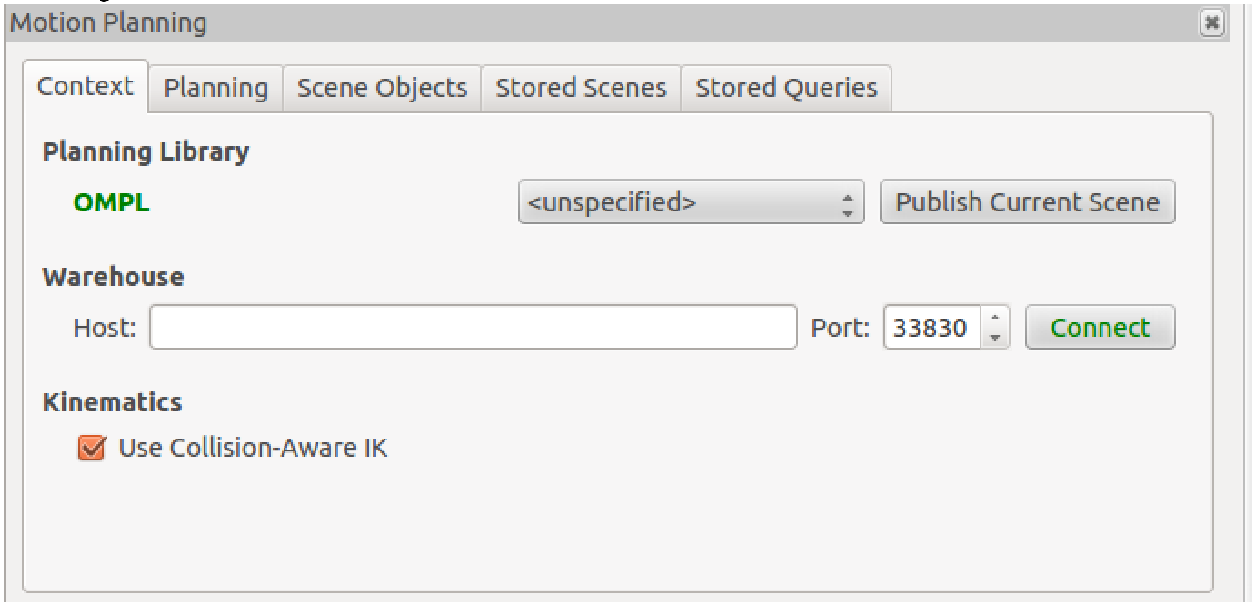


1.1.5 Moving into collision

Note what happens when you try to move one of the arms into collision with the other. The two links that are in collision will turn red.

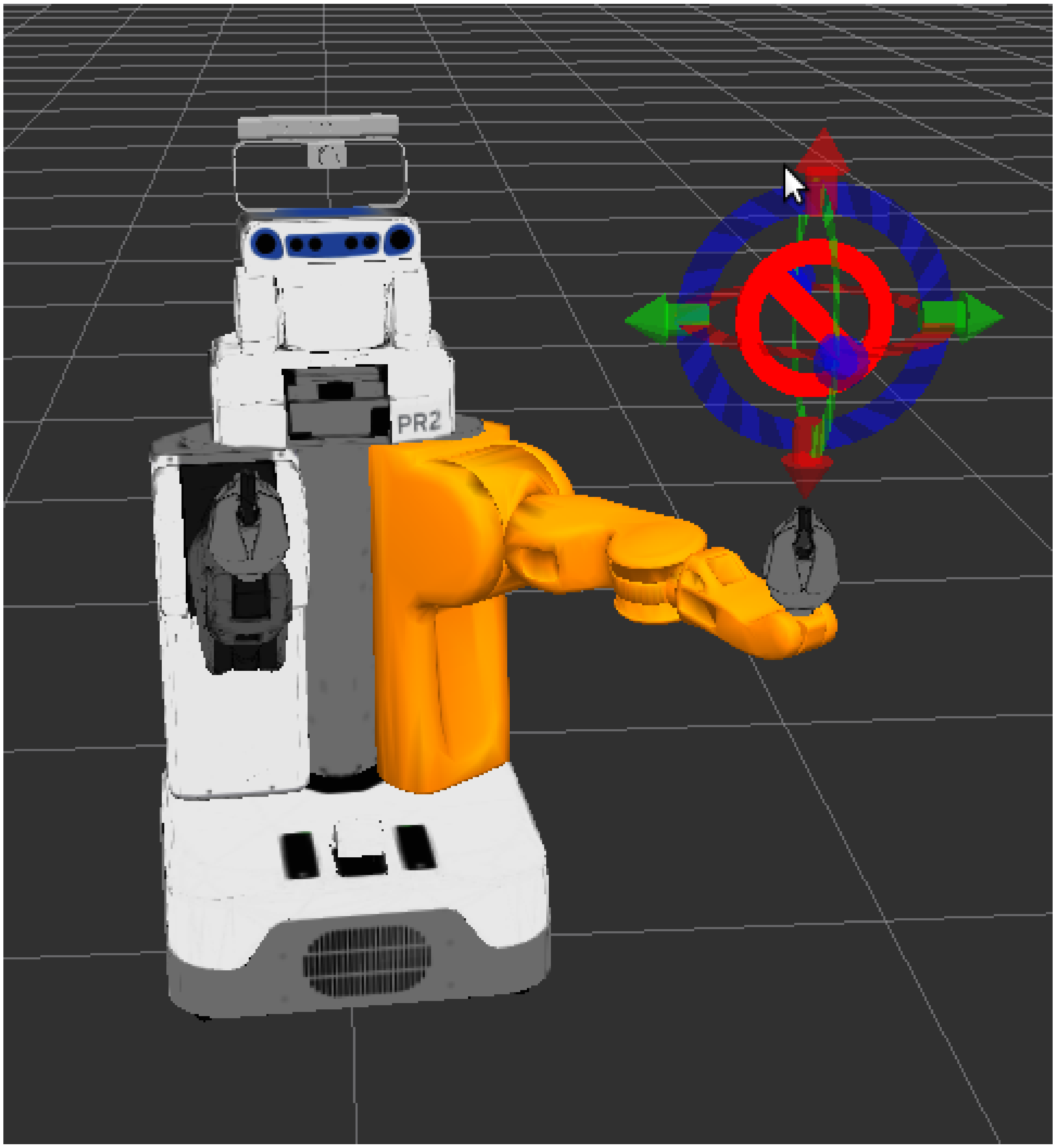

PR2

The “Use Collision-Aware IK” checkbox allows you to toggle the behavior of the IK solver. When the checkbox is ticked, the solver will keep attempting to find a collision-free solution for the desired end-effector pose. When it is unticked, the solver will allow collisions to happen in the solution. The links in collision will always still be visualized in red, regardless of the state of the checkbox.



1.1.6 Moving out of reachable workspace

Note also what happens when you try to move an end-effector out of its reachable workspace (sometimes the access denied sign will not appear).

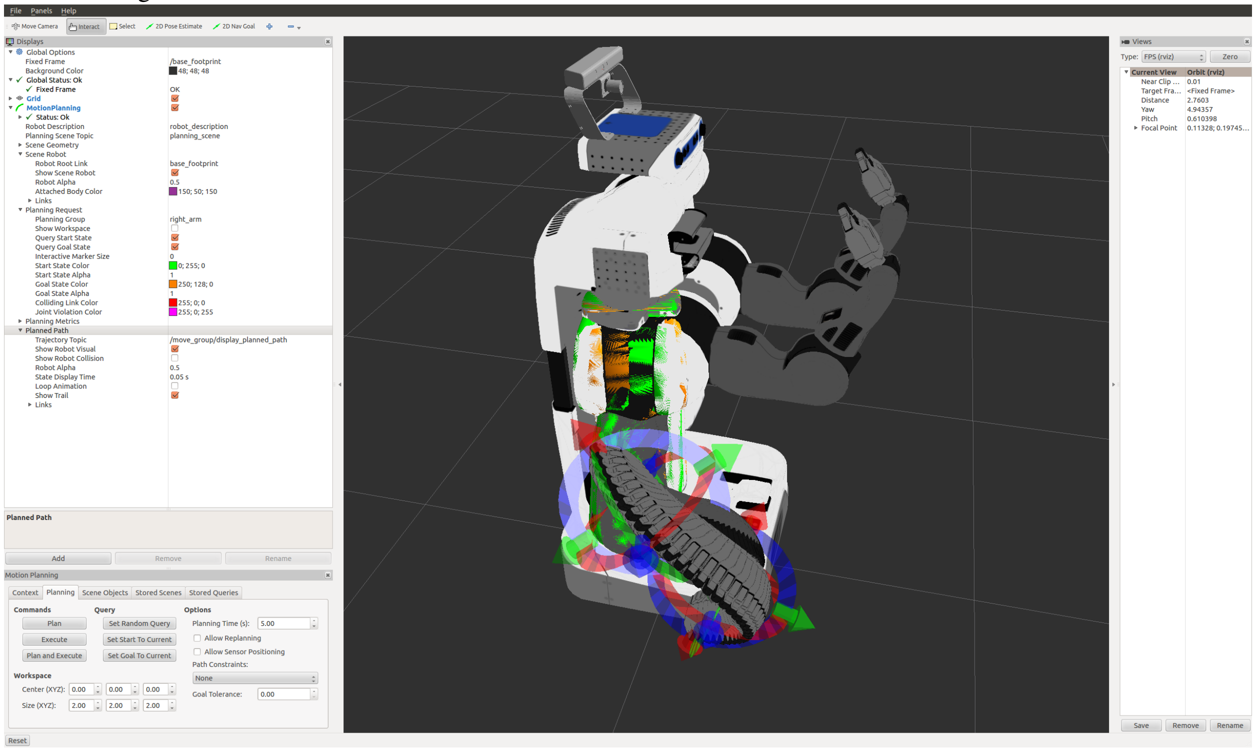


1.1.7 Step 4: Use Motion Planning with the PR2

- Now, you can start motion planning with the PR2 in the MoveIt! Rviz Plugin.
  - Move the Start State to a desired location.
  - Move the Goal State to another desired location.
  - Make sure both states are not in collision with the robot itself.
  - Make sure the Planned Path is being visualized. Also check the "Show Trail" checkbox in the Planned

Path tab.

- In the Planning tab (at the bottom), press the Plan button. You should be able to see a visualization of the arm moving and a trail.



Displays

Global Options	
Fixed Frame	/base_footprint
Background Color	⬛ 48; 48; 48
✓ Global Status: Ok	
✓ Fixed Frame	OK
Grid	✓
MotionPlanning	✓
✓ Status: Ok	
Robot Description	robot_description
Planning Scene Topic	planning_scene
Scene Geometry	
Scene Robot	
Robot Root Link	base_footprint
Show Scene Robot	✓
Robot Alpha	0.5
Attached Body Color	🟪 150; 50; 150
Links	
Planning Request	
Planning Group	right_arm
Show Workspace	☐
Query Start State	✓
Query Goal State	✓
Interactive Marker Size	0
Start State Color	🟩 0; 255; 0
Start State Alpha	1
Goal State Color	🟧 250; 128; 0
Goal State Alpha	1
Colliding Link Color	🟥 255; 0; 0
Joint Violation Color	🟪 255; 0; 255
Planning Metrics	
Planned Path	
Trajectory Topic	/move_group/display_planned_path
Show Robot Visual	✓
Show Robot Collision	☐
Robot Alpha	0.5
State Display Time	0.05 s
Loop Animation	☐
Show Trail	✓
Links	

Planned Path

Motion Planning

Context | Planning | Scene Objects | Stored Scenes | Stored Queries

Commands | Query | Options

Planning Time (s): 5.00
☐ Allow Replanning
☐ Allow Sensor Positioning
Path Constraints: None
Goal Tolerance: 0.00

Workspace
Center (XYZ): 0.00 0.00 0.00
Size (XYZ): 2.00 2.00 2.00

Views

Type: FPS (rviz)

Current View	Orbit (rviz)
Near Clip ...	0.01
Target Fra...	<Fixed Frame>
Distance	2.7603
Yaw	4.94357
Pitch	0.610398
Focal Point	0.11328; 0.19745...

1.1.8 What's Next

- MoveIt! and a simulated robot - You can now start using MoveIt! with a simulated robot in Gazebo.

1.2 Move Group Interface Tutorial

In MoveIt!, the primary user interface is through the MoveGroup class. It provides easy to use functionality for most operations that a user may want to carry out, specifically setting joint or pose goals, creating motion plans, moving the robot, adding objects into the environment and attaching/detaching objects from the robot.

1.2.1 Setup

MoveIt! operates on sets of joints called "planning groups" and stores them in an object called the *JointModelGroup*. Throughout MoveIt! the terms "planning group" and "joint model group" are used interchangably.

```
static const std::string PLANNING_GROUP = "right_arm";
```

The MoveGroup class can be easily setup using just the name of the planning group you would like to control and plan for.

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

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

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

(Optional) Create a publisher for visualizing plans in Rviz.

```
ros::Publisher display_publisher =
    node_handle.advertise<moveit_msgs::DisplayTrajectory>("/move_group/display_planned_path", 1, true
moveit_msgs::DisplayTrajectory display_trajectory;
```

1.2.2 Getting Basic Information

We can print the name of the reference frame for this robot.

```
ROS_INFO("Reference frame: %s", move_group.getPlanningFrame().c_str());
```

We can also print the name of the end-effector link for this group.

```
ROS_INFO("End effector link: %s", move_group.getEndEffectorLink().c_str());
```

1.2.3 Planning to a Pose goal

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

```
geometry_msgs::Pose target_pose1;
target_pose1.orientation.w = 1.0;
target_pose1.position.x = 0.28;
target_pose1.position.y = -0.7;
target_pose1.position.z = 1.0;
move_group.setPoseTarget(target_pose1);
```

Now, we call the planner to compute the plan and visualize it. Note that we are just planning, not asking move_group to actually move the robot.

```
moveit::planning_interface::MoveGroupInterface::Plan my_plan;

bool success = move_group.plan(my_plan);
ROS_INFO("Visualizing plan 1 (pose goal) %s", success ? "" : "FAILED");

/* Sleep to give Rviz time to visualize the plan. */
sleep(5.0);
```

1.2.4 Visualizing plans

Now that we have a plan we can visualize it in Rviz. This is not necessary because the group.plan() call we made above did this automatically. But explicitly publishing plans is useful in cases that we want to visualize a previously created plan.

```
ROS_INFO("Visualizing plan 1 (again)");
display_trajectory.trajectory_start = my_plan.start_state_;
display_trajectory.trajectory.push_back(my_plan.trajectory_);
display_publisher.publish(display_trajectory);
/* Sleep to give Rviz time to visualize the plan. */
sleep(5.0);
```

1.2.5 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. We will not use that function in this tutorial since it is a blocking function and requires a controller to be active and report success on execution of a trajectory.

```
/* Uncomment below line when working with a real robot*/
/* group.move() */
```

1.2.6 Planning to a joint-space goal

Let's set a joint space goal and move towards it. This will replace the pose target we set above.

To start, we'll create an pointer that references the current robot's state. RobotState is the object that contains all the current position/velocity/acceleration data

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

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

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

Now, let's modify one of the joints, plan to the new joint space goal and visualize the plan.

```
joint_group_positions[0] = -1.0;  // radians
move_group.setJointValueTarget(joint_group_positions);

success = move_group.plan(my_plan);
ROS_INFO("Visualizing plan 2 (joint space goal) %s", success ? "" : "FAILED");

/* Sleep to give Rviz time to visualize the plan. */
sleep(5.0);
```

1.2.7 Planning with Path Constraints

Path constraints can easily be specified for a link on the robot. Let's specify a path constraint and a pose goal for our group. First define the path constraint.

```
moveit_msgs::OrientationConstraint ocm;
ocm.link_name = "r_wrist_roll_link";
ocm.header.frame_id = "base_link";
ocm.orientation.w = 1.0;
ocm.absolute_x_axis_tolerance = 0.1;
ocm.absolute_y_axis_tolerance = 0.1;
ocm.absolute_z_axis_tolerance = 0.1;
ocm.weight = 1.0;
```

Now, set it as the path constraint for the group.

```
moveit_msgs::Constraints test_constraints;
test_constraints.orientation_constraints.push_back(ocm);
move_group.setPathConstraints(test_constraints);
```

We will reuse the old goal that we had and plan to it. Note that this will only work if the current state already satisfies the path constraints. So, we need to set the start state to a new pose.

```
robot_state::RobotState start_state(*move_group.getCurrentState());
geometry_msgs::Pose start_pose2;
start_pose2.orientation.w = 1.0;
start_pose2.position.x = 0.55;
start_pose2.position.y = -0.05;
start_pose2.position.z = 0.8;
start_state.setFromIK(joint_model_group, start_pose2);
move_group.setStartState(start_state);
```

Now we will plan to the earlier pose target from the new start state that we have just created.

```
move_group.setPoseTarget(target_pose1);

success = move_group.plan(my_plan);
ROS_INFO("Visualizing plan 3 (constraints) %s", success ? "" : "FAILED");

/* Sleep to give Rviz time to visualize the plan. */
sleep(10.0);
```

When done with the path constraint be sure to clear it.

```
move_group.clearPathConstraints();
```

1.2.8 Cartesian Paths

You can plan a cartesian path directly by specifying a list of waypoints for the end-effector to go through. Note that we are starting from the new start state above. The initial pose (start state) does not need to be added to the waypoint list.

```
std::vector<geometry_msgs::Pose> waypoints;

geometry_msgs::Pose target_pose3 = start_pose2;
target_pose3.position.x += 0.2;
target_pose3.position.z += 0.2;
waypoints.push_back(target_pose3);  // up and out

target_pose3.position.y -= 0.2;
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 (back to start)
```

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

```
moveit_msgs::RobotTrajectory trajectory;
double fraction = move_group.computeCartesianPath(waypoints,
                                                   0.01,  // eef_step
                                                   0.0,   // jump_threshold
                                                   trajectory);
ROS_INFO("Visualizing plan 4 (cartesian path) (%.2f%% acheived)", fraction * 100.0);
```

```
/* Sleep to give Rviz time to visualize the plan. */
sleep(15.0);
```

1.2.9 Adding/Removing Objects and Attaching/Detaching Objects

```

```

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

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

Define a collision object ROS message.

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

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

/* Define a box to add to the world. */
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;

/* A pose for the box (specified relative to frame_id) */
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);
```

Now, let's add the collision object into the world

```
ROS_INFO("Add an object into the world");
planning_scene_interface.addCollisionObjects(collision_objects);

/* Sleep so we have time to see the object in RViz */
sleep(2.0);
```

Planning with collision detection can be slow. Lets set the planning time to be sure the planner has enough time to plan around the box. 10 seconds should be plenty.

```
move_group.setPlanningTime(10.0);
```

Now when we plan a trajectory it will avoid the obstacle

```
move_group.setStartState(*move_group.getCurrentState());
move_group.setPoseTarget(target_pose1);
```

```
success = move_group.plan(my_plan);
ROS_INFO("Visualizing plan 5 (pose goal move around box) %s", success ? "" : "FAILED");

/* Sleep to give Rviz time to visualize the plan. */
sleep(10.0);
```

Now, let's attach the collision object to the robot.

```
ROS_INFO("Attach the object to the robot");
move_group.attachObject(collision_object.id);

/* Sleep to give Rviz time to show the object attached (different color). */
sleep(4.0);
```

Now, let's detach the collision object from the robot.

```
ROS_INFO("Detach the object from the robot");
move_group.detachObject(collision_object.id);

/* Sleep to give Rviz time to show the object detached. */
sleep(4.0);
```

Now, let's remove the collision object from the world.

