6
Using the ROS MoveIt! and Navigation Stack

In the previous chapters, we have been discussing the design and simulation of a robotic arm and mobile robot. We controlled each joint of the robotic arm in Gazebo using the ROS controller and moved the mobile robot inside Gazebo using the teleop node.

In this chapter, we are going to address the *motion planning* problem. Moving a robot by directly controlling its joints manually might be a difficult task, especially if we want to add position or velocity constraints to the robot motion. Similarly, driving a mobile robot, avoiding obstacles, requires the planning of a path. For this reason, we will solve these problems using the ROS **MoveIt!** and **Navigation stack**.

MoveIt! is a set of packages and tools for doing mobile manipulation in ROS. The official web page (`http://moveit.ros.org/`) contains the documentations, the list of robots using MoveIt!, and various examples to demonstrate pick and place, grasping, simple motion planning using inverse kinematics, and so on.

MoveIt! contains state-of-the-art software for motion planning, manipulation, 3D perception, kinematics, collision checking, control, and navigation. Apart from the command line interface, MoveIt! has some good GUI to interface a new robot to MoveIt!. Also, there is a RViz plugin, which enables motion planning from RViz itself. We will also see how to motion plan our robot using MoveIt! C++ APIs.

Next is the Navigation stack, another set of powerful tools and libraries to work mainly for mobile robot navigation. The Navigation stack contains ready-to-use navigation algorithms which can be used in mobile robots, especially for differential wheeled robots. Using these stacks, we can make the robot autonomous, and that is the final concept that we are going to see in the Navigation stack.

The first section of this chapter will discuss more on the MoveIt! package, installation, and architecture. After discussing the main concepts of MoveIt!, we will see how to create a MoveIt! package for our robotic arm, which can provide collision-aware path planning to our robot. Using this package, we can perform motion planning (inverse kinematics) in RViz, and can interface to Gazebo or the real robot for executing the paths.

After discussing the interfacing, we will discuss more about the Navigation stack and see how to perform autonomous navigation using **SLAM** (**Simultaneous Localization And Mapping**) and **amcl** (**Adaptive Monte Carlo Localization**).

Installing MoveIt!

Let's start with installing MoveIt!. The installation procedure is very simple and is just a single command. Using the following commands, we install the MoveIt! core, a set of plugins ad planners for ROS Kinetic:

```
$ sudo apt-get install ros-kinetic-moveit ros-kinetic-moveit-plugins ros-
kinetic-moveit-planners
```

MoveIt! architecture

Let's start with MoveIt! and its architecture. Understanding the architecture of MoveIt! helps to program and interface the robot to MoveIt!. We will quickly go through the architecture and the important concepts of MoveIt!, and start interfacing and programming our robots.

Here is the MoveIt! architecture, included in their official web page, at `http://moveit.ros.org/documentation/concepts`:

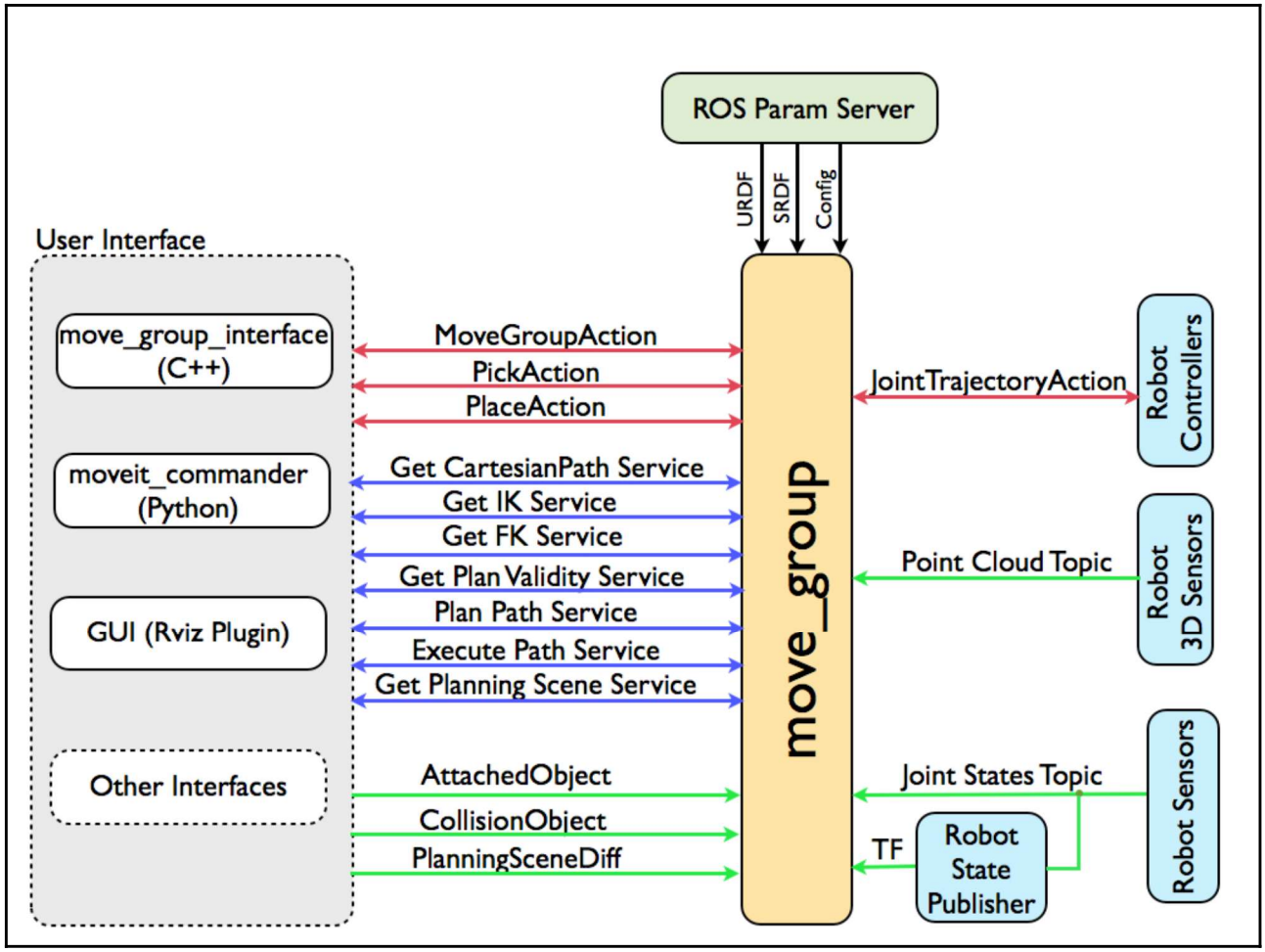

Figure 1: MoveIt! architecture diagram

The move_group node

We can say that `move_group` is the heart of MoveIt!, as this node acts as an integrator of the various components of the robot and delivers actions/services according to the user's needs.

From the architecture, it's clear that the `move_group` node collects robot information such as point cloud, joint state of the robot, and transform (TF) of the robot in the form of topics and services.

From the parameter server, it collects the robot kinematics data, such as `robot_description` (URDF), **Semantic Robot Description Format** (**SRDF**), and the configuration files. The SRDF file and the configuration files are generated while we generate a MoveIt! package for our robot. The configuration files contains the parameter file for setting joint limits, perception, kinematics, end effector, and so on. We will see the files when we discuss generating the MoveIt! package for our robot.

When MoveIt! gets all this information about the robot and its configuration, we can say it is properly configured and we can start commanding the robot from the user interfaces. We can either use C++ or Python MoveIt! APIs to command the `move_group` node to perform actions such as pick/place, IK, and FK, among others. Using the RViz motion planning plugin, we can command the robot from the RViz GUI itself.

As we already discussed, the `move_group` node is an integrator; it does not run any kind of motion planning algorithms directly, but instead connects all the functionalities as plugins. There are plugins for kinematics solvers, motion planning, and so on. We can extend the capabilities through these plugins.

