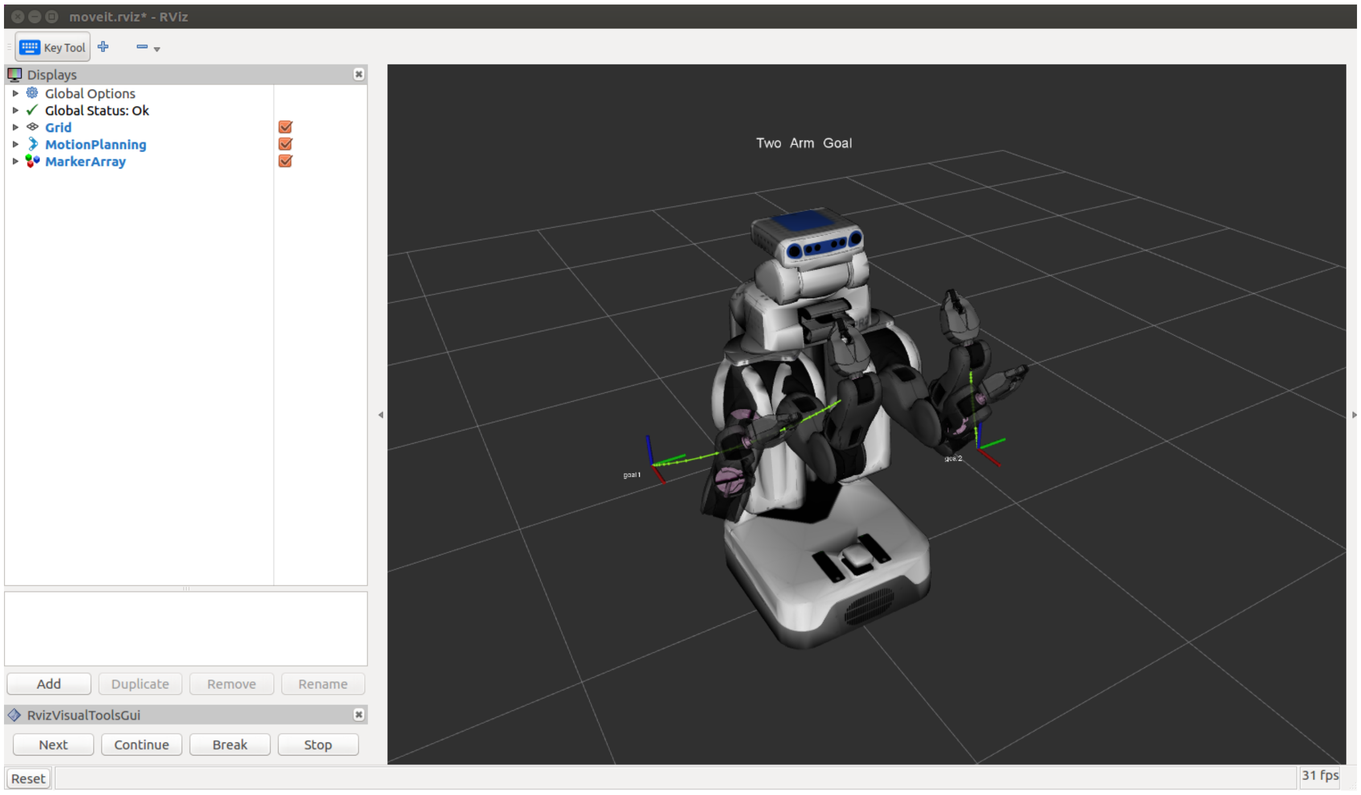
Move Group Interface Tutorial

In MoveIt!, the primary user interface is through the MoveGroupInterface 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. This interface communicates over ROS topics, services, and actions to the MoveGroup Node.



Watch the YouTube video demo

Compiling the example code

You do not need to build all of MoveIt! from source, but you can follow similar instructions as documented on the MoveIt! source install page for setting up your catkin workspace. Within your catkin workspace, download this example:

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

Temporary PR2 on Kinetic Instructions

You will also need a **pr2_moveit_config** package to run this tutorial. Currently this is unreleased in ROS Kinetic but the following is a temporary workaround:

```
git clone https://github.com/PR2/pr2_common.git -b kinetic-devel
git clone https://github.com/davetcoleman/pr2_moveit_config.git
rosdep install --from-paths . --ignore-src --rosdistro kinetic
```

Start Rviz

Start Rviz and wait for everything to finish loading:

```
roslaunch pr2_moveit_config demo.launch
```