```
ROS_INFO("Remove the object from the world");
std::vector<std::string> object_ids;
object_ids.push_back(collision_object.id);
planning_scene_interface.removeCollisionObjects(object_ids);

/* Sleep to give Rviz time to show the object is no longer there. */
sleep(4.0);
```

1.2.10 Dual-arm pose goals

First define a new group for addressing the two arms. Then define two separate pose goals, one for each end-effector. Note that we are reusing the goal for the right arm above

```
moveit::planning_interface::MoveGroupInterface two_arms_move_group("arms");

two_arms_move_group.setPoseTarget(target_pose1, "r_wrist_roll_link");

geometry_msgs::Pose target_pose2;
target_pose2.orientation.w = 1.0;
target_pose2.position.x = 0.7;
target_pose2.position.y = 0.15;
target_pose2.position.z = 1.0;

two_arms_move_group.setPoseTarget(target_pose2, "l_wrist_roll_link");
```

Now, we can plan and visualize

```
moveit::planning_interface::MoveGroupInterface::Plan two_arms_plan;
two_arms_move_group.plan(two_arms_plan);
sleep(4.0);
```

1.2.11 The entire code

The entire code can be seen here in the moveit_pr2 github project.

1.2.12 Compiling the code

Follow the instructions for compiling code from source.

1.2.13 The launch file

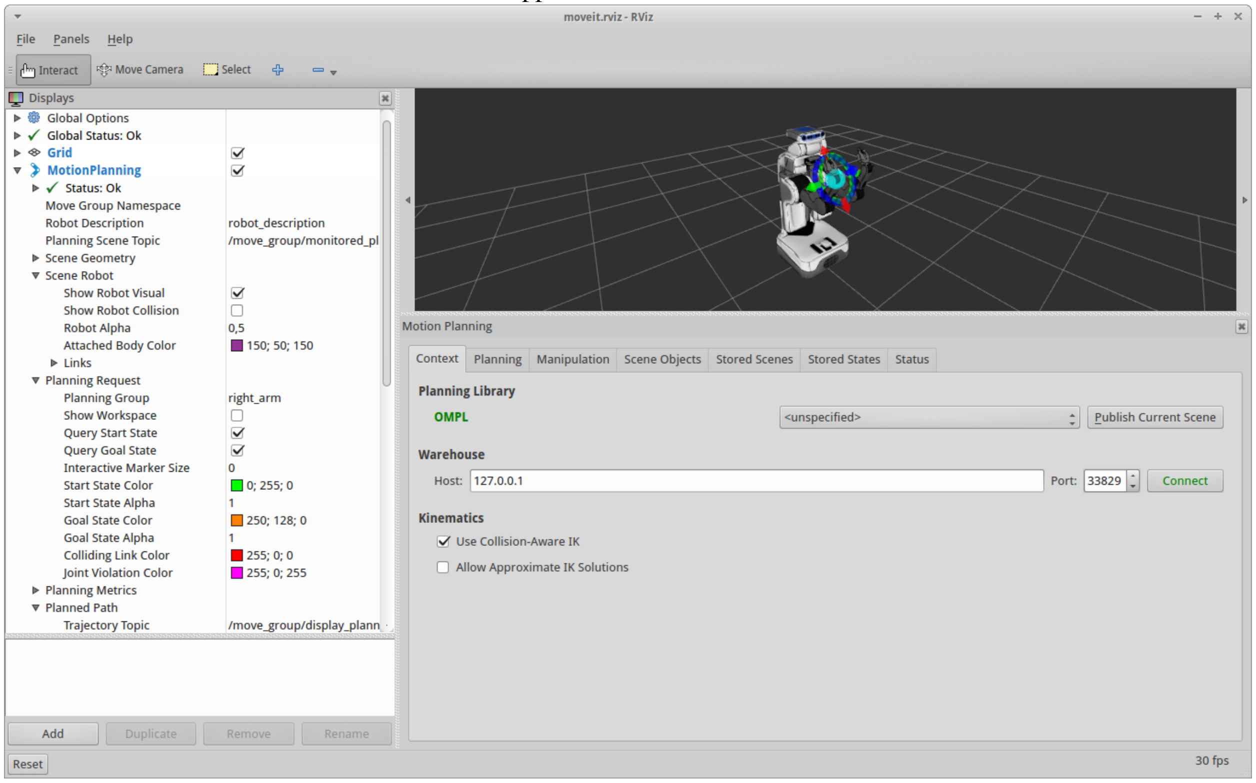
The entire launch file is here on github. All the code in this tutorial can be compiled and run from the moveit_tutorials package that you have as part of your MoveIt! setup.

1.2.14 Running the code

Roslaunch the launch file to run the code directly from moveit_tutorials:

```
roslaunch moveit_tutorials move_group_interface_tutorial.launch
```

After a short moment, the Rviz window should appear:



Global Options	
Global Status: Ok	
Grid	✓
MotionPlanning	✓
Status: Ok	
Move Group Namespace	
Robot Description	robot_description
Planning Scene Topic	/move_group/monitored_pl
Scene Geometry	
Scene Robot	
Show Robot Visual	✓
Show Robot Collision	☐
Robot Alpha	0,5
Attached Body Color	🟪 150; 50; 150
Links	
Planning Request	
Planning Group	right_arm
Show Workspace	☐
Query Start State	✓
Query Goal State	✓
Interactive Marker Size	0
Start State Color	🟩 0; 255; 0
Start State Alpha	1
Goal State Color	🟧 250; 128; 0
Goal State Alpha	1
Colliding Link Color	🟥 255; 0; 0
Joint Violation Color	🟪 255; 0; 255
Planning Metrics	
Planned Path	
Trajectory Topic	/move_group/display_plann

Planning Library

OMPL <unspecified>

Warehouse

Host: 127.0.0.1 Port: 33829

Kinematics

✓ Use Collision-Aware IK

☐ Allow Approximate IK Solutions

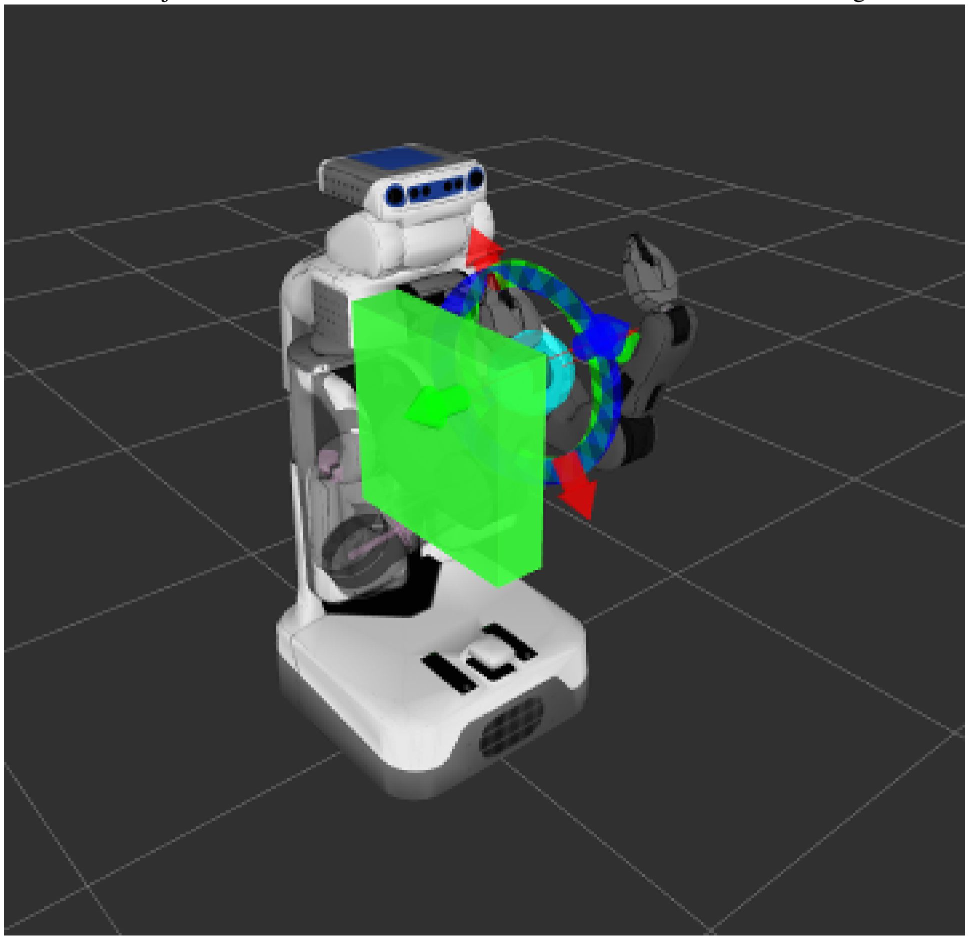
The *Motion Planning* section in the bottom right part of the window can be closed to get a better view of the robot.

1.2.15 Expected Output

In Rviz, we should be able to see the following (there will be a delay of 5-10 seconds between each step):

1. The robot moves its right arm to the pose goal to its right front.

2. The robot repeats the same motion from 1.
3. The robot moves its right arm to the joint goal at its right side.
4. The robot moves its right arm back to a new pose goal while maintaining the end-effector level.
5. The robot moves its right arm along the desired cartesian path (a triangle up+forward, left, down+back).
6. A box object is added into the environment to the right of the right arm.



7. The robot moves its right arm to the pose goal, avoiding collision with the box.
8. The object is attached to the wrist (its color will change to purple/orange/green).
9. The object is detached from the wrist (its color will change back to green).
10. The object is removed from the environment.

1.3 Move Group Python Interface Tutorial

In MoveIt!, the primary user interface is through the RobotCommander class. It provides functionality for most operations that a user may want to carry out, specifically setting joint or pose goals, creating motion plans, moving the robot, adding objects into the environment and attaching/detaching objects from the robot.

1.3.1 Setup

To use the python interface to move_group, import the moveit_commander module. We also import rospy and some messages that we will use.

```
import sys
import copy
import rospy
import moveit_commander
import moveit_msgs.msg
import geometry_msgs.msg
```

First initialize moveit_commander and rospy.

```
print "============ Starting tutorial setup"
moveit_commander.roscpp_initialize(sys.argv)
rospy.init_node('move_group_python_interface_tutorial',
                anonymous=True)
```

Instantiate a RobotCommander object. This object is an interface to the robot as a whole.

```
robot = moveit_commander.RobotCommander()
```

Instantiate a PlanningSceneInterface object. This object is an interface to the world surrounding the robot.

```
scene = moveit_commander.PlanningSceneInterface()
```

Instantiate a MoveGroupCommander object. This object is an interface to one group of joints. In this case the group is the joints in the left arm. This interface can be used to plan and execute motions on the left arm.

```
group = moveit_commander.MoveGroupCommander("left_arm")
```

We create this DisplayTrajectory publisher which is used below to publish trajectories for RVIZ to visualize.

```
display_trajectory_publisher = rospy.Publisher(
                                    '/move_group/display_planned_path',
                                    moveit_msgs.msg.DisplayTrajectory)
```

Wait for RVIZ to initialize. This sleep is ONLY to allow Rviz to come up.

```
print "============ Waiting for RVIZ..."
rospy.sleep(10)
print "============ Starting tutorial "
```

1.3.2 Getting Basic Information

We can get the name of the reference frame for this robot

```
print "============ Reference frame: %s" % group.get_planning_frame()
```

We can also print the name of the end-effector link for this group

```
print "============ Reference frame: %s" % group.get_end_effector_link()
```

We can get a list of all the groups in the robot

```
print "============ Robot Groups:"
print robot.get_group_names()
```

Sometimes for debugging it is useful to print the entire state of the robot.

```
print "============ Printing robot state"
print robot.get_current_state()
print "============"
```

1.3.3 Planning to a Pose goal

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

```
print "============ Generating plan 1"
pose_target = geometry_msgs.msg.Pose()
pose_target.orientation.w = 1.0
pose_target.position.x = 0.7
pose_target.position.y = -0.05
pose_target.position.z = 1.1
group.set_pose_target(pose_target)
```

Now, we call the planner to compute the plan and visualize it if successful Note that we are just planning, not asking move_group to actually move the robot

```
plan1 = group.plan()

print "============ Waiting while RVIZ displays plan1..."
rospy.sleep(5)
```

You can ask RVIZ to visualize a plan (aka trajectory) for you. But the group.plan() method does this automatically so this is not that useful here (it just displays the same trajectory again).

```
print "============ Visualizing plan1"
display_trajectory = moveit_msgs.msg.DisplayTrajectory()

display_trajectory.trajectory_start = robot.get_current_state()
display_trajectory.trajectory.append(plan1)
display_trajectory_publisher.publish(display_trajectory);

print "============ Waiting while plan1 is visualized (again)..."
rospy.sleep(5)
```

1.3.4 Moving to a pose goal

Moving to a pose goal is similar to the step above except we now use the go() function. Note that the pose goal we had set earlier is still active and so the robot will try to move to that goal. We will not use that function in this tutorial since it is a blocking function and requires a controller to be active and report success on execution of a trajectory.

```
# Uncomment below line when working with a real robot
# group.go(wait=True)
```

1.3.5 Planning to a joint-space goal

Let's set a joint space goal and move towards it. First, we will clear the pose target we had just set.

```
group.clear_pose_targets()
```

Then, we will get the current set of joint values for the group

```
group_variable_values = group.get_current_joint_values()
print "============ Joint values: ", group_variable_values
```

Now, let's modify one of the joints, plan to the new joint space goal and visualize the plan

```
group_variable_values[0] = 1.0
group.set_joint_value_target(group_variable_values)

plan2 = group.plan()

print "============ Waiting while RVIZ displays plan2..."
rospy.sleep(5)
```

1.3.6 Cartesian Paths

You can plan a cartesian path directly by specifying a list of waypoints for the end-effector to go through.

```
waypoints = []

# start with the current pose
waypoints.append(group.get_current_pose().pose)

# first orient gripper and move forward (+x)
wpose = geometry_msgs.msg.Pose()
wpose.orientation.w = 1.0
wpose.position.x = waypoints[0].position.x + 0.1
wpose.position.y = waypoints[0].position.y
wpose.position.z = waypoints[0].position.z
waypoints.append(copy.deepcopy(wpose))

# second move down
wpose.position.z -= 0.10
waypoints.append(copy.deepcopy(wpose))

# third move to the side
wpose.position.y += 0.05
waypoints.append(copy.deepcopy(wpose))
```

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

```
(plan3, fraction) = group.compute_cartesian_path(
                             waypoints,   # waypoints to follow
                             0.01,        # eef_step
                             0.0)         # jump_threshold

print "============ Waiting while RVIZ displays plan3..."
rospy.sleep(5)
```

1.3.7 Adding/Removing Objects and Attaching/Detaching Objects

First, we will define the collision object message

```
collision_object = moveit_msgs.msg.CollisionObject()
```

When finished shut down moveit_commander.

```
moveit_commander.roscpp_shutdown()
```

1.3.8 The entire code

The entire code can be seen here in the moveit_pr2 github project.

1.3.9 The launch file

The entire launch file is here on github. All the code in this tutorial can be run from the moveit_tutorials package that you have as part of your MoveIt! setup.

1.3.10 Running the code

Make sure your python file is executable:

```
roscd moveit_tutorials/doc/pr2_tutorials/planning/scripts/
chmod +x move_group_python_interface_tutorial.py
```

Launch the moveit! demo interface:

```
roslaunch pr2_moveit_config demo.launch
```

Now run the python code directly using rosrun:

```
rosrun moveit_tutorials move_group_python_interface_tutorial.py
```

Please note that due to a bug in ros-Indigo discussed in issue #15 the moveit_commander throws an exception when shutting down. This does not interfere with the functioning of the code itself.

1.3.11 Expected Output

In Rviz, we should be able to see the following (there will be a delay of 5-10 seconds between each step):

1. The robot moves its left arm to the pose goal in front of it (plan1)
2. The robot again moves its left arm to the same goal (plan1 again)
3. The robot moves its left arm to the joint goal to the side,
4. The robot moves its left arm along the desired cartesian path.

CHAPTER 2

Advanced

This set of advanced tutorials is meant for developers who are using MoveIt!'s C++ API. Most users wanting to access MoveIt! in C++ or Python should use the move_group_interface (above).

2.1 Kinematic Model Tutorial

In this section, we will walk you through the C++ API for using kinematics.

2.1.1 The RobotModel and RobotState classes

The RobotModel and RobotState classes are the core classes that give you access to the kinematics. In this example, we will walk through the process of using the classes for the right arm of the PR2.

Start

Setting up to start using the RobotModel class is very easy. In general, you will find that most higher-level components will return a shared pointer to the RobotModel. You should always use that when possible. In this example, we will start with such a shared pointer and discuss only the basic API. You can have a look at the actual code API for these classes to get more information about how to use more features provided by these classes.

We will start by instantiating a RobotModelLoader object, which will look up the robot description on the ROS parameter server and construct a RobotModel for us to use.

```
robot_model_loader::RobotModelLoader robot_model_loader("robot_description");
robot_model::RobotModelPtr kinematic_model = robot_model_loader.getModel();
ROS_INFO("Model frame: %s", kinematic_model->getModelFrame().c_str());
```

Using the RobotModel, we can construct a RobotState that maintains the configuration of the robot. We will set all joints in the state to their default values. We can then get a JointModelGroup, which represents the robot model for a particular group, e.g. the "right_arm" of the PR2 robot.

```
robot_state::RobotStatePtr kinematic_state(new robot_state::RobotState(kinematic_model));
kinematic_state->setToDefaultValues();
const robot_state::JointModelGroup *joint_model_group = kinematic_model->getJointModelGroup("right_a

const std::vector<std::string> &joint_names = joint_model_group->getJointModelNames();
```

Get Joint Values

We can retreive the current set of joint values stored in the state for the right arm.

```
std::vector<double> joint_values;
kinematic_state->copyJointGroupPositions(joint_model_group, joint_values);
for (std::size_t i = 0; i < joint_names.size(); ++i)
{
  ROS_INFO("Joint %s: %f", joint_names[i].c_str(), joint_values[i]);
}
```

Joint Limits

setJointGroupPositions() does not enforce joint limits by itself, but a call to enforceBounds() will do it.

```
/* Set one joint in the right arm outside its joint limit */
joint_values[0] = 1.57;
kinematic_state->setJointGroupPositions(joint_model_group, joint_values);

/* Check whether any joint is outside its joint limits */
ROS_INFO_STREAM("Current state is " << (kinematic_state->satisfiesBounds() ? "valid" : "not valid"));

/* Enforce the joint limits for this state and check again*/
kinematic_state->enforceBounds();
ROS_INFO_STREAM("Current state is " << (kinematic_state->satisfiesBounds() ? "valid" : "not valid"));
```

Forward Kinematics

Now, we can compute forward kinematics for a set of random joint values. Note that we would like to find the pose of the "r_wrist_roll_link" which is the most distal link in the "right_arm" of the robot.

```
kinematic_state->setToRandomPositions(joint_model_group);
const Eigen::Affine3d &end_effector_state = kinematic_state->getGlobalLinkTransform("r_wrist_roll_lin

/* Print end-effector pose. Remember that this is in the model frame */
ROS_INFO_STREAM("Translation: " << end_effector_state.translation());
ROS_INFO_STREAM("Rotation: " << end_effector_state.rotation());
```

Inverse Kinematics

We can now solve inverse kinematics (IK) for the right arm of the PR2 robot. To solve IK, we will need the following: * The desired pose of the end-effector (by default, this is the last link in the "right_arm" chain): end_effector_state that we computed in the step above. * The number of attempts to be made at solving IK: 5 * The timeout for each attempt: 0.1 s

```
bool found_ik = kinematic_state->setFromIK(joint_model_group, end_effector_state, 10, 0.1);
```

Now, we can print out the IK solution (if found):

```
if (found_ik)
{
  kinematic_state->copyJointGroupPositions(joint_model_group, joint_values);
  for (std::size_t i = 0; i < joint_names.size(); ++i)
  {
    ROS_INFO("Joint %s: %f", joint_names[i].c_str(), joint_values[i]);
  }
}
else
{
  ROS_INFO("Did not find IK solution");
}
```

Get the Jacobian

We can also get the Jacobian from the RobotState.

```
Eigen::Vector3d reference_point_position(0.0, 0.0, 0.0);
Eigen::MatrixXd jacobian;
kinematic_state->getJacobian(joint_model_group,
                             kinematic_state->getLinkModel(joint_model_group->getLinkModelNames().bac
                             reference_point_position, jacobian);
ROS_INFO_STREAM("Jacobian: " << jacobian);
```

The entire code

The entire code can be seen here in the moveit_pr2 github project.

Compiling the code

Follow the instructions for compiling code from source.

The launch file

To run the code, you will need a launch file that does two things:

- Uploads the PR2 URDF and SRDF onto the param server, and
- Puts the kinematics_solver configuration generated by the MoveIt! Setup Assistant onto the ROS parameter server in the namespace of the node that instantiates the classes in this tutorial.

```
<launch>
  <include file="$(find pr2_moveit_config)/launch/planning_context.launch">
    <arg name="load_robot_description" value="true"/>
  </include>

  <node name="kinematic_model_tutorial"
        pkg="moveit_tutorials"
        type="kinematic_model_tutorial"
        respawn="false" output="screen">
    <rosparam command="load"
              file="$(find pr2_moveit_config)/config/kinematics.yaml"/>
  </node>
</launch>
```

All the code in this tutorial can be compiled and run from the moveit_tutorials package that you have as part of your MoveIt! setup.

Running the code

Roslaunch the launch file to run the code directly from moveit_tutorials:

```
roslaunch moveit_tutorials kinematic_model_tutorial.launch
```

Expected Output

The expected output will be in the following form. The numbers will not match since we are using random joint values:

```
[ INFO] [1384819451.749126980]: Model frame: /odom_combined
[ INFO] [1384819451.749228320]: Joint r_shoulder_pan_joint: 0.000000
[ INFO] [1384819451.749268059]: Joint r_shoulder_lift_joint: 0.000000
[ INFO] [1384819451.749298929]: Joint r_upper_arm_roll_joint: 0.000000
[ INFO] [1384819451.749337412]: Joint r_upper_arm_joint: -1.135650
[ INFO] [1384819451.749376593]: Joint r_elbow_flex_joint: 0.000000
[ INFO] [1384819451.749404669]: Joint r_forearm_roll_joint: -1.050000
[ INFO] [1384819451.749480167]: Joint r_forearm_joint: 0.000000
[ INFO] [1384819451.749545399]: Joint r_wrist_flex_joint: 0.000000
[ INFO] [1384819451.749577945]: Joint r_wrist_roll_joint: 0.000000
[ INFO] [1384819451.749660707]: Current state is not valid
[ INFO] [1384819451.749723425]: Current state is valid
[ INFO] [1384819451.750553768]: Translation: 0.0464629
-0.660372
1.08426
[ INFO] [1384819451.750715730]: Rotation: -0.0900434  -0.945724  -0.312248
  -0.91467 -0.0455189    0.40163
 -0.394045   0.321768  -0.860926
[ INFO] [1384819451.751673044]: Joint r_shoulder_pan_joint: -1.306166
[ INFO] [1384819451.751729266]: Joint r_shoulder_lift_joint: 0.502776
[ INFO] [1384819451.751791355]: Joint r_upper_arm_roll_joint: 0.103366
[ INFO] [1384819451.751853793]: Joint r_upper_arm_joint: -1.975539
[ INFO] [1384819451.751907012]: Joint r_elbow_flex_joint: 2.805000
[ INFO] [1384819451.751963863]: Joint r_forearm_roll_joint: -1.851939
[ INFO] [1384819451.752015895]: Joint r_forearm_joint: -0.414720
[ INFO] [1384819451.752064923]: Joint r_wrist_flex_joint: 0.000000
[ INFO] [1384819451.752117933]: Joint r_wrist_roll_joint: 0.000000
[ INFO] [1384819451.753252155]: Jacobian:    0.472372   0.0327816    -0.28711   0.0708816 1.38778e-1
  0.0964629   -0.120972  -0.0626032   -0.311436           0           0           0
          0   -0.381159  -0.0266776  -0.0320097 8.67362e-19           0           0
          0    0.965189    0.229185    0.973042  -0.0665617   -0.991369  -0.0900434
          0    0.261552   -0.845745    0.212168   -0.116995     0.12017    -0.91467
          1           0    -0.48186   0.0904127    0.990899  -0.0524048   -0.394045
```

2.2 Planning Scene Tutorial

The PlanningScene class provides the main interface that you will use for collision checking and constraint checking. In this tutorial, we will explore the C++ interface to this class.

2.2.1 Setup

The PlanningScene class can be easily setup and configured using a RobotModel or a URDF and SRDF. This is, however, not the recommended way to instantiate a PlanningScene. The PlanningSceneMonitor is the recommended method to create and maintain the current planning scene (and is discussed in detail in the next tutorial) using data from the robot's joints and the sensors on the robot. In this tutorial, we will instantiate a PlanningScene class directly, but this method of instantiation is only intended for illustration.

```
robot_model_loader::RobotModelLoader robot_model_loader("robot_description");
robot_model::RobotModelPtr kinematic_model = robot_model_loader.getModel();
planning_scene::PlanningScene planning_scene(kinematic_model);
```

2.2.2 Collision Checking

Self-collision checking

The first thing we will do is check whether the robot in its current state is in *self-collision*, i.e. whether the current configuration of the robot would result in the robot's parts hitting each other. To do this, we will construct a CollisionRequest object and a CollisionResult object and pass them into the collision checking function. Note that the result of whether the robot is in self-collision or not is contained within the result. Self collision checking uses an *unpadded* version of the robot, i.e. it directly uses the collision meshes provided in the URDF with no extra padding added on.

```
collision_detection::CollisionRequest collision_request;
collision_detection::CollisionResult collision_result;
planning_scene.checkSelfCollision(collision_request, collision_result);
ROS_INFO_STREAM("Test 1: Current state is " << (collision_result.collision ? "in" : "not in") << " se
```

Change the state

Now, let's change the current state of the robot. The planning scene maintains the current state internally. We can get a reference to it and change it and then check for collisions for the new robot configuration. Note in particular that we need to clear the collision_result before making a new collision checking request.

```
robot_state::RobotState& current_state = planning_scene.getCurrentStateNonConst();
current_state.setToRandomPositions();
collision_result.clear();
planning_scene.checkSelfCollision(collision_request, collision_result);
ROS_INFO_STREAM("Test 2: Current state is " << (collision_result.collision ? "in" : "not in") << " se
```

Checking for a group

Now, we will do collision checking only for the right_arm of the PR2, i.e. we will check whether there are any collisions between the right arm and other parts of the body of the robot. We can ask for this specifically by adding the group name "right_arm" to the collision request.

```
collision_request.group_name = "right_arm";
current_state.setToRandomPositions();
collision_result.clear();
planning_scene.checkSelfCollision(collision_request, collision_result);
ROS_INFO_STREAM("Test 3: Current state is " << (collision_result.collision ? "in" : "not in") << " se
```

Getting Contact Information

First, manually set the right arm to a position where we know internal (self) collisions do happen. Note that this state is now actually outside the joint limits of the PR2, which we can also check for directly.

```
std::vector<double> joint_values;
const robot_model::JointModelGroup* joint_model_group = current_state.getJointModelGroup("right_arm")
current_state.copyJointGroupPositions(joint_model_group, joint_values);
joint_values[0] = 1.57;  // hard-coded since we know collisions will happen here
current_state.setJointGroupPositions(joint_model_group, joint_values);
ROS_INFO_STREAM("Current state is " << (current_state.satisfiesBounds(joint_model_group) ? "valid" :
```

Now, we can get contact information for any collisions that might have happened at a given configuration of the right arm. We can ask for contact information by filling in the appropriate field in the collision request and specifying the maximum number of contacts to be returned as a large number.

```
collision_request.contacts = true;
collision_request.max_contacts = 1000;
```

```
collision_result.clear();
planning_scene.checkSelfCollision(collision_request, collision_result);
ROS_INFO_STREAM("Test 4: Current state is " << (collision_result.collision ? "in" : "not in") << " se
collision_detection::CollisionResult::ContactMap::const_iterator it;
for (it = collision_result.contacts.begin(); it != collision_result.contacts.end(); ++it)
{
  ROS_INFO("Contact between: %s and %s", it->first.first.c_str(), it->first.second.c_str());
}
```

Modifying the Allowed Collision Matrix

The AllowedCollisionMatrix (ACM) provides a mechanism to tell the collision world to ignore collisions between certain object: both parts of the robot and objects in the world. We can tell the collision checker to ignore all collisions between the links reported above, i.e. even though the links are actually in collision, the collision checker will ignore those collisions and return not in collision for this particular state of the robot.

Note also in this example how we are making copies of both the allowed collision matrix and the current state and passing them in to the collision checking function.

```
collision_detection::AllowedCollisionMatrix acm = planning_scene.getAllowedCollisionMatrix();
robot_state::RobotState copied_state = planning_scene.getCurrentState();

collision_detection::CollisionResult::ContactMap::const_iterator it2;
for (it2 = collision_result.contacts.begin(); it2 != collision_result.contacts.end(); ++it2)
{
  acm.setEntry(it2->first.first, it2->first.second, true);
}
collision_result.clear();
planning_scene.checkSelfCollision(collision_request, collision_result, copied_state, acm);
ROS_INFO_STREAM("Test 5: Current state is " << (collision_result.collision ? "in" : "not in") << " se
```

Full Collision Checking

While we have been checking for self-collisions, we can use the checkCollision functions instead which will check for both self-collisions and for collisions with the environment (which is currently empty). This is the set of collision checking functions that you will use most often in a planner. Note that collision checks with the environment will use the padded version of the robot. Padding helps in keeping the robot further away from obstacles in the environment.*/

```
collision_result.clear();
planning_scene.checkCollision(collision_request, collision_result, copied_state, acm);
ROS_INFO_STREAM("Test 6: Current state is " << (collision_result.collision ? "in" : "not in") << " se
```

2.2.3 Constraint Checking

The PlanningScene class also includes easy to use function calls for checking constraints. The constraints can be of two types: (a) constraints chosen from the KinematicConstraint set: i.e. JointConstraint, PositionConstraint, OrientationConstraint and VisibilityConstraint and (b) user defined constraints specified through a callback. We will first look at an example with a simple KinematicConstraint.

Checking Kinematic Constraints

We will first define a simple position and orientation constraint on the end-effector of the right_arm of the PR2 robot. Note the use of convenience functions for filling up the constraints (these functions are found in the utils.h file from the kinematic_constraints directory in moveit_core).

```
std::string end_effector_name = joint_model_group->getLinkModelNames().back();

geometry_msgs::PoseStamped desired_pose;
desired_pose.pose.orientation.w = 1.0;
desired_pose.pose.position.x = 0.75;
desired_pose.pose.position.y = -0.185;
desired_pose.pose.position.z = 1.3;
desired_pose.header.frame_id = "base_footprint";
moveit_msgs::Constraints goal_constraint =
    kinematic_constraints::constructGoalConstraints(end_effector_name, desired_pose);
```

Now, we can check a state against this constraint using the isStateConstrained functions in the PlanningScene class.

```
copied_state.setToRandomPositions();
copied_state.update();
bool constrained = planning_scene.isStateConstrained(copied_state, goal_constraint);
ROS_INFO_STREAM("Test 7: Random state is " << (constrained ? "constrained" : "not constrained"));
```

There's a more efficient way of checking constraints (when you want to check the same constraint over and over again, e.g. inside a planner). We first construct a KinematicConstraintSet which pre-processes the ROS Constraints messages and sets it up for quick processing.

```
kinematic_constraints::KinematicConstraintSet kinematic_constraint_set(kinematic_model);
kinematic_constraint_set.add(goal_constraint, planning_scene.getTransforms());
bool constrained_2 = planning_scene.isStateConstrained(copied_state, kinematic_constraint_set);
ROS_INFO_STREAM("Test 8: Random state is " << (constrained_2 ? "constrained" : "not constrained"));
```

There's a direct way to do this using the KinematicConstraintSet class.

```
kinematic_constraints::ConstraintEvaluationResult constraint_eval_result =
    kinematic_constraint_set.decide(copied_state);
ROS_INFO_STREAM("Test 9: Random state is " << (constraint_eval_result.satisfied ? "constrained" : "no
```

User-defined constraints

User defined constraints can also be specified to the PlanningScene class. This is done by specifying a callback using the setStateFeasibilityPredicate function. Here's a simple example of a user-defined callback that checks whether the "r_shoulder_pan" of the PR2 robot is at a positive or negative angle:

```
bool userCallback(const robot_state::RobotState& kinematic_state, bool verbose)
{
  const double* joint_values = kinematic_state.getJointPositions("r_shoulder_pan_joint");
  return (joint_values[0] > 0.0);
}
```

Now, whenever isStateFeasible is called, this user-defined callback will be called.

```
planning_scene.setStateFeasibilityPredicate(userCallback);
bool state_feasible = planning_scene.isStateFeasible(copied_state);
ROS_INFO_STREAM("Test 10: Random state is " << (state_feasible ? "feasible" : "not feasible"));
```

Whenever isStateValid is called, three checks are conducted: (a) collision checking (b) constraint checking and (c) feasibility checking using the user-defined callback.

```
bool state_valid = planning_scene.isStateValid(copied_state, kinematic_constraint_set, "right_arm");
ROS_INFO_STREAM("Test 10: Random state is " << (state_valid ? "valid" : "not valid"));
```

Note that all the planners available through MoveIt! and OMPL will currently perform collision checking, constraint checking and feasibility checking using user-defined callbacks.

2.2.4 The entire code

The entire code can be seen here in the moveit_pr2 github project.

2.2.5 Compiling the code

Follow the instructions for compiling code from source.

2.2.6 The launch file

The entire launch file is here on github. All the code in this tutorial can be compiled and run from the moveit_tutorials package that you have as part of your MoveIt! setup.

2.2.7 Running the code

Roslaunch the launch file to run the code directly from moveit_tutorials:

```
roslaunch moveit_tutorials planning_scene_tutorial.launch
```

2.2.8 Expected Output

The output should look something like this, though we are using random joint values so some things may be different:

```
[ INFO] [1385487628.853237681]: Test 1: Current state is not in self collision
[ INFO] [1385487628.857680844]: Test 2: Current state is in self collision
[ INFO] [1385487628.861798756]: Test 3: Current state is not in self collision
[ INFO] [1385487628.861876838]: Current state is not valid
[ INFO] [1385487628.866177315]: Test 4: Current state is in self collision
[ INFO] [1385487628.866228020]: Contact between: l_shoulder_pan_link and r_forearm_link
[ INFO] [1385487628.866259030]: Contact between: l_shoulder_pan_link and r_shoulder_lift_link
[ INFO] [1385487628.866305963]: Contact between: l_shoulder_pan_link and r_shoulder_pan_link
[ INFO] [1385487628.866331036]: Contact between: l_shoulder_pan_link and r_upper_arm_link
[ INFO] [1385487628.866358135]: Contact between: l_shoulder_pan_link and r_upper_arm_roll_link
[ INFO] [1385487628.870629418]: Test 5: Current state is not in self collision
[ INFO] [1385487628.877406467]: Test 6: Current state is not in self collision
[ INFO] [1385487628.879610797]: Test 7: Random state is not constrained
[ INFO] [1385487628.880027331]: Test 8: Random state is not constrained
[ INFO] [1385487628.880315077]: Test 9: Random state is not constrained
[ INFO] [1385487628.880377445]: Test 10: Random state is feasible
[ INFO] [1385487628.887157707]: Test 10: Random state is not valid
```

2.3 ROS API Planning Scene Tutorial

In this tutorial, we will examine the use of planning scene diffs to perform two operations:

- Adding and removing objects into the world
- Attaching and detaching objects to the robot

2.3.1 ROS API

The ROS API to the planning scene publisher is through a topic interface using "diffs". A planning scene diff is the difference between the current planning scene (maintained by the move_group node) and the new planning scene desired by the user.

2.3.2 Advertise the required topic

Note that this topic may need to be remapped in the launch file

```
ros::Publisher planning_scene_diff_publisher = node_handle.advertise<moveit_msgs::PlanningScene>("pla
while (planning_scene_diff_publisher.getNumSubscribers() < 1)
{
  ros::WallDuration sleep_t(0.5);
  sleep_t.sleep();
}
```

2.3.3 Define the attached object message

We will use this message to add or subtract the object from the world and to attach the object to the robot

```
moveit_msgs::AttachedCollisionObject attached_object;
attached_object.link_name = "r_wrist_roll_link";
/* The header must contain a valid TF frame*/
attached_object.object.header.frame_id = "r_wrist_roll_link";
/* The id of the object */
attached_object.object.id = "box";

/* A default pose */
geometry_msgs::Pose pose;
pose.orientation.w = 1.0;

/* Define a box to be attached */
shape_msgs::SolidPrimitive primitive;
primitive.type = primitive.BOX;
primitive.dimensions.resize(3);
primitive.dimensions[0] = 0.1;
primitive.dimensions[1] = 0.1;
primitive.dimensions[2] = 0.1;

attached_object.object.primitives.push_back(primitive);
attached_object.object.primitive_poses.push_back(pose);
```

Note that attaching an object to the robot requires the corresponding operation to be specified as an ADD operation

```
attached_object.object.operation = attached_object.object.ADD;
```

2.3.4 Add an object into the environment

Add the object into the environment by adding it to the set of collision objects in the "world" part of the planning scene. Note that we are using only the "object" field of the attached_object message here.

```
ROS_INFO("Adding the object into the world at the location of the right wrist.");
moveit_msgs::PlanningScene planning_scene;
planning_scene.world.collision_objects.push_back(attached_object.object);
planning_scene.is_diff = true;
planning_scene_diff_publisher.publish(planning_scene);
sleep_time.sleep();
```

2.3.5 Interlude: Synchronous vs Asynchronous updates

There are two separate mechanisms available to interact with the move_group node using diffs:

- Send a diff via a rosservice call and block until the diff is applied (synchronous update)
- Send a diff via a topic, continue even though the diff might not be applied yet (asynchronous update)

While most of this tutorial uses the latter mechanism (given the long sleeps inserted for visualization purposes asynchronous updates do not pose a problem), it would is perfectly justified to replace the planning_scene_diff_publisher by the following service client:

```
ros::ServiceClient planning_scene_diff_client =
    node_handle.serviceClient<moveit_msgs::ApplyPlanningScene>("apply_planning_scene");
planning_scene_diff_client.waitForExistence();
```

and send the diffs to the planning scene via a service call:

```
moveit_msgs::ApplyPlanningScene srv;
srv.request.scene = planning_scene;
planning_scene_diff_client.call(srv);
```

Note that this does not continue until we are sure the diff has been applied.

2.3.6 Attach an object to the robot

When the robot picks up an object from the environment, we need to "attach" the object to the robot so that any component dealing with the robot model knows to account for the attached object, e.g. for collision checking.

Attaching an object requires two operations

- Removing the original object from the environment
- Attaching the object to the robot

```
/* First, define the REMOVE object message*/
moveit_msgs::CollisionObject remove_object;
remove_object.id = "box";
remove_object.header.frame_id = "odom_combined";
remove_object.operation = remove_object.REMOVE;
```

Note how we make sure that the diff message contains no other attached objects or collisions objects by clearing those fields first.

```
/* Carry out the REMOVE + ATTACH operation */
ROS_INFO("Attaching the object to the right wrist and removing it from the world.");
planning_scene.world.collision_objects.clear();
planning_scene.world.collision_objects.push_back(remove_object);
planning_scene.robot_state.attached_collision_objects.push_back(attached_object);
planning_scene_diff_publisher.publish(planning_scene);

sleep_time.sleep();
```

2.3.7 Detach an object from the robot

Detaching an object from the robot requires two operations

- Detaching the object from the robot
- Re-introducing the object into the environment

```
/* First, define the DETACH object message*/
moveit_msgs::AttachedCollisionObject detach_object;
detach_object.object.id = "box";
detach_object.link_name = "r_wrist_roll_link";
detach_object.object.operation = attached_object.object.REMOVE;
```

Note how we make sure that the diff message contains no other attached objects or collisions objects by clearing those fields first.

```
/* Carry out the DETACH + ADD operation */
ROS_INFO("Detaching the object from the robot and returning it to the world.");
planning_scene.robot_state.attached_collision_objects.clear();
planning_scene.robot_state.attached_collision_objects.push_back(detach_object);
planning_scene.world.collision_objects.clear();
planning_scene.world.collision_objects.push_back(attached_object.object);
planning_scene_diff_publisher.publish(planning_scene);

sleep_time.sleep();
```

2.3.8 Remove the object from the collision world

Removing the object from the collision world just requires using the remove object message defined earlier. Note, also how we make sure that the diff message contains no other attached objects or collisions objects by clearing those fields first.

```
ROS_INFO("Removing the object from the world.");
planning_scene.robot_state.attached_collision_objects.clear();
planning_scene.world.collision_objects.clear();
planning_scene.world.collision_objects.push_back(remove_object);
planning_scene_diff_publisher.publish(planning_scene);
```

2.3.9 The entire code

The entire code can be seen here in the moveit_pr2 github project.

2.3.10 Compiling the code

Follow the instructions for compiling code from source.

2.3.11 The launch file

The entire launch file is here on github. All the code in this tutorial can be compiled and run from the moveit_tutorials package that you have as part of your MoveIt! setup.

2.3.12 Running the code

Roslaunch the launch file to run the code directly from moveit_tutorials:

```
roslaunch moveit_tutorials planning_scene_ros_api_tutorial.launch
```

2.3.13 Expected Output

In rviz, you should be able to see the following:

- Object appear in the planning scene
- Object gets attached to the robot
- Object gets detached from the robot
- Object is removed from the planning scene

2.4 Motion Planners Tutorial

In MoveIt!, the motion planners are loaded using a plugin infrastructure. This allows MoveIt! to load motion planners at runtime. In this example, we will run through the C++ code required to do this.

2.4.1 Start

Setting up to start using a planner is pretty easy. Planners are setup as plugins in MoveIt! and you can use the ROS pluginlib interface to load any planner that you want to use. Before we can load the planner, we need two objects, a RobotModel and a PlanningScene. We will start by instantiating a RobotModelLoader object, which will look up the robot description on the ROS parameter server and construct a RobotModel for us to use.

```
robot_model_loader::RobotModelLoader robot_model_loader("robot_description");
robot_model::RobotModelPtr robot_model = robot_model_loader.getModel();
```

Using the RobotModel, we can construct a PlanningScene that maintains the state of the world (including the robot).

```
planning_scene::PlanningScenePtr planning_scene(new planning_scene::PlanningScene(robot_model));
```

We will now construct a loader to load a planner, by name. Note that we are using the ROS pluginlib library here.

```
boost::scoped_ptr<pluginlib::ClassLoader<planning_interface::PlannerManager>> planner_plugin_loader;
planning_interface::PlannerManagerPtr planner_instance;
std::string planner_plugin_name;
```

We will get the name of planning plugin we want to load from the ROS param server, and then load the planner making sure to catch all exceptions.

```
if (!node_handle.getParam("planning_plugin", planner_plugin_name))
  ROS_FATAL_STREAM("Could not find planner plugin name");
try
{
  planner_plugin_loader.reset(new pluginlib::ClassLoader<planning_interface::PlannerManager>(
      "moveit_core", "planning_interface::PlannerManager"));
}
catch (pluginlib::PluginlibException& ex)
{
  ROS_FATAL_STREAM("Exception while creating planning plugin loader " << ex.what());
}
try
{
  planner_instance.reset(planner_plugin_loader->createUnmanagedInstance(planner_plugin_name));
  if (!planner_instance->initialize(robot_model, node_handle.getNamespace()))
    ROS_FATAL_STREAM("Could not initialize planner instance");
  ROS_INFO_STREAM("Using planning interface '" << planner_instance->getDescription() << "'");
}
catch (pluginlib::PluginlibException& ex)
{
  const std::vector<std::string>& classes = planner_plugin_loader->getDeclaredClasses();
  std::stringstream ss;
  for (std::size_t i = 0; i < classes.size(); ++i)
    ss << classes[i] << " ";
  ROS_ERROR_STREAM("Exception while loading planner '" << planner_plugin_name << "': " << ex.what() <
                                                       << "Available plugins: " << ss.str());
}

/* Sleep a little to allow time to startup rviz, etc. */
ros::WallDuration sleep_time(15.0);
sleep_time.sleep();
```

2.4.2 Pose Goal

We will now create a motion plan request for the right arm of the PR2 specifying the desired pose of the end-effector as input.

```
planning_interface::MotionPlanRequest req;
planning_interface::MotionPlanResponse res;
geometry_msgs::PoseStamped pose;
pose.header.frame_id = "torso_lift_link";
pose.pose.position.x = 0.75;
pose.pose.position.y = 0.0;
pose.pose.position.z = 0.0;
pose.pose.orientation.w = 1.0;
```

A tolerance of 0.01 m is specified in position and 0.01 radians in orientation

```
std::vector<double> tolerance_pose(3, 0.01);
std::vector<double> tolerance_angle(3, 0.01);
```

We will create the request as a constraint using a helper function available from the kinematic_constraints package.