Running the demo

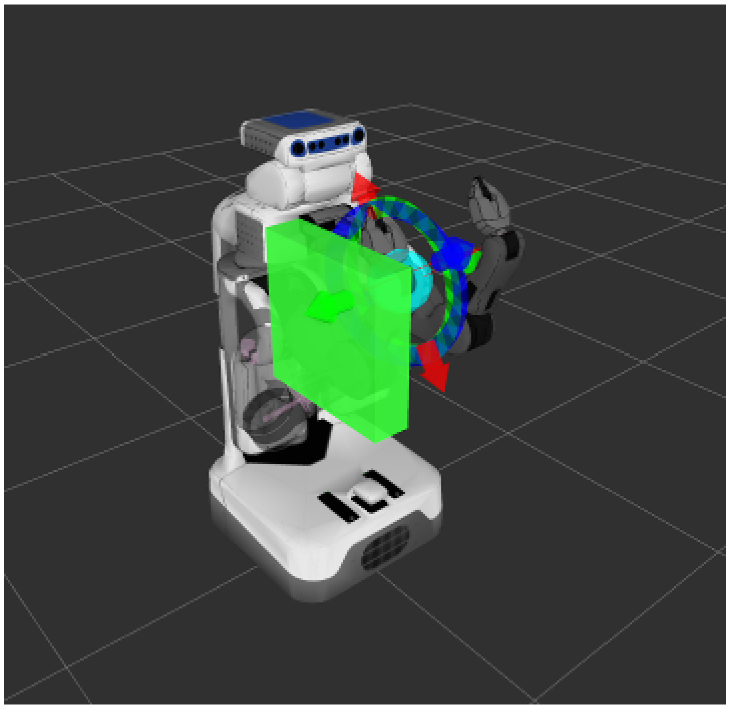
In a new window, run the move_group_interface_tutorial.launch roslaunch file:

```
roslaunch moveit_tutorials move_group_interface_tutorial.launch
```

After a short moment, the Rviz window should appear and look similar to the one at the top of this page. Press the **Next** button at the bottom of the screen or press ‘N’ on your keyboard while Rviz is focused to progress through each demo step.

Expected Output

Watch the YouTube video demo for expected output. In Rviz, we should be able to see the following:

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

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

Explaining the Demo

The entire code is located in the moveit_tutorials github repo under the subfolder pr2_tutorials/planning. Next we step through the code piece by piece to explain its functionality.

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 MoveGroupInterface 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);
```

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

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

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);
```

Visualization

The package MoveItVisualTools provides many capabilties for visualizing objects, robots, and trajectories in Rviz as well as debugging tools such as step-by-step introspection of a script

```
namespace rvt = rviz_visual_tools;
moveit_visual_tools::MoveItVisualTools visual_tools("odom_combined");
visual_tools.deleteAllMarkers();
```

Remove control is an introspection tool that allows users to step through Rviz via buttons and keyboard shortcuts in Rviz

```
visual_tools.loadRemoteControl();
```

Rviz provides many types of markers, in this demo we will use text, cylinders, and spheres

```
Eigen::Affine3d text_pose = Eigen::Affine3d::Identity();
text_pose.translation().z() = 1.75; // above head of PR2
visual_tools.publishText(text_pose, "MoveGroupInterface Demo", rvt::WHITE, rvt::XLARGE);
```

Batch publishing is used to reduce the number of messages being sent to Rviz for large visualizations

```
visual_tools.trigger();
```

Getting Basic Information

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

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

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

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

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

Visualizing plans

We can also visualize the plan as a line with markers in Rviz.

```
ROS_INFO_NAMED("tutorial", "Visualizing plan 1 as trajectory line");
visual_tools.publishAxisLabeled(target_pose1, "pose1");
visual_tools.publishText(text_pose, "Pose Goal", rvt::WHITE, rvt::XLARGE);
visual_tools.publishTrajectoryLine(my_plan.trajectory_, joint_model_group);
visual_tools.trigger();
visual_tools.prompt("next step");
```

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() */
```

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_NAMED("tutorial", "Visualizing plan 2 (joint space goal) %s", success ? "" :
"FAILED");
```

Visualize the plan in Rviz

```
visual_tools.deleteAllMarkers();
visual_tools.publishText(text_pose, "Joint Space Goal", rvt::WHITE, rvt::XLARGE);
visual_tools.publishTrajectoryLine(my_plan.trajectory_, joint_model_group);
visual_tools.trigger();
visual_tools.prompt("next step");
```

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);
```

Planning with constraints can be slow because every sample must call and inverse kinematics solver. Lets increase the planning time from the default 5 seconds to be sure the planner has enough time to succeed.

```
move_group.setPlanningTime(10.0);

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

Visualize the plan in Rviz

```
visual_tools.deleteAllMarkers();
visual_tools.publishAxisLabeled(start_pose2, "start");
visual_tools.publishAxisLabeled(target_pose1, "goal");
visual_tools.publishText(text_pose, "Constrained Goal", rvt::WHITE, rvt::XLARGE);
visual_tools.publishTrajectoryLine(my_plan.trajectory_, joint_model_group);
visual_tools.trigger();
visual_tools.prompt("next step");
```

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

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

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 but adding it can help with visualizations

```
std::vector<geometry_msgs::Pose> waypoints;
waypoints.push_back(start_pose2);

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

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

target_pose3.position.z -= 0.2;
target_pose3.position.y += 0.2;
target_pose3.position.x -= 0.2;
waypoints.push_back(target_pose3);  // down and right (back to start)
```

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

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

We want the cartesian path to be interpolated at a resolution of 1 cm which is why we will specify 0.01 as the max step in cartesian translation. We will specify the jump threshold as 0.0, effectively disabling it. Warning - disabling the jump threshold while operating real hardware can cause large unpredictable motions of redundant joints and could be a safety issue

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

Visualize the plan in Rviz

```
visual_tools.deleteAllMarkers();
visual_tools.publishText(text_pose, "Joint Space Goal", rvt::WHITE, rvt::XLARGE);
visual_tools.publishPath(waypoints, rvt::LIME_GREEN, rvt::SMALL);
for (std::size_t i = 0; i < waypoints.size(); ++i)
  visual_tools.publishAxisLabeled(waypoints[i], "pt" + std::to_string(i), rvt::SMALL);
visual_tools.trigger();
visual_tools.prompt("next step");
```

Adding/Removing Objects and Attaching/Detaching Objects

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;
```

efine 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_NAMED("tutorial", "Add an object into the world");
planning_scene_interface.addCollisionObjects(collision_objects);
```

Show text in Rviz of status

```
visual_tools.publishText(text_pose, "Add object", rvt::WHITE, rvt::XLARGE);
visual_tools.trigger();
```

Sleep so we have time to see the object in RViz

```
ros::Duration(1.0).sleep();
```

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_NAMED("tutorial", "Visualizing plan 5 (pose goal move around cuboid) %s",
success ? "" : "FAILED");
```

Visualize the plan in Rviz

```
visual_tools.deleteAllMarkers();
visual_tools.publishText(text_pose, "Obstacle Goal", rvt::WHITE, rvt::XLARGE);
visual_tools.publishTrajectoryLine(my_plan.trajectory_, joint_model_group);
visual_tools.trigger();
visual_tools.prompt("next step");
```

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

```
ROS_INFO_NAMED("tutorial", "Attach the object to the robot");
move_group.attachObject(collision_object.id);
```

Show text in Rviz of status

```
visual_tools.publishText(text_pose, "Object attached to robot", rvt::WHITE, rvt::XLARGE);
visual_tools.trigger();

/* Sleep to give Rviz time to show the object attached (different color).*/
ros::Duration(1.0).sleep();
```

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

```
ROS_INFO_NAMED("tutorial", "Detach the object from the robot");
move_group.detachObject(collision_object.id);
```

Show text in Rviz of status

```
visual_tools.publishText(text_pose, "Object dettached from robot", rvt::WHITE,
rvt::XLARGE);
visual_tools.trigger();

/* Sleep to give Rviz time to show the object detached.*/
ros::Duration(1.0).sleep();
```

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

```
ROS_INFO_NAMED("tutorial", "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);
```

Show text in Rviz of status

```
visual_tools.publishText(text_pose, "Object removed", rvt::WHITE, rvt::XLARGE);
visual_tools.trigger();

/* Sleep to give Rviz time to show the object is no longer there.*/
ros::Duration(1.0).sleep();
```

Dual-arm pose goals

First define a new group for addressing the two arms.

```
static const std::string PLANNING_GROUP2 = "arms";
moveit::planning_interface::MoveGroupInterface two_arms_move_group(PLANNING_GROUP2);
```

Define two separate pose goals, one for each end-effector. Note that we are reusing the goal for the right arm above

```
two_arms_move_group.setPoseTarget(target_pose1, "r_wrist_roll_link");

geometry_msgs::Pose target_pose4;
target_pose4.orientation.w = 1.0;
target_pose4.position.x = 0.7;
target_pose4.position.y = 0.15;
target_pose4.position.z = 1.0;

two_arms_move_group.setPoseTarget(target_pose4, "l_wrist_roll_link");
```

Now, we can plan and visualize

```
moveit::planning_interface::MoveGroupInterface::Plan two_arms_plan;

success = two_arms_move_group.plan(two_arms_plan);
ROS_INFO_NAMED("tutorial", "Visualizing plan 7 (dual arm plan) %s", success ? "" :
"FAILED");
```

Visualize the plan in Rviz

```
visual_tools.deleteAllMarkers();
visual_tools.publishAxisLabeled(target_pose1, "goal1");
visual_tools.publishAxisLabeled(target_pose4, "goal2");
visual_tools.publishText(text_pose, "Two Arm Goal", rvt::WHITE, rvt::XLARGE);
joint_model_group = move_group.getCurrentState()->getJointModelGroup(PLANNING_GROUP2);
visual_tools.publishTrajectoryLine(two_arms_plan.trajectory_, joint_model_group);
visual_tools.trigger();
```