```
req.group_name = "right_arm";
moveit_msgs::Constraints pose_goal =
    kinematic_constraints::constructGoalConstraints("r_wrist_roll_link", pose, tolerance_pose, tolera
req.goal_constraints.push_back(pose_goal);
```

We now construct a planning context that encapsulate the scene, the request and the response. We call the planner using this planning context

```
planning_interface::PlanningContextPtr context =
    planner_instance->getPlanningContext(planning_scene, req, res.error_code_);
context->solve(res);
if (res.error_code_.val != res.error_code_.SUCCESS)
{
  ROS_ERROR("Could not compute plan successfully");
  return 0;
}
```

2.4.3 Visualize the result

```
ros::Publisher display_publisher =
    node_handle.advertise<moveit_msgs::DisplayTrajectory>("/move_group/display_planned_path", 1, true
moveit_msgs::DisplayTrajectory display_trajectory;

/* Visualize the trajectory */
ROS_INFO("Visualizing the trajectory");
moveit_msgs::MotionPlanResponse response;
res.getMessage(response);

display_trajectory.trajectory_start = response.trajectory_start;
display_trajectory.trajectory.push_back(response.trajectory);
display_publisher.publish(display_trajectory);

sleep_time.sleep();
```

2.4.4 Joint Space Goals

```
/* First, set the state in the planning scene to the final state of the last plan */
robot_state::RobotState& robot_state = planning_scene->getCurrentStateNonConst();
planning_scene->setCurrentState(response.trajectory_start);
const robot_state::JointModelGroup* joint_model_group = robot_state.getJointModelGroup("right_arm");
robot_state.setJointGroupPositions(joint_model_group, response.trajectory.joint_trajectory.points.bac
```

Now, setup a joint space goal

```
robot_state::RobotState goal_state(robot_model);
std::vector<double> joint_values(7, 0.0);
joint_values[0] = -2.0;
joint_values[3] = -0.2;
joint_values[5] = -0.15;
goal_state.setJointGroupPositions(joint_model_group, joint_values);
moveit_msgs::Constraints joint_goal = kinematic_constraints::constructGoalConstraints(goal_state, joi
req.goal_constraints.clear();
req.goal_constraints.push_back(joint_goal);
```

Call the planner and visualize the trajectory

```
/* Re-construct the planning context */
context = planner_instance->getPlanningContext(planning_scene, req, res.error_code_);
/* Call the Planner */
context->solve(res);
/* Check that the planning was successful */
if (res.error_code_.val != res.error_code_.SUCCESS)
{
  ROS_ERROR("Could not compute plan successfully");
  return 0;
}
/* Visualize the trajectory */
ROS_INFO("Visualizing the trajectory");
res.getMessage(response);
display_trajectory.trajectory_start = response.trajectory_start;
display_trajectory.trajectory.push_back(response.trajectory);

/* Now you should see two planned trajectories in series*/
display_publisher.publish(display_trajectory);

/* We will add more goals. But first, set the state in the planning
   scene to the final state of the last plan */
robot_state.setJointGroupPositions(joint_model_group, response.trajectory.joint_trajectory.points.bac

/* Now, we go back to the first goal*/
req.goal_constraints.clear();
req.goal_constraints.push_back(pose_goal);
context = planner_instance->getPlanningContext(planning_scene, req, res.error_code_);
context->solve(res);
res.getMessage(response);
display_trajectory.trajectory.push_back(response.trajectory);
display_publisher.publish(display_trajectory);
```

2.4.5 Adding Path Constraints

Let's add a new pose goal again. This time we will also add a path constraint to the motion.

```
/* Let's create a new pose goal */
pose.pose.position.x = 0.65;
pose.pose.position.y = -0.2;
pose.pose.position.z = -0.1;
moveit_msgs::Constraints pose_goal_2 =
    kinematic_constraints::constructGoalConstraints("r_wrist_roll_link", pose, tolerance_pose, tolera
/* First, set the state in the planning scene to the final state of the last plan */
robot_state.setJointGroupPositions(joint_model_group, response.trajectory.joint_trajectory.points.bac
/* Now, let's try to move to this new pose goal*/
req.goal_constraints.clear();
req.goal_constraints.push_back(pose_goal_2);

/* But, let's impose a path constraint on the motion.
   Here, we are asking for the end-effector to stay level*/
geometry_msgs::QuaternionStamped quaternion;
quaternion.header.frame_id = "torso_lift_link";
quaternion.quaternion.w = 1.0;
req.path_constraints = kinematic_constraints::constructGoalConstraints("r_wrist_roll_link", quaternio
```

Imposing path constraints requires the planner to reason in the space of possible positions of the end-effector (the workspace of the robot) because of this, we need to specify a bound for the allowed planning volume as well; Note: a default bound is automatically filled by the WorkspaceBounds request adapter (part of the OMPL pipeline, but that is not being used in this example). We use a bound that definitely includes the reachable space for the arm. This is fine because sampling is not done in this volume when planning for the arm; the bounds are only used to determine if the sampled configurations are valid.

```
req.workspace_parameters.min_corner.x = req.workspace_parameters.min_corner.y =
    req.workspace_parameters.min_corner.z = -2.0;
req.workspace_parameters.max_corner.x = req.workspace_parameters.max_corner.y =
    req.workspace_parameters.max_corner.z = 2.0;
```

Call the planner and visualize all the plans created so far.

```
context = planner_instance->getPlanningContext(planning_scene, req, res.error_code_);
context->solve(res);
res.getMessage(response);
display_trajectory.trajectory.push_back(response.trajectory);
```

Now you should see four planned trajectories in series

```
display_publisher.publish(display_trajectory);
```

2.4.6 The entire code

The entire code can be seen here in the moveit_pr2 github project.

2.4.7 Compiling the code

Follow the instructions for compiling code from source.

2.4.8 The launch file

The entire launch file is here on github. All the code in this tutorial can be compiled and run from the moveit_tutorials package that you have as part of your MoveIt! setup.

2.4.9 Running the code

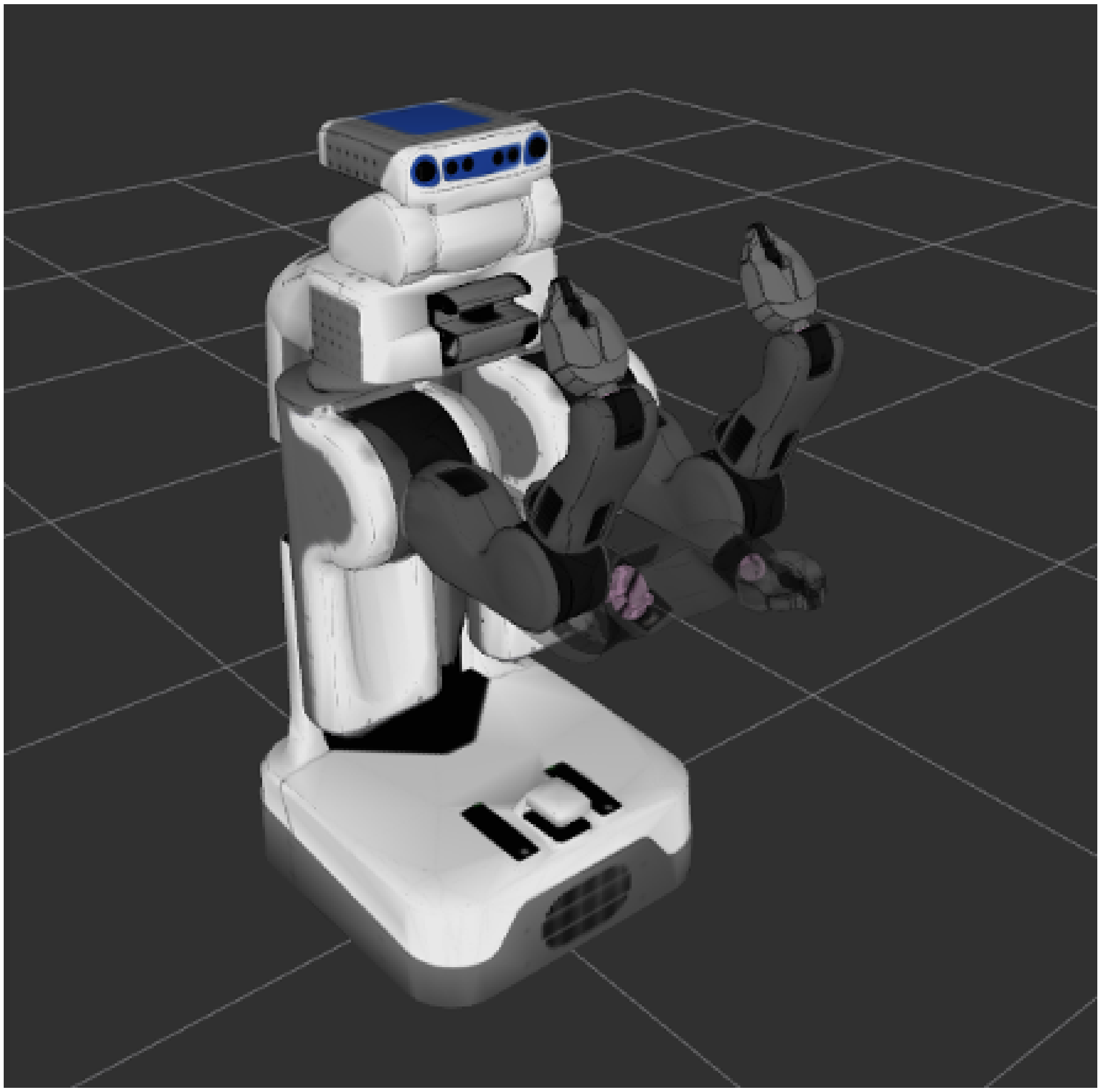
Roslaunch the launch file to run the code directly from moveit_tutorials:

```
roslaunch moveit_tutorials motion_planning_api_tutorial.launch
```

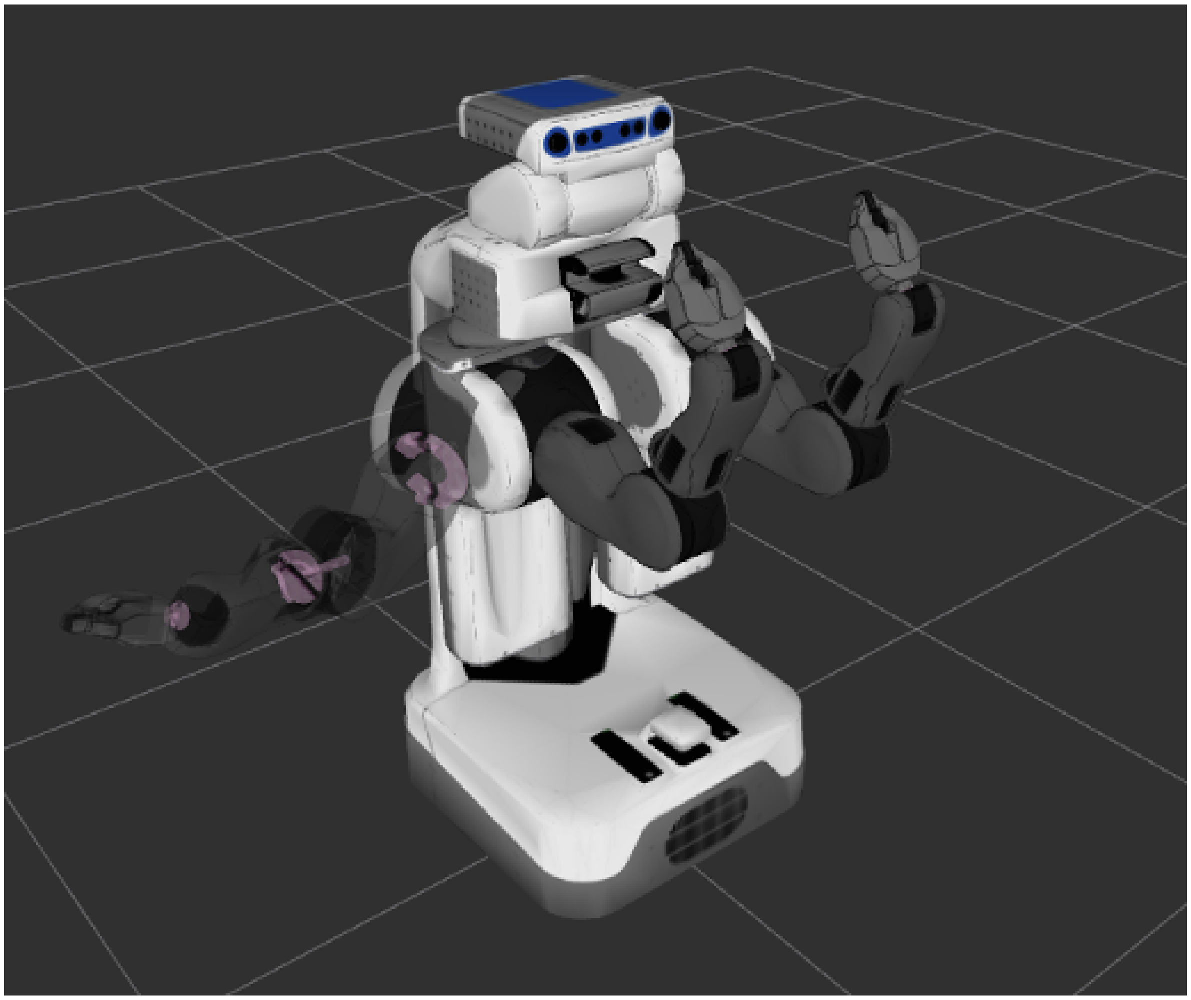
2.4.10 Expected Output

In Rviz, we should be able to see four trajectories being replayed eventually:

1. The robot moves its right arm to the pose goal in front of it,



2. The robot moves its right arm to the joint goal to the side,
3. The robot moves its right arm back to the original pose goal in front of it,
4. The robot moves its right arm back to a new pose goal while maintaining the end-effector level.



2.5 Motion Planning Pipeline Tutorial

In MoveIt!, the motion planners are setup to plan paths. However, there are often times when we may want to pre-process the motion planning request or post-process the planned path (e.g. for time parameterization). In such cases, we use the planning pipeline which chains a motion planner with pre-processing and post-processing stages. The pre and post-processing stages, called planning request adapters, can be configured by name from the ROS parameter server. In this tutorial, we will run you through the C++ code to instantiate and call such a planning pipeline.

2.5.1 Start

Setting up to start using a planning pipeline is pretty easy. Before we can load the planner, we need two objects, a RobotModel and a PlanningScene. We will start by instantiating a RobotModelLoader object, which will look up the robot description on the ROS parameter server and construct a RobotModel for us to use.

```
robot_model_loader::RobotModelLoader robot_model_loader("robot_description");
robot_model::RobotModelPtr robot_model = robot_model_loader.getModel();
```

Using the RobotModel, we can construct a PlanningScene that maintains the state of the world (including the robot).

```
planning_scene::PlanningScenePtr planning_scene(new planning_scene::PlanningScene(robot_model));
```

We can now setup the PlanningPipeline object, which will use the ROS param server to determine the set of request adapters and the planning plugin to use

```
planning_pipeline::PlanningPipelinePtr planning_pipeline(
    new planning_pipeline::PlanningPipeline(robot_model, node_handle, "planning_plugin", "request_ada

/* Sleep a little to allow time to startup rviz, etc. */
ros::WallDuration sleep_time(20.0);
sleep_time.sleep();
```

2.5.2 Pose Goal

We will now create a motion plan request for the right arm of the PR2 specifying the desired pose of the end-effector as input.

```
planning_interface::MotionPlanRequest req;
planning_interface::MotionPlanResponse res;
geometry_msgs::PoseStamped pose;
pose.header.frame_id = "torso_lift_link";
pose.pose.position.x = 0.75;
pose.pose.position.y = 0.0;
pose.pose.position.z = 0.0;
pose.pose.orientation.w = 1.0;
```

A tolerance of 0.01 m is specified in position and 0.01 radians in orientation

```
std::vector<double> tolerance_pose(3, 0.01);
std::vector<double> tolerance_angle(3, 0.01);
```

We will create the request as a constraint using a helper function available from the kinematic_constraints package.

```
req.group_name = "right_arm";
moveit_msgs::Constraints pose_goal =
    kinematic_constraints::constructGoalConstraints("r_wrist_roll_link", pose, tolerance_pose, tolera
req.goal_constraints.push_back(pose_goal);
```

Now, call the pipeline and check whether planning was successful.

```
planning_pipeline->generatePlan(planning_scene, req, res);
/* Check that the planning was successful */
if (res.error_code_.val != res.error_code_.SUCCESS)
{
  ROS_ERROR("Could not compute plan successfully");
  return 0;
}
```

2.5.3 Visualize the result

```
ros::Publisher display_publisher =
    node_handle.advertise<moveit_msgs::DisplayTrajectory>("/move_group/display_planned_path", 1, true
moveit_msgs::DisplayTrajectory display_trajectory;

/* Visualize the trajectory */
ROS_INFO("Visualizing the trajectory");
```

```
moveit_msgs::MotionPlanResponse response;
res.getMessage(response);

display_trajectory.trajectory_start = response.trajectory_start;
display_trajectory.trajectory.push_back(response.trajectory);
display_publisher.publish(display_trajectory);

sleep_time.sleep();
```

2.5.4 Joint Space Goals

```
/* First, set the state in the planning scene to the final state of the last plan */
robot_state::RobotState& robot_state = planning_scene->getCurrentStateNonConst();
planning_scene->setCurrentState(response.trajectory_start);
const robot_model::JointModelGroup* joint_model_group = robot_state.getJointModelGroup("right_arm");
robot_state.setJointGroupPositions(joint_model_group, response.trajectory.joint_trajectory.points.bac
```

Now, setup a joint space goal

```
robot_state::RobotState goal_state(robot_model);
std::vector<double> joint_values(7, 0.0);
joint_values[0] = -2.0;
joint_values[3] = -0.2;
joint_values[5] = -0.15;
goal_state.setJointGroupPositions(joint_model_group, joint_values);
moveit_msgs::Constraints joint_goal = kinematic_constraints::constructGoalConstraints(goal_state, joi

req.goal_constraints.clear();
req.goal_constraints.push_back(joint_goal);
```

Call the pipeline and visualize the trajectory

```
planning_pipeline->generatePlan(planning_scene, req, res);
/* Check that the planning was successful */
if (res.error_code_.val != res.error_code_.SUCCESS)
{
  ROS_ERROR("Could not compute plan successfully");
  return 0;
}
/* Visualize the trajectory */
ROS_INFO("Visualizing the trajectory");
res.getMessage(response);
display_trajectory.trajectory_start = response.trajectory_start;
display_trajectory.trajectory.push_back(response.trajectory);
```

Now you should see two planned trajectories in series

```
display_publisher.publish(display_trajectory);
sleep_time.sleep();
```

2.5.5 Using a Planning Request Adapter

A planning request adapter allows us to specify a series of operations that should happen either before planning takes place or after the planning has been done on the resultant path

```
```

First, let's purposefully set the initial state to be outside the joint limits and let the planning request adapter deal with it

```
/* First, set the state in the planning scene to the final state of the last plan */
robot_state = planning_scene->getCurrentStateNonConst();
planning_scene->setCurrentState(response.trajectory_start);
robot_state.setJointGroupPositions(joint_model_group, response.trajectory.joint_trajectory.points.bac
```

Now, set one of the joints slightly outside its upper limit

```
const robot_model::JointModel* joint_model = joint_model_group->getJointModel("r_shoulder_pan_joint")
const robot_model::JointModel::Bounds& joint_bounds = joint_model->getVariableBounds();
std::vector<double> tmp_values(1, 0.0);
tmp_values[0] = joint_bounds[0].min_position_ - 0.01;
robot_state.setJointPositions(joint_model, tmp_values);
req.goal_constraints.clear();
req.goal_constraints.push_back(pose_goal);
```

Call the planner again and visualize the trajectories

```
  planning_pipeline->generatePlan(planning_scene, req, res);
  if (res.error_code_.val != res.error_code_.SUCCESS)
  {
    ROS_ERROR("Could not compute plan successfully");
    return 0;
  }
  /* Visualize the trajectory */
  ROS_INFO("Visualizing the trajectory");
  res.getMessage(response);
  display_trajectory.trajectory_start = response.trajectory_start;
  display_trajectory.trajectory.push_back(response.trajectory);
  /* Now you should see three planned trajectories in series*/
  display_publisher.publish(display_trajectory);

  sleep_time.sleep();
  ROS_INFO("Done");
  return 0;
}
```

2.5.6 The entire code

The entire code can be seen here in the moveit_pr2 github project.

2.5.7 Compiling the code

Follow the instructions for compiling code from source.

2.5.8 The launch file

The entire launch file is here on github. All the code in this tutorial can be compiled and run from the moveit_tutorials package that you have as part of your MoveIt! setup.

2.5.9 Running the code

Roslaunch the launch file to run the code directly from moveit_tutorials:

```
roslaunch moveit_tutorials motion_planning_api_tutorial.launch
```

2.5.10 Expected Output

In Rviz, we should be able to see three trajectories being replayed eventually:

1. The robot moves its right arm to the pose goal in front of it,
2. The robot moves its right arm to the joint goal to the side,
3. The robot moves its right arm back to the original pose goal in front of it,

2.6 Controller Manager Tutorials

MoveIt! comes with a series of fake trajectory controllers to be used in simulation. For example, the `demo.launch` generated by MoveIt's setup assistant, employs fake controllers for nice visualization in rviz.

For configuration, edit the file `config/fake_controllers.yaml`, and adjust the desired controller type. The following controllers are available:

- **interpolate**: perform smooth interpolation between via points - the default for visualization
- **via points**: traverse via points, w/o interpolation in between - useful for visual debugging
- **last point**: warp directly to the last point of the trajectory - fastest method for offline benchmarking

2.6.1 YAML file examples

```
rate: 10 (Hz, used for interpolation controller)
controller_list:
  - name: fake_arm_controller
    type: interpolate | via points | last point
    joints:
      - joint_1
      - joint_2
      - joint_3
      - joint_4
      - joint_5
      - joint_6
  - name: fake_gripper_controller
    joints:
      []
```

In order to load an initial pose, one can have a list of (group, pose) pairs as follows:

```
initial:
  - group: arm
    pose:  home
```

2.7 Joystick Control Teleoperation

2.7.1 Run

Startup regular MoveIt! planning node with Rviz (for example demo.launch)

Make sure you have the dependencies installed:

```
sudo apt-get install ros-indigo-joy
```

In the Motion Planning plugin of Rviz, enable "Allow External Comm." checkbox in the "Planning" tab. Enable the 'Query Goal State' robot display in the MoveIt! Motion Planning Plugins's 'Planning Request' section.

Now launch the joystick control launch file specific to your robot. If you are missing this file, first re-run the MoveIt! Setup Assistant using the latest version of the Setup Assistant:

```
roslaunch YOURROBOT_moveit_config joystick_control.launch
```

The script defaults to using `/dev/input/js0` for your game controller port. To customize, you can also use, for example:

```
roslaunch YOURROBOT_moveit_config joystick_control.launch dev:=/dev/input/js1
```

This script can read three types of joy sticks:

1. XBox360 Controller via USB
2. PS3 Controller via USB
3. PS3 Controller via Bluetooth (Please use ps3joy package at http://wiki.ros.org/ps3joy)

2.7.2 Joystick Command Mappings

Command	PS3 Controller	Xbox Controller
+-x/y	left analog stick	left analog stick
+-z	L2/R2	LT/RT
+-yaw	L1/R1	LB/RB
+-roll	left/right	left/right
+-pitch	up/down	up/down
change planning group	select/start	Y/A
change end effector	triangle/cross	back/start
plan	square	X
execute	circle	B

2.7.3 Debugging

Add "Pose" to rviz Displays and subscribe to `/joy_pose` in order to see the output from joystick.

Note that only planning groups that have IK solvers for all their End Effector parent groups will work.

2.8 Creating Custom Constraint Samplers

2.8.1 Overview

Some planning problems require more complex or custom constraint samplers for more difficult planning problems. This document explains how to creat a custom motion planning constraint sampler for use with MoveIt!.

2.8.2 Pre-requisites

2.8.3 Creating a constraint sampler

- Create a ROBOT_moveit_plugins package and within that a subfolder for your YOURROBOT_constraint_sampler plugin. Modify the template provided by hrp2jsk_moveit_plugins/hrp2jsk_moveit_constraint_sampler_plugin
- In your ROBOT_moveit_config/launch/move_group.launch file, within the <node name="move_group">, add the parameter:

  <param name="constraint_samplers" value="YOURROBOT_moveit_constraint_sampler/YOURROBOTConstraintSampler
- Now when you launch move_group, it should default to your new constraint sampler.

2.9 Benchmarking Tutorial

Note: This is the new benchmarking method only available in ROS Kinetic, onward. Documentation for the previous benchmarking tutorial for Jade and earlier can be found on the deprecated wiki

Note: To use this benchmarking method, you will need to download and install the ROS Warehouse plugin. Currently this is not available from debians and requires a source install for at least some aspects. For source instructions, see this page

The benchmarking package provides methods to benchmark motion planning algorithms and aggregate/plot statistics using the OMPL Planner Arena. The example below demonstrates how the benchmarking can be run for a Fanuc M-10iA.

2.9.1 Example

An example is provided in the `examples` folder. The launch file requires a MoveIt! configuration package for the Fanuc M-10iA available from here.

To run:

1. Launch the Fanuc M-10iA `demo.launch`:

```
roslaunch moveit_resources demo.launch db:=true
```

2. Within the *Motion Planning* RViz plugin, connect to the database by pressing the *Connect* button in the *Context* tab.
3. Save a scene on the *Stored Scenes* tab and name it `Kitchen1` by double clicking the scene in the list.

4. Move the start and goal states of the Fanuc M-10iA by using the interactive markers.
5. Save an associated query for the `Kitchen1` scene and name the query `Pick1`.
6. Also save a start state for the robot on the *Stored States* tab and name it `Start1`.
7. The config file `moveit_ros/benchmarks/examples/demo1.yaml` refers to the scenes, queries and start states used for benchmarking. Modify them appropriately.
8. Bring down your previous launch file (`ctrl+c`).
9. Change the location `output_directory` to export the benchmarked files:

```
    rosed moveit_ros_benchmarks demo_fanuc.launch
```

10. Run the benchmarks:

```
    roslaunch moveit_ros_benchmarks demo_fanuc.launch
```

2.9.2 Viewing Results

The benchmarks are executed and many interesting parameters are aggregated and written to a logfile. A script (`moveit_benchmark_statistics.py`) is supplied to parse this data and plot the statistics.

Run:

```
rosrun moveit_ros_benchmarks moveit_benchmark_statistics.py <path_of_logfile>
```

To generate a PDF of plots:

```
python -p <plot_filename> moveit_benchmark_statistics.py <path_of_logfile>
```

Alternatively, upload the database file generated by `moveit_benchmark_statistics.py` to plannerarena.org and interactively visualize the results.

2.9.3 Parameters of the BenchmarkOptions Class

This class reads in parameters and options for the benchmarks to run from the ROS parameter server. The format of the parameters is assumed to be in the following form:

```
benchmark_config:

  warehouse:
      host: [hostname/IP address of ROS Warehouse node]                    # Default localhos
      port: [port number of ROS Warehouse node]                            # Default 33829
      scene_name: [Name of the planning scene to use for benchmarks]       # REQUIRED

  parameters:
      runs: [Number of runs for each planning algorithm on each request]   # Default 10
      group: [The name of the group to plan]                               # REQUIRED
      timeout: [The maximum time for a single run; seconds]                # Default 10.0
      output_directory: [The directory to write the output to]             # Default is curren

      start_states: [Regex for the stored start states in the warehouse to try]   # Default ""
      path_constraints: [Regex for the path constraints to benchmark]             # Default ""

       queries: [Regex for the motion plan queries in the warehouse to try]        # Default .*
       goal_constraints: [Regex for the goal constraints to benchmark]             # Default ""
       trajectory_constraints: [Regex for the trajectory constraints to benchmark] # Default ""
```

```
    workspace: [Bounds of the workspace the robot plans in.  This is an AABB]   # Optional
        frame_id: [The frame the workspace parameters are specified in]
        min_corner: [Coordinates of the minimum corner of the AABB]
            x: [x-value]
            y: [y-value]
            z: [z-value]
        max_corner: [Coordinates of the maximum corner of the AABB]
            x: [x-value]
            y: [y-value]
            z: [z-value]

planners:
    - plugin: [The name of the planning plugin the planners are in]             # REQUIRED
      planners:                                                                 # REQUIRED
        - A list of planners
        - from the plugin above
        - to benchmark the
        - queries in.
    - plugin: ...
        - ...
```

2.9.4 Parameters of the BenchmarkExecutor Class

This class creates a set of `MotionPlanRequests` that respect the parameters given in the supplied instance of `BenchmarkOptions` and then executes the requests on each of the planners specified. From the `BenchmarkOptions`, queries, `goal_constraints`, and `trajectory_constraints` are treated as separate queries. If a set of `start_states` is specified, each query, `goal_constraint`, and `trajectory_constraint` is attempted with each start state (existing start states from a query are ignored). Similarly, the (optional) set of path constraints is combined combinatorially with the start query and start `goal_constraint` pairs (existing `path_constraint` from a query are ignored). The workspace, if specified, overrides any existing workspace parameters.

The benchmarking pipeline does not utilize `MoveGroup`, and `PlanningRequestAdaptors` are **not** invoked.

It is possible to customize a benchmark run by deriving a class from `BenchmarkExecutor` and overriding one or more of the virtual functions. Additionally, a set of functions exists for ease of customization in derived classes:

- `preRunEvent`: invoked immediately before each call to solve
- `postRunEvent`: invoked immediately after each call to solve
- `plannerSwitchEvent`: invoked when the planner changes during benchmarking
- `querySwitchEvent`: invoked before a new benchmark problem begin execution

Note, in the above, a benchmark is a concrete instance of a `PlanningScene`, start state, goal constraints / `trajectory_constraints`, and (optionally) `path_constraints`. A run is one attempt by a specific planner to solve the benchmark.

2.10 Running Tests In MoveIt!

How to test changes to MoveIt! on various robots, including unit and integration tests.

Note: *This is a stub tutorial, to be expanded upon in the future*

2.10.1 Integration Test

A Python-based integration test is available for testing higher-level move_group functionality in MoveIt! - to run

```
rostest moveit_ros_planning_interface python_move_group.test
```

2.10.2 Test Robots

PR2

From the package moveit_pr2

```
roslaunch pr2_moveit_config demo.launch
```

Fanuc M-10iA

From the package moveit_resources

```
roslaunch moveit_resources demo.launch
```

2.10.3 Unit Tests

To run unit tests locally on the entire MoveIt! catkin workspace using catkin-tools:

```
catkin run_tests -iv
```

To run a test for just 1 package:

```
catkin run_tests --no-deps --this -iv
```

2.10.4 Kinematic Tests

An additional test suite for the KinematicBase features in MoveIt! is available in the package moveit_kinematic_tests

CHAPTER 3

Integration with New Robot

Before attempting to integrate a new robot with MoveIt!, check whether your robot has already been setup (see the list of robots running MoveIt!). Otherwise, follow the tutorials in this section to integrate your robot with MoveIt! (and share your results on the MoveIt! mailing list)

3.1 Setup Assistant Tutorial

3.1.1 Overview

The MoveIt! Setup Assistant is a graphical user interface for configuring any robot for use with MoveIt!. Its primary function is generating a Semantic Robot Description Format (SRDF) file for your robot. Additionally, it generates other necessary configuration files for use with the MoveIt! pipeline. To learn more about the SRDF, you can go through the SRDF Overview page.

3.1.2 Pre-requisites

MoveIt! and ROS

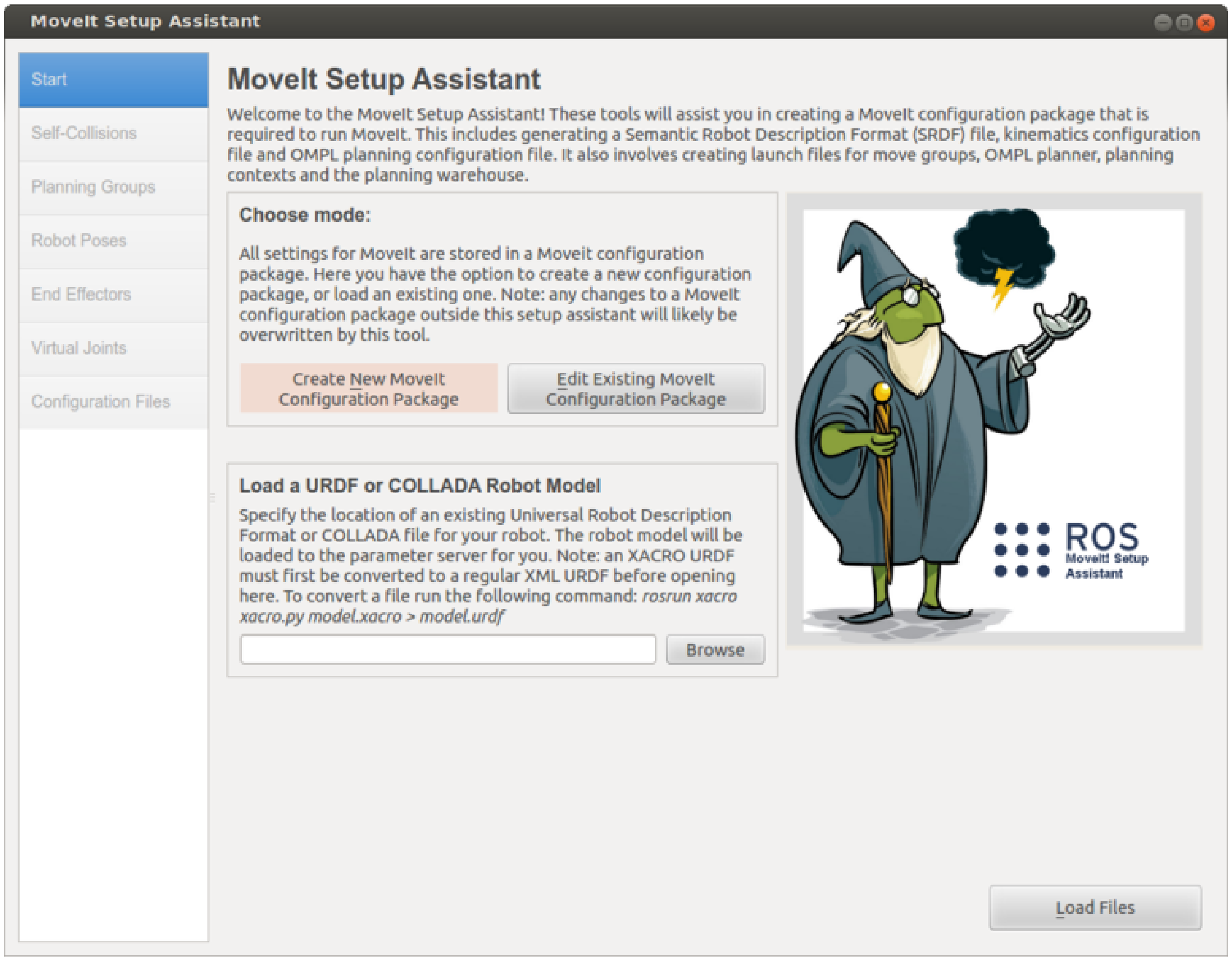
- Follow the instructions for installing MoveIt! first if you have not already done that.

3.1.3 Step 1: Start

- To start the MoveIt! Setup Assistant:

```
roslaunch moveit_setup_assistant setup_assistant.launch
```

- This will bringup the start screen with two choices: *Create New MoveIt! Configuration Package* or *Edit Existing MoveIt! Configuration Package*.
- Click on the *Create New MoveIt! Configuration Package* button to bring up the following screen:



MoveIt Setup Assistant

Welcome to the MoveIt Setup Assistant! These tools will assist you in creating a MoveIt configuration package that is required to run MoveIt. This includes generating a Semantic Robot Description Format (SRDF) file, kinematics configuration file and OMPL planning configuration file. It also involves creating launch files for move groups, OMPL planner, planning contexts and the planning warehouse.

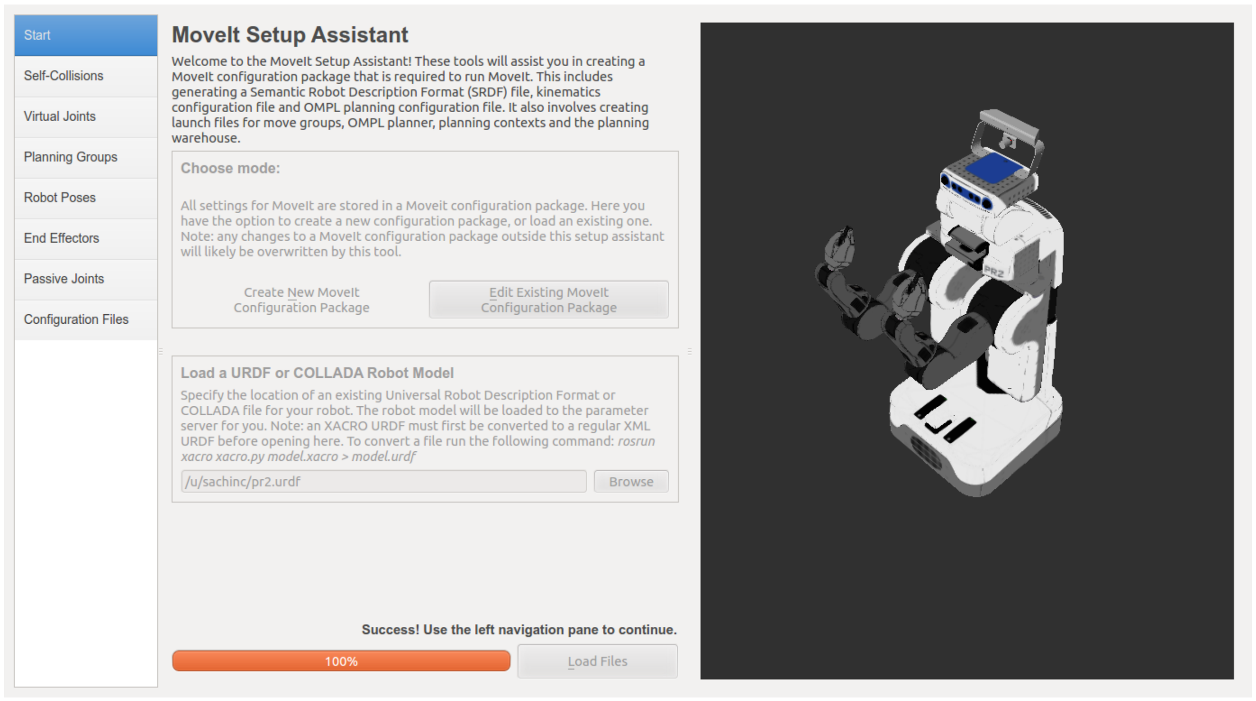
Choose mode:

All settings for MoveIt are stored in a Moveit configuration package. Here you have the option to create a new configuration package, or load an existing one. Note: any changes to a MoveIt configuration package outside this setup assistant will likely be overwritten by this tool.

Load a URDF or COLLADA Robot Model

Specify the location of an existing Universal Robot Description Format or COLLADA file for your robot. The robot model will be loaded to the parameter server for you. Note: an XACRO URDF must first be converted to a regular XML URDF before opening here. To convert a file run the following command: *rosrun xacro xacro.py model.xacro > model.urdf*

- Click on the browse button and navigate to the *pr2.urdf.xacro* file installed when you installed ros-indigo-moveit-full-pr2. (This file gets installed in /opt/ros/indigo/share/pr2_description/robots/pr2.urdf.xacro on Ubuntu with ROS Indigo.) Choose that file and then click *Load Files*. The Setup Assistant will load the files (this might take a few seconds) and present you with this screen:



MoveIt Setup Assistant

Welcome to the MoveIt Setup Assistant! These tools will assist you in creating a MoveIt configuration package that is required to run MoveIt. This includes generating a Semantic Robot Description Format (SRDF) file, kinematics configuration file and OMPL planning configuration file. It also involves creating launch files for move groups, OMPL planner, planning contexts and the planning warehouse.

Choose mode:

All settings for MoveIt are stored in a Moveit configuration package. Here you have the option to create a new configuration package, or load an existing one. Note: any changes to a MoveIt configuration package outside this setup assistant will likely be overwritten by this tool.

Load a URDF or COLLADA Robot Model

Specify the location of an existing Universal Robot Description Format or COLLADA file for your robot. The robot model will be loaded to the parameter server for you. Note: an XACRO URDF must first be converted to a regular XML URDF before opening here. To convert a file run the following command: *rosrun xacro xacro.py model.xacro > model.urdf*

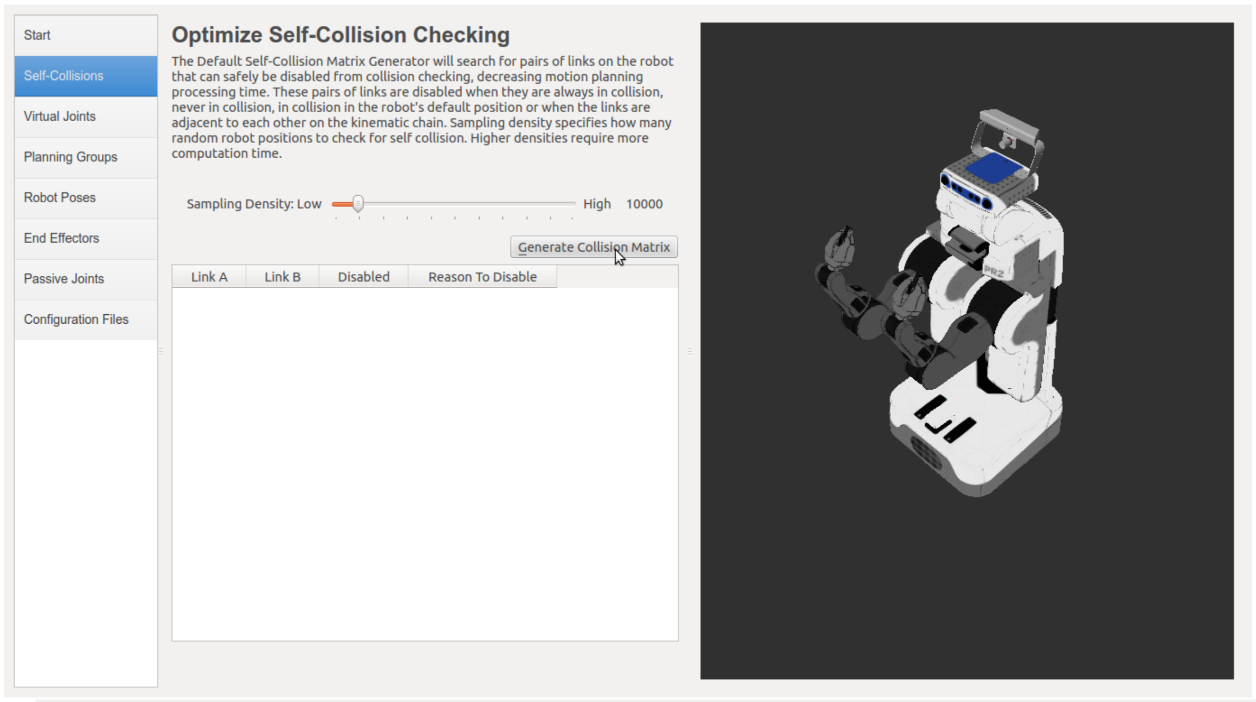
/u/sachinc/pr2.urdf

Success! Use the left navigation pane to continue.

3.1.4 Step 2: Generate Self-Collision Matrix

The Default Self-Collision Matrix Generator searches for pairs of links on the robot that can safely be disabled from collision checking, decreasing motion planning processing time. These pairs of links are disabled when they are always in collision, never in collision, in collision in the robot's default position or when the links are adjacent to each other on the kinematic chain. The sampling density specifies how many random robot positions to check for self collision. Higher densities require more computation time while lower densities have a higher possibility of disabling pairs that should not be disabled. The default value is 10,000 collision checks. Collision checking is done in parallel to decrease processing time.

- Click on the *Self-Collisions* pane selector on the left-hand side and click on the *Regenerate Default Collision Matrix* button. The Setup Assistant will work for a few second before presenting you the results of its computation in the main table.



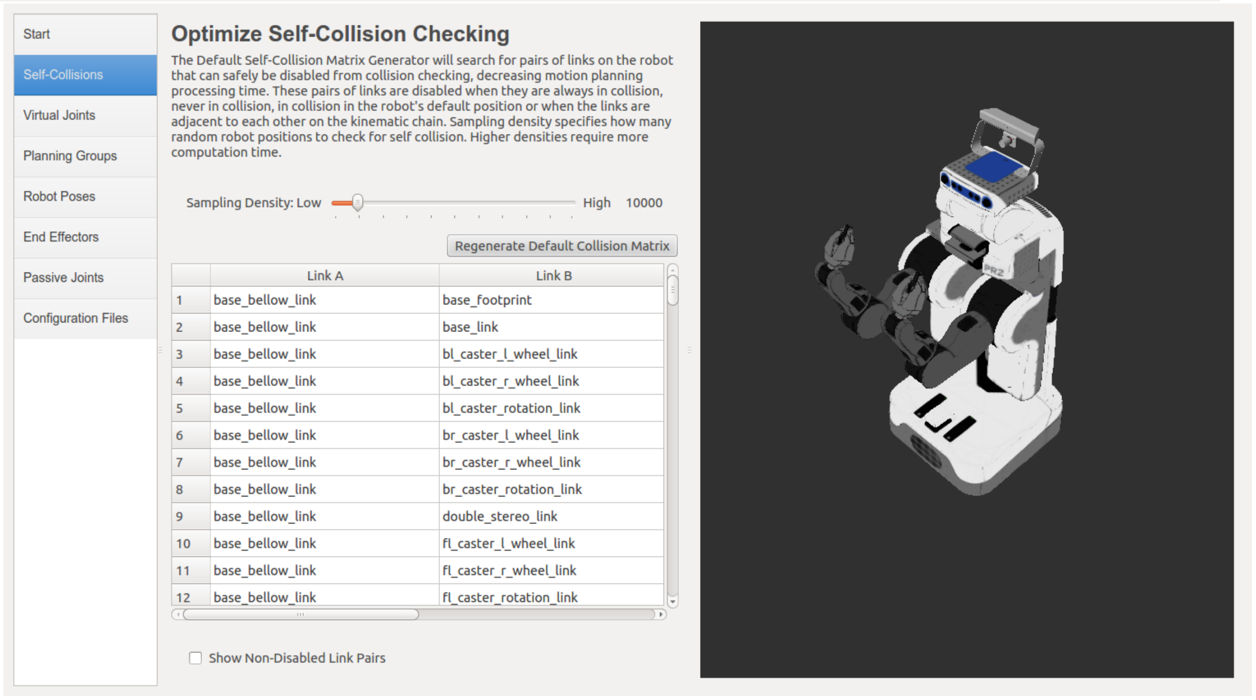
Optimize Self-Collision Checking

The Default Self-Collision Matrix Generator will search for pairs of links on the robot that can safely be disabled from collision checking, decreasing motion planning processing time. These pairs of links are disabled when they are always in collision, never in collision, in collision in the robot's default position or when the links are adjacent to each other on the kinematic chain. Sampling density specifies how many random robot positions to check for self collision. Higher densities require more computation time.

Sampling Density: Low High 10000

Link A	Link B	Disabled	Reason To Disable

→



Optimize Self-Collision Checking

The Default Self-Collision Matrix Generator will search for pairs of links on the robot that can safely be disabled from collision checking, decreasing motion planning processing time. These pairs of links are disabled when they are always in collision, never in collision, in collision in the robot's default position or when the links are adjacent to each other on the kinematic chain. Sampling density specifies how many random robot positions to check for self collision. Higher densities require more computation time.

Sampling Density: Low High 10000

	Link A	Link B
1	base_bellow_link	base_footprint
2	base_bellow_link	base_link
3	base_bellow_link	bl_caster_l_wheel_link
4	base_bellow_link	bl_caster_r_wheel_link
5	base_bellow_link	bl_caster_rotation_link
6	base_bellow_link	br_caster_l_wheel_link
7	base_bellow_link	br_caster_r_wheel_link
8	base_bellow_link	br_caster_rotation_link
9	base_bellow_link	double_stereo_link
10	base_bellow_link	fl_caster_l_wheel_link
11	base_bellow_link	fl_caster_r_wheel_link
12	base_bellow_link	fl_caster_rotation_link

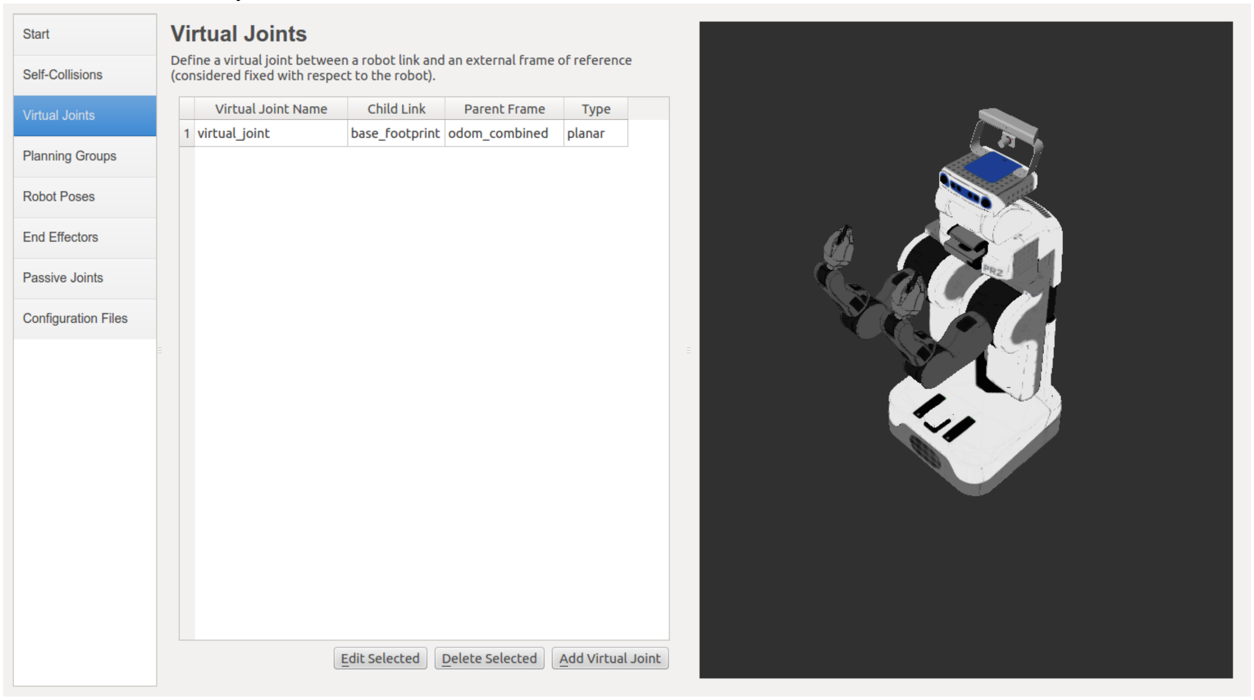
☐ Show Non-Disabled Link Pairs

3.1.5 Step 3: Add Virtual Joints

Virtual joints are used primarily to attach the robot to the world. For the PR2 we will define only one virtual joint attaching the *base_footprint* of the PR2 to the *odom_combined* world frame. This virtual joint represents the motion of the base of the robot in a plane.

- Click on the *Virtual Joints* pane selector. Click on *Add Virtual Joint*
- Set the joint name as "virtual_joint"

- Set the child link as "base_footprint" and the parent frame name as "odom_combined".
- Set the Joint Type as "planar".
- Click *Save* and you should see this screen:



Virtual Joints

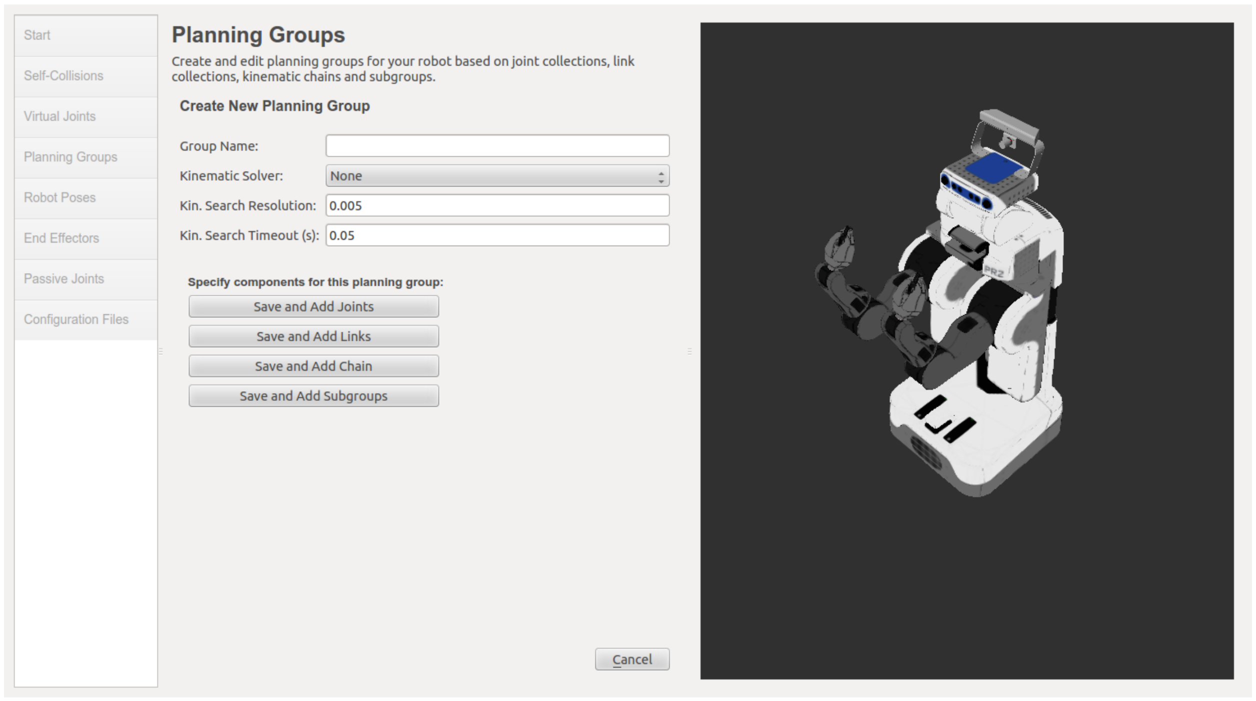
Define a virtual joint between a robot link and an external frame of reference (considered fixed with respect to the robot).

	Virtual Joint Name	Child Link	Parent Frame	Type
1	virtual_joint	base_footprint	odom_combined	planar

3.1.6 Step 4: Add Planning Groups

Planning groups are used for semantically describing different parts of your robot, such as defining what an arm is, or an end effector.

- Click on the *Planning Groups* pane selector.
- Click on *Add Group* and you should see the following screen:



Planning Groups

Create and edit planning groups for your robot based on joint collections, link collections, kinematic chains and subgroups.

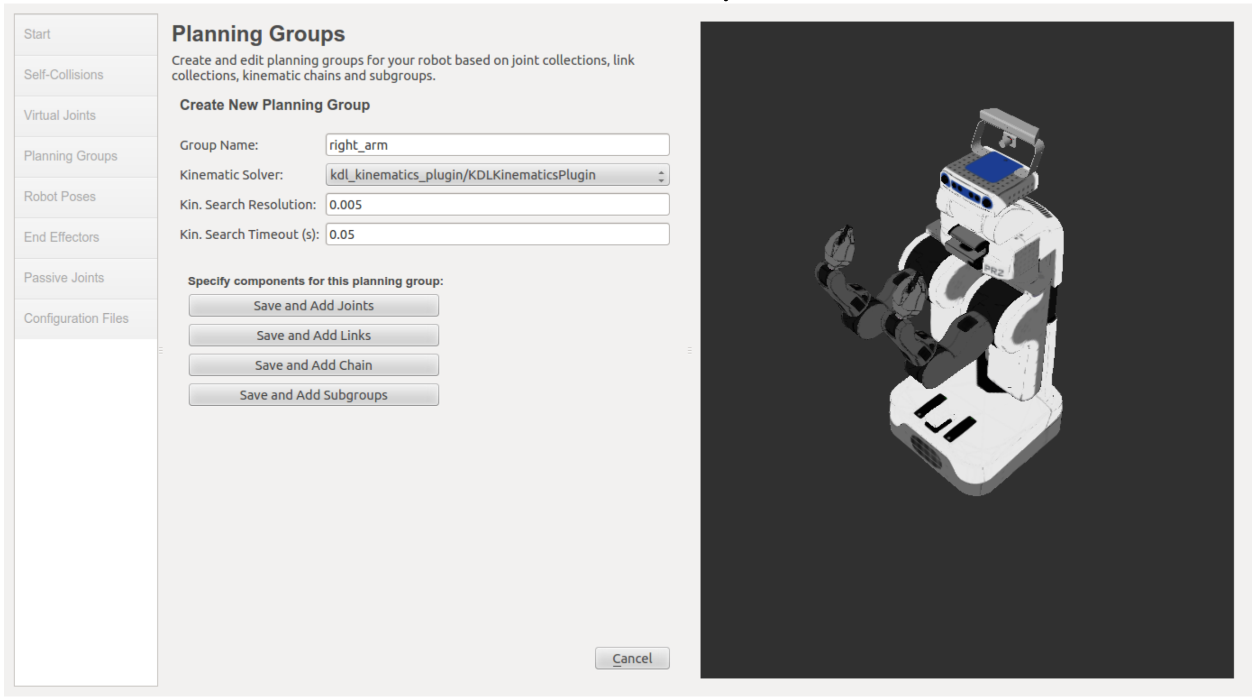
Create New Planning Group

Group Name:
Kinematic Solver: None
Kin. Search Resolution: 0.005
Kin. Search Timeout (s): 0.05

Specify components for this planning group:

Add the right arm

- We will first add the PR2 right arm as a planning group
  - Enter *Group Name* as **right_arm**
  - Choose *kdl_kinematics_plugin/KDLKinematicsPlugin* as the kinematics solver. *Note: if you have a custom robot and would like a powerful custom IK solver, see* Kinematics/IKFast
  - Let *Kin. Search Resolution* and *Kin. Search Timeout* stay at their default values.



Planning Groups

Create and edit planning groups for your robot based on joint collections, link collections, kinematic chains and subgroups.

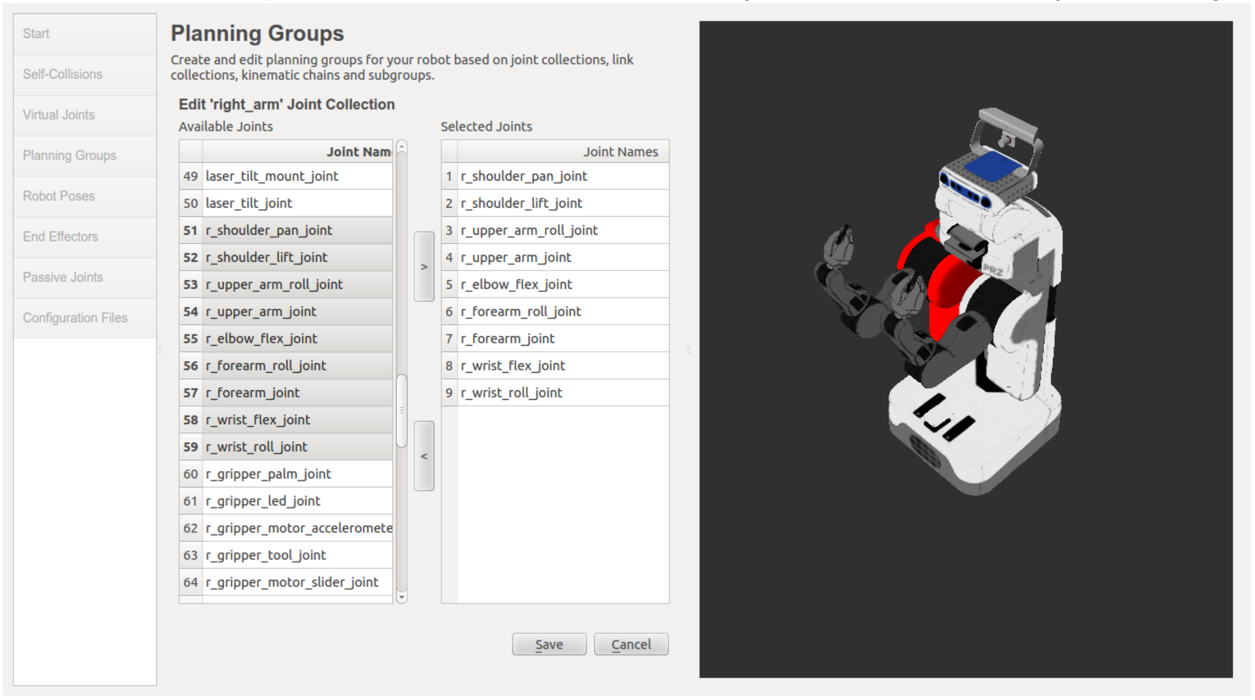
Create New Planning Group

Group Name: right_arm
Kinematic Solver: kdl_kinematics_plugin/KDLKinematicsPlugin
Kin. Search Resolution: 0.005
Kin. Search Timeout (s): 0.05

Specify components for this planning group:

- Now, click on the *Save and Add Joints* button. You will see a list of joints on the left hand side. You need to choose all the joints that belong to the right arm and add them to the right hand side. The joints are arranged in the order that they are stored in an internal tree structure. This makes it easy to select a serial chain of joints.

– Click on **r_shoulder_pan_joint**, hold down the **Shift** button on your keyboard and then click on the *r_wrist_roll_joint*. Now click on the **>** button to add these joints into the list of selected joints on the right.



Planning Groups

Create and edit planning groups for your robot based on joint collections, link collections, kinematic chains and subgroups.

Edit 'right_arm' Joint Collection

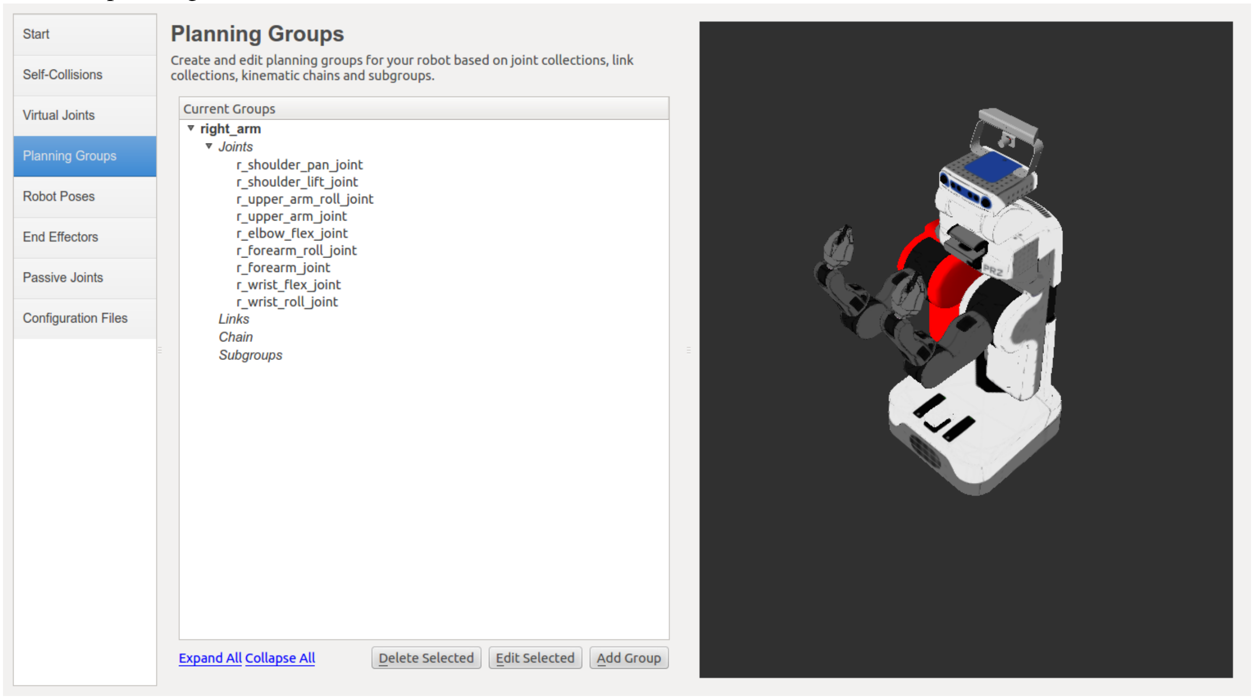
Available Joints

	Joint Nam
49	laser_tilt_mount_joint
50	laser_tilt_joint
51	r_shoulder_pan_joint
52	r_shoulder_lift_joint
53	r_upper_arm_roll_joint
54	r_upper_arm_joint
55	r_elbow_flex_joint
56	r_forearm_roll_joint
57	r_forearm_joint
58	r_wrist_flex_joint
59	r_wrist_roll_joint
60	r_gripper_palm_joint
61	r_gripper_led_joint
62	r_gripper_motor_acceleromete
63	r_gripper_tool_joint
64	r_gripper_motor_slider_joint

Selected Joints

	Joint Names
1	r_shoulder_pan_joint
2	r_shoulder_lift_joint
3	r_upper_arm_roll_joint
4	r_upper_arm_joint
5	r_elbow_flex_joint
6	r_forearm_roll_joint
7	r_forearm_joint
8	r_wrist_flex_joint
9	r_wrist_roll_joint

• Click *Save* to save the selected group. Note that each arm of the PR2 has only 7 joints and yet we added 9 joints here. Some of the joints (r_upper_arm_joint and r_forearm_joint) are **Fixed** joints and will not be used for planning or kinematics.



Planning Groups

Create and edit planning groups for your robot based on joint collections, link collections, kinematic chains and subgroups.

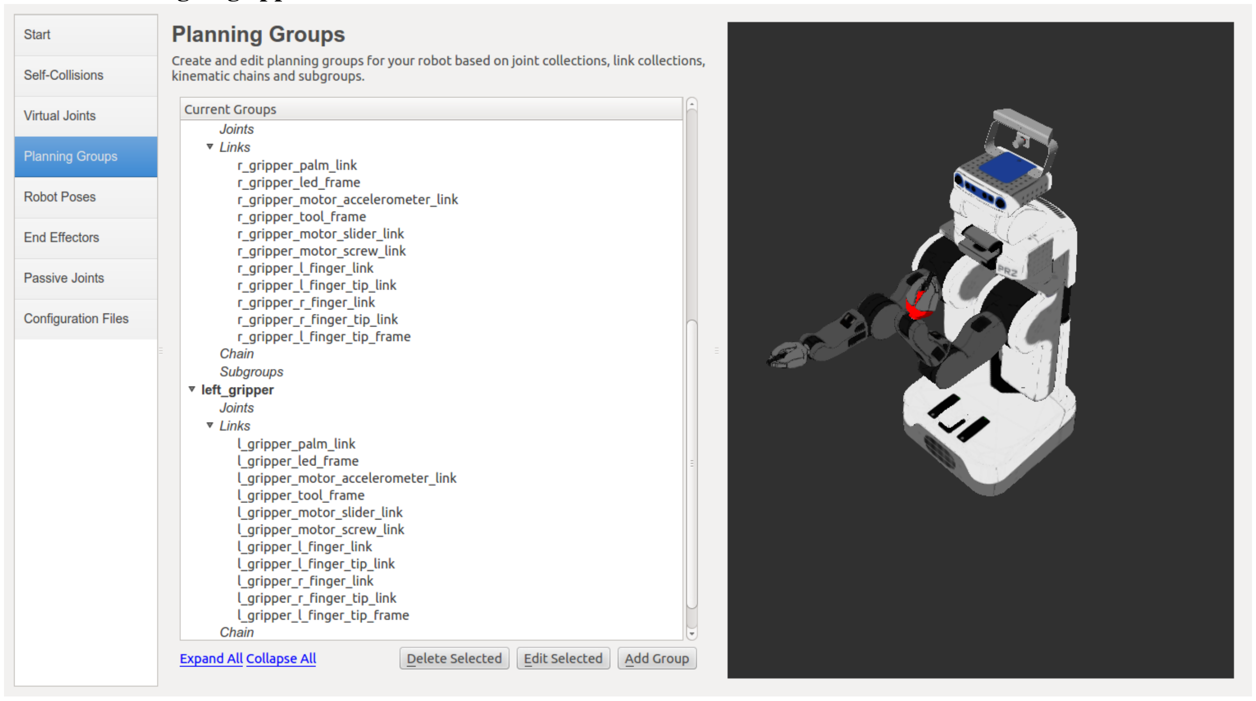
Current Groups
- right_arm
  - *Joints*
    - r_shoulder_pan_joint
    - r_shoulder_lift_joint
    - r_upper_arm_roll_joint
    - r_upper_arm_joint
    - r_elbow_flex_joint
    - r_forearm_roll_joint
    - r_forearm_joint
    - r_wrist_flex_joint
    - r_wrist_roll_joint
  - *Links*
  - *Chain*
  - *Subgroups*

Add the left arm

Now, add the left arm in a similar manner choosing the joints from the l_shoulder_pan_joint to the l_wrist_roll_joint.

Add the grippers

- We will also add two groups for the right and left end effectors. NOTE that you will do this using a different procedure than adding the arms.
  - Click on the *Add Group* button.
  - Enter *Group Name* as **right_gripper**
  - Let *Kin. Search Resolution* and *Kin. Search Timeout* stay at their default values.
  - Click on the *Save and Add Links* button.
  - Choose all links that start with **right_gripper** and add them to the list of *Selected Links* on the right hand side.
  - Click *Save*
  - Repeat the same procedure for the left arm of the PR2, choosing links that start with **left_gripper** instead of **right_gripper** this time.



Planning Groups

Create and edit planning groups for your robot based on joint collections, link collections, kinematic chains and subgroups.

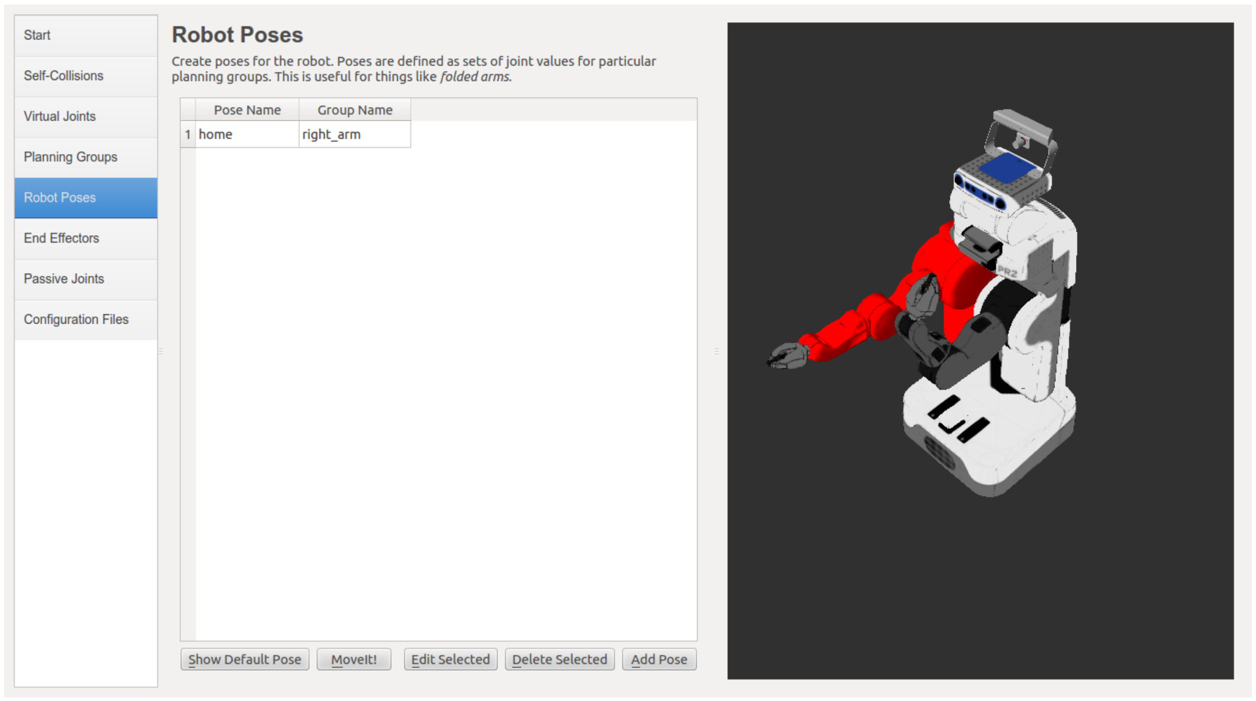
Current Groups

```
    Joints
  ▾ Links
      r_gripper_palm_link
      r_gripper_led_frame
      r_gripper_motor_accelerometer_link
      r_gripper_tool_frame
      r_gripper_motor_slider_link
      r_gripper_motor_screw_link
      r_gripper_l_finger_link
      r_gripper_l_finger_tip_link
      r_gripper_r_finger_link
      r_gripper_r_finger_tip_link
      r_gripper_l_finger_tip_frame
    Chain
    Subgroups
▾ left_gripper
    Joints
  ▾ Links
      l_gripper_palm_link
      l_gripper_led_frame
      l_gripper_motor_accelerometer_link
      l_gripper_tool_frame
      l_gripper_motor_slider_link
      l_gripper_motor_screw_link
      l_gripper_l_finger_link
      l_gripper_l_finger_tip_link
      l_gripper_r_finger_link
      l_gripper_r_finger_tip_link
      l_gripper_l_finger_tip_frame
    Chain
```

3.1.7 Step 5: Add Robot Poses

The Setup Assistant allows you to add certain fixed poses into the configuration. This helps if, for example, you want to define a certain position of the robot as a **Home** position.

- Click on the *Robot Poses* pane.
- Click *Add Pose*. Choose a name for the pose. The robot will be in its *Default* position where the joint values are set to the mid-range of the allowed joint value range. Move the individual joints around until you are happy and then *Save* the pose. Note how poses are associated with particular groups. You can save individual poses for each group.
- **IMPORTANT TIP**: Try to move all the joints around. If there is something wrong with the joint limits in your URDF, you should be able to see it immediately here.



Robot Poses

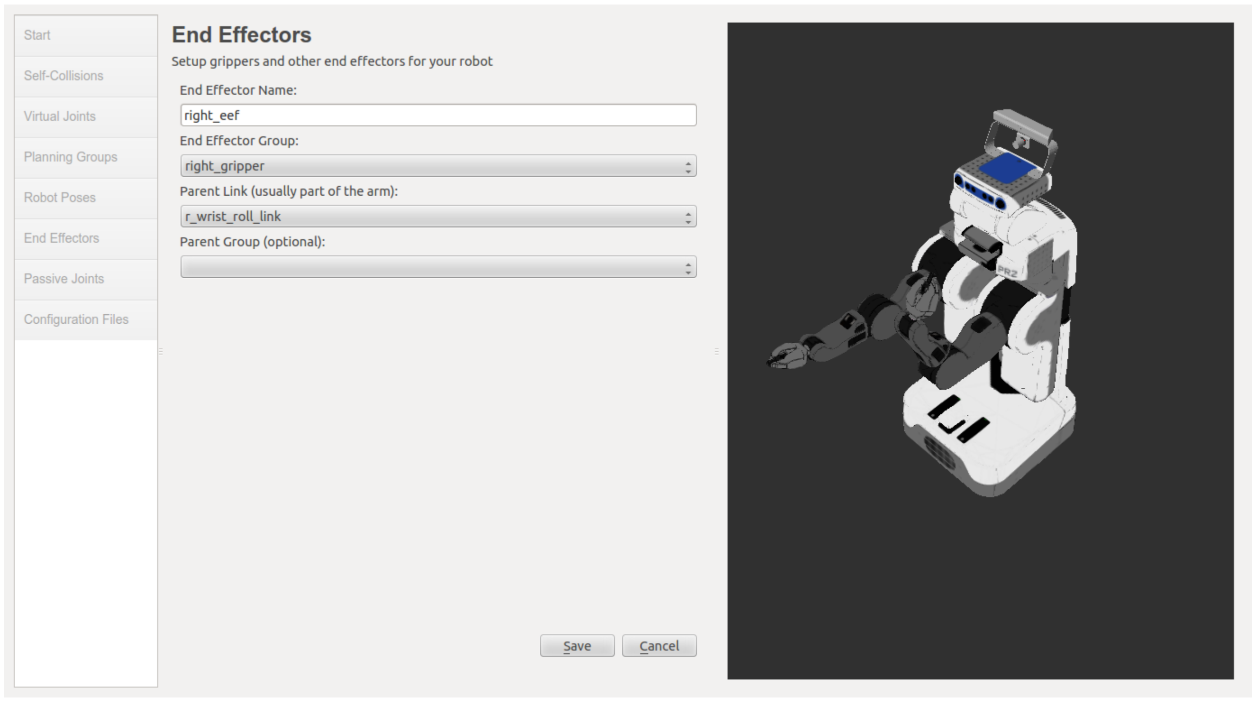
Create poses for the robot. Poses are defined as sets of joint values for particular planning groups. This is useful for things like *folded arms*.

	Pose Name	Group Name
1	home	right_arm

3.1.8 Step 6: Label End Effectors

We have already added the right and left grippers of the PR2. Now, we will designate these two groups as special groups: **end-effectors**. Designating these groups as end effector allows some special operations to happen on them internally.

- Click on the *End Effectors* pane.
- Click *Add End-Effectors*.
- Choose **right_eef** as the *End Effector Name* for the right gripper.
- Select **right_gripper** as the *End Effector Group*.
- Select **r_wrist_roll_link** as the *Parent Link* for this end-effector.
- Leave *Parent Group* blank.



End Effectors

Setup grippers and other end effectors for your robot

End Effector Name: right_eef

End Effector Group: right_gripper

Parent Link (usually part of the arm): r_wrist_roll_link

Parent Group (optional):

- Click *Save*.
- Add the **left_eef** in a similar manner.

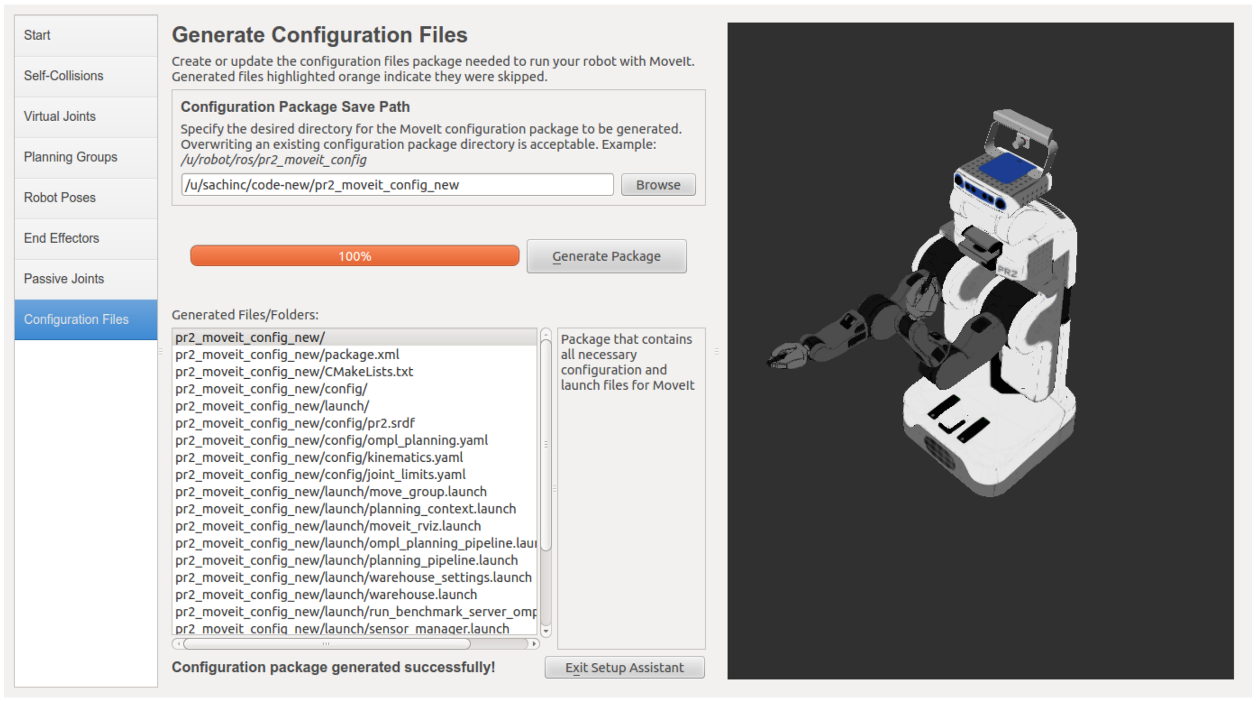
3.1.9 Step 7: Add Passive Joints

The passive joints tab is meant to allow specification of any passive joints that might exist in a robot. This tells the planners that they cannot (kinematically) plan for these joints. Examples of passive joints include passive casters. The PR2 does not have any passive joints so we will skip this step.

3.1.10 Step 8: Generate Configuration Files

You are almost there. One last step - generating all the configuration files that you will need to start using MoveIt!

- Click on the *Configuration Files* pane. Choose a location and name for the ROS package that will be generated containing your new set of configuration files (e.g. click browse, select a good location (e.g. your home dir), click **Create New Folder**, enter “pr2_moveit_generated”, and click **Choose**. “pr2_moveit_generated” is the location used in the rest of the documentation on this wiki). This does not have to be within your ROS package path. All generated files will go directly into the directory you have chosen.
- Click on the *Generate Package* button. The Setup Assistant will now generate and write a set of launch and config files into the directory of your choosing. All the generated files will appear in the Generated Files/Folders tab and you can click on each of them for a description of what they contain.



Generate Configuration Files

Create or update the configuration files package needed to run your robot with MoveIt. Generated files highlighted orange indicate they were skipped.

Configuration Package Save Path

Specify the desired directory for the MoveIt configuration package to be generated. Overwriting an existing configuration package directory is acceptable. Example: */u/robot/ros/pr2_moveit_config*

/u/sachinc/code-new/pr2_moveit_config_new

Generated Files/Folders:

```
pr2_moveit_config_new/
pr2_moveit_config_new/package.xml
pr2_moveit_config_new/CMakeLists.txt
pr2_moveit_config_new/config/
pr2_moveit_config_new/launch/
pr2_moveit_config_new/config/pr2.srdf
pr2_moveit_config_new/config/ompl_planning.yaml
pr2_moveit_config_new/config/kinematics.yaml
pr2_moveit_config_new/config/joint_limits.yaml
pr2_moveit_config_new/launch/move_group.launch
pr2_moveit_config_new/launch/planning_context.launch
pr2_moveit_config_new/launch/moveit_rviz.launch
pr2_moveit_config_new/launch/ompl_planning_pipeline.lau
pr2_moveit_config_new/launch/planning_pipeline.launch
pr2_moveit_config_new/launch/warehouse_settings.launch
pr2_moveit_config_new/launch/warehouse.launch
pr2_moveit_config_new/launch/run_benchmark_server_omp
pr2_moveit_config_new/launch/sensor_manager.launch
```

Package that contains all necessary configuration and launch files for MoveIt

Configuration package generated successfully!

- Congratulations!! - You are now done generating the configuration files you need for MoveIt!

3.1.11 What's Next

The MoveIt! Rviz plugin

- Start looking at how you can use the generated configuration files to play with MoveIt! using the MoveIt! Rviz Plugin.

Setup IKFast Inverse Kinematics Solver

- A faster IK solver than the default KDL solver, but takes some additional steps to setup: Kinematics/IKFast

3.1.12 Additional Reading

The SRDF

- See the SRDF page for more details on the components of the SRDF mentioned in this tutorial.

URDF

- The URDF is the native robot description format in ROS and allows you to describe the kinematics, inertial, visual and sensing properties of your robot. Read through the URDF specific documentation to see how the URDF is used with MoveIt!

3.2 Controllers Configuration Tutorial

In this section, we will walk through configuring MoveIt! with the controllers on your robot. We will assume that your robot offers a `FollowJointTrajectory` action service for the arms on your robot and (optionally) a `GripperCommand` service for your gripper.

3.2.1 YAML Configuration

The first file to create is a YAML configuration file (call it *controllers.yaml* and place it in the *config* directory of your MoveIt! config directory). This will specify the controller configuration for your robot. Here's an example file for configuring a `FollowJointTrajectory` action controller for two different arms (left and right) and a `GripperCommand` gripper controller for two grippers

```
controller_list:
 - name: r_arm_controller
   action_ns: follow_joint_trajectory
   type: FollowJointTrajectory
   default: true
   joints:
     - r_shoulder_pan_joint
     - r_shoulder_lift_joint
     - r_upper_arm_roll_joint
     - r_elbow_flex_joint
     - r_forearm_roll_joint
     - r_wrist_flex_joint
     - r_wrist_roll_joint
 - name: l_arm_controller
   action_ns: follow_joint_trajectory
   type: FollowJointTrajectory
   default: true
   joints:
     - l_shoulder_pan_joint
     - l_shoulder_lift_joint
     - l_upper_arm_roll_joint
     - l_elbow_flex_joint
     - l_forearm_roll_joint
     - l_wrist_flex_joint
     - l_wrist_roll_joint
 - name: gripper_controller
   action_ns: gripper_action
   type: GripperCommand
   default: true
   joints:
     - l_gripper_joint
     - r_gripper_joint
```

We will walk through the parameters for both types of controllers.

3.2.2 FollowJointTrajectory Controller Interface

The parameters are:

- *name*: The name of the controller. (See debugging information below for important notes).
- *action_ns*: The action namespace for the controller. (See debugging information below for important notes).
- *type*: The type of action being used (here FollowJointTrajectory).
- *default*: The default controller is the primary controller chosen by MoveIt! for communicating with a particular set of joints.
- *joints*: Names of all the joints that are being addressed by this interface.

3.2.3 GripperCommand Controller Interface

The parameters are:

- *name*: The name of the controller. (See debugging information below for important notes).
- *action_ns*: The action namespace for the controller. (See debugging information below for important notes).
- *type*: The type of action being used (here GripperCommand).
- *default*: The default controller is the primary controller chosen by MoveIt! for communicating with a particular set of joints.
- *joints*: Names of all the joints that are being addressed by this interface.

3.2.4 Create the Controller launch file

Now, create the controller launch file (call it *robot_moveit_controller_manager.launch* where *robot* is the name of your robot - the robot name needs to match the name specified when you created your MoveIt! config directory).

Add the following lines to this file

```
<launch>
 <!-- Set the param that trajectory_execution_manager needs to find the controller plugin -->
 <arg name="moveit_controller_manager" default="moveit_simple_controller_manager/MoveItSimpleControll
 <param name="moveit_controller_manager" value="$(arg moveit_controller_manager)"/>
 <!-- load controller_list -->
 <rosparam file="$(find my_robot_name_moveit_config)/config/controllers.yaml"/>
</launch>
```

MAKE SURE to replace *my_robot_name_moveit_config* with the correct path for your MoveIt! config directory.

Now, you should be ready to have MoveIt! talk to your robot.

3.2.5 Debugging Information

The `FollowJointTrajectory` or `GripperCommand` interfaces on your robot must be communicating in the namespace: `\name\action_ns`. In the above example, you should be able to see the following topics (using *rostopic list*) on your robot:

- /r_arm_controller/follow_joint_trajectory/goal
- /r_arm_controller/follow_joint_trajectory/feedback
- /r_arm_controller/follow_joint_trajectory/result
- /l_arm_controller/follow_joint_trajectory/goal
- /l_arm_controller/follow_joint_trajectory/feedback
- /l_arm_controller/follow_joint_trajectory/result
- /gripper_controller/gripper_action/goal
- /gripper_controller/gripper_action/feedback
- /gripper_controller/gripper_action/result

You should also be able to see (using *rostopic info topic_name*) that the topics are published/subscribed to by the controllers on your robot and also by the *move_group* node.

3.3 3D Perception/Configuration Tutorial

In this section, we will walk through configuring the 3D sensors on your robot with MoveIt!. The primary component in MoveIt

- The PointCloud Occupany Map Updater: which can take as input point clouds (`sensor_msgs/PointCloud2`)
- The Depth Image Occupancy Map Updater: which can take as input Depth Images (`sensor_msgs/Image`)

3.3.1 YAML Configuration file (Point Cloud)

We will have to generate a YAML configuration file for configuring the 3D sensors. An example file for processing point clouds can be found in the moveit_pr2 github project

```
sensors:
  - sensor_plugin: occupancy_map_monitor/PointCloudOctomapUpdater
    point_cloud_topic: /head_mount_kinect/depth_registered/points
    max_range: 5.0
    point_subsample: 1
    padding_offset: 0.1
    padding_scale: 1.0
    filtered_cloud_topic: filtered_cloud
```

The general parameters are:

- *sensor_plugin*: The name of the plugin that we are using.

Parameters specific to the Point cloud updater are:

- *point_cloud_topic*: This specifies the topic to listen on for a point cloud.
- *max_range*: (in m) Points further than this will not be used.
- *point_subsample*: Choose one of every *point_subsample* points.
- *padding_offset*: The size of the padding (in cm).
- *padding_scale*:
- *filtered_cloud_topic*: The topic on which the filtered cloud will be published (mainly for debugging). The filtering cloud is the resultant cloud after self-filtering has been performed.

3.3.2 YAML Configuration file (Depth Map)

We will have to generate a rgbd.yaml configuration file for configuring the 3D sensors. An example file for processing point clouds can be found in the moveit_advanced github project

```
sensors:
  - sensor_plugin: occupancy_map_monitor/DepthImageOctomapUpdater
    image_topic: /head_mount_kinect/depth_registered/image_raw
    queue_size: 5
    near_clipping_plane_distance: 0.3
    far_clipping_plane_distance: 5.0
    shadow_threshold: 0.2
    padding_scale: 4.0
    padding_offset: 0.03
    filtered_cloud_topic: filtered_cloud
```

The general parameters are:

- *sensor_plugin*: The name of the plugin that we are using.

Parameters specific to the Depth Map updater are:

- *image_topic*: This specifies the topic to listen on for a depth image.
- *queue_size*: he number of images to queue up
- *near_clipping_plane_distance*:
- *far_clipping_plane_distance*:
- *shadow_threshold*:
- *padding_offset*: The size of the padding (in cm).
- *padding_scale*:
- *filtered_cloud_topic*: The topic on which the filtered cloud will be published (mainly for debugging). The filtering cloud is the resultant cloud after self-filtering has been performed.

3.3.3 Update the launch file

Add the YAML file to the launch script

You will now need to update the *moveit_sensor_manager.launch* file in the "launch" directory of your MoveIt! configuration directory with this sensor information (this file is auto-generated by the Setup Assistant but is empty). You will need to add the following line into that file to configure the set of sensor sources for MoveIt! to use:

```
<rosparam command="load" file="$(find my_moveit_config)/config/sensors_kinect.yaml" />
```

Note that you will need to input the path to the right file you have created above.

Octomap Configuration

You will also need to configure the Octomap by adding the following lines into the *moveit_sensor_manager.launch*:

```
<param name="octomap_frame" type="string" value="odom_combined" />
<param name="octomap_resolution" type="double" value="0.05" />
<param name="max_range" type="double" value="5.0" />
```

MoveIt! uses an octree based framework to represent the world around it. The *Octomap* parameters above are configuration pa

- *octomap_frame*: specifies the coordinate frame in which this representation will be stored. If you are working with a mobile robot, this frame should be a fixed frame in the world.
- *octomap_resolution*: specifies the resolution at which this representation is maintained (in meters).
- *max_range*: specifies the maximum range value to be applied for any sensor input to this node.

3.4 Generate IKFast Plugin Tutorial

In this section, we will walk through configuring an IKFast plugin for MoveIt!

3.4.1 What is IKFast?

From Wikipedia: IKFast, the Robot Kinematics Compiler, is a powerful inverse kinematics solver provided within Rosen Diankov's OpenRAVE motion planning software. Unlike most inverse kinematics solvers, IKFast can analytically solve the kinematics equations of any complex kinematics chain, and generate language-specific files (like C++) for later use. The end result is extremely stable solutions that can run as fast as 5 microseconds on recent processors

3.4.2 MoveIt! IKFast

MoveIt! IKFast is a tool that generates a IKFast kinematics plugin for MoveIt using OpenRave generated cpp files. This tutorial will step you through setting up your robot to utilize the power of IKFast. MoveIt! IKFast is tested on ROS Groovy with Catkin using OpenRave 0.8 with a 6dof and 7dof robot arm manipulator. While it works in theory, currently the IKFast plugin generator tool does not work with >7 degree of freedom arms.

3.4.3 Pre-requisites

You should have already created a MoveIt! configuration package for your robot, by using the Setup Assistant.

3.4.4 MoveIt! IKFast Installation

Install the MoveIt! IKFast package either from debs or from source.

Binary Install

```
sudo apt-get install ros-indigo-moveit-ikfast
```

Source

Inside your catkin workspace

```
git clone https://github.com/ros-planning/moveit_ikfast.git
```

3.4.5 OpenRAVE Installation

Binary Install (only Indigo / Ubuntu 14.04)

sudo apt-get install ros-indigo-openrave

Note: you have to set *export PYTHONPATH=$PYTHONPATH:'openrave-config –python-dir* (2016.9.1)

Source Install

Thanks to instructions from Stéphane Caron in https://scaron.info/teaching/troubleshooting-openrave-installation.html

```
git clone --branch latest_stable https://github.com/rdiankov/openrave.git
cd openrave && mkdir build && cd build
cmake ..
make -j4
sudo make install
```

Working commit numbers 9c79ea26... comfirmed for Ubuntu 14.04 and comfirmed, according to Stéphane Caron, for 16.04.

Please report your results with this on the moveit-users mailing list.

3.4.6 Create Collada File For Use With OpenRave

First you will need robot description file that is in Collada or OpenRave robot format.

If your robot is not in this format we recommend you create a ROS URDF file, then convert it to a Collada .dae file using the following command:

```
rosrun collada_urdf urdf_to_collada <myrobot_name>.urdf <myrobot_name>.dae
```

where <myrobot_name> is the name of your robot.

Often floating point issues arrise in converting a URDF file to Collada file, so a script has been created to round all the numbers down to x decimal places in your .dae file. Its probably best if you skip this step initially and see if IKFast can generate a solution with your default values, but if the generator takes longer than, say, an hour, try the following:

```
rosrun moveit_ikfast round_collada_numbers.py <input_dae> <output_dae> <decimal places>
```

From experience we recommend 5 decimal places, but if the OpenRave ikfast generator takes to long to find a solution, lowering the number of decimal places should help. For example:

```
rosrun moveit_ikfast round_collada_numbers.py <myrobot_name>.dae <myrobot_name>.rounded.dae 5
```

To see the links in your newly generated Collada file

You may need to install package **libsoqt4-dev** to have the display working:

```
openrave-robot.py <myrobot_name>.dae --info links
```

This is useful if you have a 7-dof arm and you need to fill in a –freeindex parameter, discussed later.

To test your newly generated Collada file in OpenRave:

```
openrave <myrobot_name>.dae
```

Create IKFast Solution CPP File

Once you have a numerically rounded Collada file its time to generate the C++ .h header file that contains the analytical IK solution for your robot.

3.4.7 Select IK Type

You need to choose which sort of IK you want. See this page for more info. The most common IK type is *transform6d*.

3.4.8 Choose Planning Group

If your robot has more than one arm or "planning group" that you want to generate an IKFast solution for, choose one to generate first. The following instructions will assume you have chosen one <planning_group_name> that you will create a plugin for. Once you have verified that the plugin works, repeat the following instructions for any other planning groups you have. For example, you might have 2 planning groups:

```
<planning_group_name> = "left_arm"
<planning_group_name> = "right_arm"
```

3.4.9 Identify Link Numbers

You also need the link index numbers for the *base_link* and *end_link* between which the IK will be calculated. You can count the number of links by viewing a list of links in your model:

```
openrave-robot.py <myrobot_name>.dae --info links
```

A typical 6-DOF manipulator should have 6 arm links + a dummy base_link as required by ROS specifications. If no extra links are present in the model, this gives: *baselink=0* and *eelink=6*. Often, an additional tool_link will be provided to position the grasp/tool frame, giving *eelink=7*.

The manipulator below also has another dummy mounting_link, giving *baselink=1* and *eelink=8*.

name	index	parents
base_link	0	
mounting_link	1	base_link
link1_rotate	2	mounting_link
link2	3	link1_rotate
link3	4	link2
link4	5	link3
link5	6	link4
link6_wrist	7	link5
tool_link	8	link6_wrist

Generate IK Solver

To generate the IK solution between the manipulator's base and tool frames for a 6 dof arm, use the following command format:

```
python `openrave-config --python-dir`/openravepy/_openravepy_/ikfast.py --robot=<myrobot_name>.dae --
```

where <ikfast_output_path> is recommended to be a path that points to a file named ikfast61_<planning_group_name>.cpp.

For a 7 dof arm, you will need to specify a free link:

```
python `openrave-config --python-dir`/openravepy/_openravepy_/ikfast.py --robot=<myrobot_name>.dae --
```

The speed and success of this process will depend on the complexity of your robot. A typical 6 DOF manipulator with 3 intersecting axis at the base or wrist will take only a few minutes to generate the IK.

Known issue –freeindex argument is known to have a bug that it cannot handle tree index correctly. Say –baselink=2 –eelink=16 and links index from 3 to 9 is not related to current planning group chain. In that case –freeindex will expect index 2 as link 2, but index 3 as link 10 ... and index 9 as link 16.

You should consult the OpenRAVE mailing list and ROS Answers for information about 5 and 7 DOF manipulators.

Create Plugin

Create the package that will contain the IK plugin. We recommend you name the package <myrobot_name>_ikfast_<planning_group_name>_plugin. From here on out we'll refer to your IKFast package as simply <moveit_ik_plugin_pkg>:

```
cd ~/catkin_ws/src
catkin_create_pkg <moveit_ik_plugin_pkg>
```

Build your workspace so the new package is detected (can be 'roscd'):

```
cd ~/catkin_ws
catkin_make
```

Create the plugin source code:

```
rosrun moveit_ikfast create_ikfast_moveit_plugin.py <myrobot_name> <planning_group_name> <moveit_ik_p
```

Or without ROS:

```
python /path/to/create_ikfast_moveit_plugin.py <myrobot_name> <planning_group_name> <moveit_ik_plugi
```

Parameters

- *myrobot_name* - name of robot as in your URDF
- *planning_group_name* - name of the planning group you would like to use this solver for, as referenced in your SRDF and kinematics.yaml
- *moveit_ik_plugin_pkg* - name of the new package you just created
- *ikfast_output_path* - file path to the location of your generated IKFast output.cpp file

This will generate a new source file <myrobot_name>_<planning_group_name>_ikfast_moveit_plugin.cpp in the src/ directory, and modify various configuration files.

Build your workspace again to create the ik plugin:

```
cd ~/catkin_ws
catkin_make
```

This will build the new plugin library lib/lib<myrobot_name>_<planning_group_name>_moveit_ikfast_moveit_plugin.so that can be used with MoveIt!

Usage

The IKFast plugin should function identically to the default KDL IK Solver, but with greatly increased performance. The MoveIt configuration file is automatically edited by the moveit_ikfast script but you can switch between the KDL and IKFast solvers using the *kinematics_solver* parameter in the robot's kinematics.yaml file

```
rosed <myrobot_name>_moveit_config/config/kinematics.yaml
```

Edit these parts:

```
<planning_group_name>:
  kinematics_solver: <myrobot_name>_<planning_group_name>_kinematics/IKFastKinematicsPlugin
-INSTEAD OF-
  kinematics_solver: kdl_kinematics_plugin/KDLKinematicsPlugin
```

Test the Plugin

Use the MoveIt Rviz Motion Planning Plugin and use the interactive markers to see if correct IK Solutions are found.

Updating the Plugin

If any future changes occur with MoveIt! or IKFast, you might need to re-generate this plugin using our scripts. To allow you to easily do this, a bash script is automatically created in the root of your IKFast package, named *update_ikfast_plugin.sh*. This does the same thing you did manually earlier, but uses the IKFast solution header file that is copied into the ROS package.

3.5 Trac-IK Kinematics Solver

Trac-IK <https://bitbucket.org/traclabs/trac_ik> is an inverse kinematics solver developed by Traclabs that combines two IK implementations via threading to achieve more reliable solutions than common available open source IK solvers. From their documentation:

> (Trac-IK) provides an alternative Inverse Kinematics solver to the popular inverse Jacobian methods in KDL. Specifically, KDL's convergence algorithms are based on Newton's method, which does not work well in the presence of joint limits — common for many robotic platforms. TRAC-IK concurrently runs two IK implementations. One is a simple extension to KDL's Newton-based convergence algorithm that detects and mitigates local minima due to joint limits by random jumps. The second is an SQP (Sequential Quadratic Programming) nonlinear optimization approach which uses quasi-Newton methods that better handle joint limits. By default, the IK search returns immediately when either of these algorithms converges to an answer. Secondary constraints of distance and manipulability are also provided in order to receive back the "best" IK solution.

The package *trac_ik_kinematics_plugin <https://bitbucket.org/traclabs/trac_ik/src/HEAD/trac_ik_kinematics_plugin/>* provides a KinematicsBase MoveIt! interface that can replace the default KDL solver. Currently mimic joints are *not* supported.

3.5.1 Install

As of v1.4.3, **trac_ik** is part of the ROS Indigo/Jade binaries:

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

3.5.2 Usage

- Install **trac_ik_kinematics_plugin** and **trac_ik_lib package** or add to your catkin workspace.
- Find the MoveIt! *kinematics.yaml <http://docs.ros.org/kinetic/api/moveit_tutorials/html/doc/pr2_tutorials/kinematics/src/doc/kine* file created for your robot.
- Replace `kinematics_solver:  kdl_kinematics_plugin/KDLKinematicsPlugin` (or similar) with `kinematics_solver:  trac_ik_kinematics_plugin/TRAC_IKKinematicsPlugin`
- Set parameters as desired: - **kinematics_solver_timeout** (timeout in seconds, e.g., 0.005) and **position_only_ik ARE** supported. - **solve_type** can be Speed, Distance, Manipulation1, Manipulation2 (see trac_ik_lib documentation for details). Default is Speed. - **kinematics_solver_attempts** parameter is unneeded: unlike KDL, TRAC-IK solver already restarts when it gets stuck - **kinematics_solver_search_resolution** is not applicable here. - Note: The Cartesian error distance used to determine a valid solution is **1e-5**, as that is what is hard-coded into MoveIt's KDL plugin.

CHAPTER 4

Configuration

4.1 Kinematics Configuration Tutorial

In this section, we will examine some of the parameters for configuring kinematics for your robot.

4.1.1 The kinematics.yaml file

The kinematics.yaml file generated by the MoveIt! Setup Assistant is the primary configuration file for kinematics for MoveIt!. You can see an entire example file for the PR2 robot in the moveit_pr2 github project

```
right_arm:
  kinematics_solver: pr2_arm_kinematics/PR2ArmKinematicsPlugin
  kinematics_solver_search_resolution: 0.001
  kinematics_solver_timeout: 0.05
  kinematics_solver_attempts: 3
right_arm_and_torso:
  kinematics_solver: kdl_kinematics_plugin/KDLKinematicsPlugin
  kinematics_solver_search_resolution: 0.005
  kinematics_solver_timeout: 0.05
```

Parameters

The set of available parameters include:

- *kinematics_solver*: The name of your kinematics solver plugin. Note that this must match the name that you specified in the plugin description file, e.g. `example_kinematics/ExampleKinematicsPlugin`
- *kinematics_solver_search_resolution*: This specifies the resolution that a solver might use to search over the redundant space for inverse kinematics, e.g. using one of the joints for a 7 DOF arm specified as the redundant joint.
- *kinematics_solver_timeout*: This is a default timeout specified (in seconds) for each internal iteration that the inverse kinematics solver may perform. A typical iteration (e.g. for a numerical solver) will consist of a random restart from a seed state followed by a solution cycle (for which this timeout is applicable). The solver may attempt multiple restarts - the default number of restarts is defined by the kinematics_solver_attempts parameter below.
- *kinematics_solver_attempts*: The number of random restarts that will be performed on the solver. Each solution cycle after the restart will have a timeout defined by the kinematics_solver_timeout parameter above. In general, it is better to set this timeout low and fail quickly in an individual solution cycle.

The KDL Kinematics Plugin

The KDL kinematics plugin wraps around the numerical inverse kinematics solver provided by the Orocos KDL package.

- This is the default kinematics plugin currently used by MoveIt!
- It obeys joint limits specified in the URDF (and will use the safety limits if they are specified in the URDF).
- The KDL kinematics plugin currently only works with serial chains.

The LMA Kinematics Plugin

The LMA (Levenberg-Marquardt) kinematics plugin also wraps around a numerical inverse kinematics solver provided by the

- It obeys joint limits specified in the URDF (and will use the safety limits if they are specified in the URDF).
- The LMA kinematics plugin currently only works with serial chains.
- Usage: `kinematics_solver:  lma_kinematics_plugin/LMAKinematicsPlugin`

4.1.2 Position Only IK

Position only IK can easily be enabled (only if you are using the KDL Kinematics Plugin) by adding the following line to your kinematics.yaml file (for the particular group that you want to solve IK for):

```
position_only_ik: True
```

4.2 OMPL Interface

4.2.1 OMPL Optimization Objective Tutorial

Several planners that are part of the OMPL planning library are capable of optimizing for a specified optimization objective. This tutorial describes that steps that are needed to configure these objectives. The optimal planners that are currently exposed to MoveIt! are:

- geometric::RRTstar
- geometric::PRMstar

And the following optimization objectives are available:

- PathLengthOptimizationObjective (Default)
- MechanicalWorkOptimizationObjective
- MaximizeMinClearanceObjective
- StateCostIntegralObjective
- MinimaxObjective

The configuration of these optimization objectives can be done in the *ompl_planning.yaml*. A parameter with the name **optimization_objective** is added as a configuration parameter. The value of the parameter is set to be the name of the selected optimization objective. For example, to configure RRTstar to use the *MaximizeMinClearanceObjective*, the planner entry in the ompl_planning.yaml will look like:

```
RRTstarkConfigDefault:
    type: geometric::RRTstar
    optimization_objective: MaximizeMinClearanceObjective
    range: 0.0
    goal_bias: 0.05
    delay_collision_checking: 1
```

For more information on the OMPL optimal planners, the reader is referred to the OMPL - Optimal Planning documentation.

4.3 CHOMP Interface

Note: *The chomp planner has not been tested extensively yet.*

4.3.1 Run Generic Demo for Fanuc M-10iA

To run the demo you'll need the moveit_resources package.

Once you have this package simply run:

```
roslaunch moveit_resources demo_chomp.launch
```

4.3.2 Assumptions:

1. You have the latest version of moveit installed. On ROS kinetic you may need to build it from source.
2. You have a moveit configuration package for your robot already. For example, if you have a Kinova Jaco arm, it's probably called "jaco_moveit_config". This is typically built using the Moveit Setup Assistant.
3. Lets assume that you are using the **jaco** manipulator. And hence, the moveit config package is *jaco_moveit_config*.

4.3.3 Using CHOMP with your own robot

1. Simply download chomp_planning_pipeline.launch.xml file into the launch directory of your moveit config package. So into the *jaco_moveit_config/launch* directory.
2. Copy the *demo.launch* file to *demo_chomp.launch*. Note that this file is also in the launch directory of the *jaco_moveit_config* package.
3. Find the lines where *move_group.launch* is included and change it to:

```
    <include file="$(find jaco_moveit_config)/launch/move_group.launch">
      <arg name="allow_trajectory_execution" value="true"/>
      <arg name="fake_execution" value="true"/>
      <arg name="info" value="true"/>
      <arg name="debug" value="$(arg debug)"/>
      <arg name="planner" value="chomp" />
    </include>
```

You probably only need to change the planner arg to chomp.

4. Run the demo:

```
    roslaunch jaco_moveit_config demo_chomp.launch
```

CHAPTER 5

Attribution

The original MoveIt! tutorials were created by Sachin Chitta, Dave Hershberger, and Acorn Pooley at SRI International. Further improvements have been made by Dave Coleman, Michael Gorner, and Francisco Suarez. Please help us improve these docs!