After motion planning, the generated trajectory talks to the controllers in the robot using the `FollowJointTrajectoryAction` interface. This is an action interface in which an action server is run on the robot, and `move_node` initiates an action client which talks to this server and executes the trajectory on the real robot/Gazebo simulator.

At the end of the MoveIt! discussion, we will see how to connect MoveIt! with RViz GUI to Gazebo. The following screenshot shows a robotic arm that is controlling from RViz and the trajectory is executed inside Gazebo:

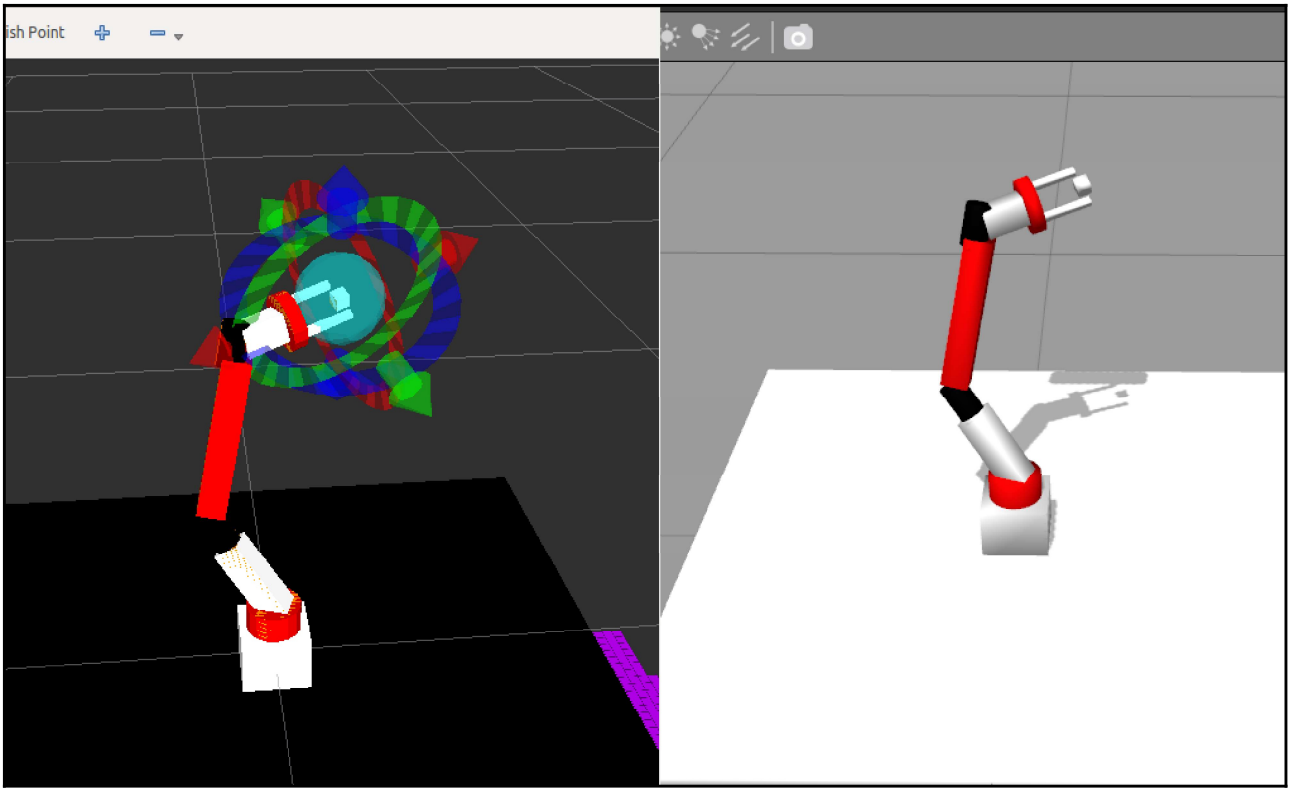

Figure 2: Trajectory from RViz GUI executing in Gazebo

Motion planning using MoveIt!

Assume that we know the starting pose of the robot, a desired goal pose of the robot, the geometrical description of the robot, and geometrical description of the world, then motion planning is the technique to find an optimum path that moves the robot gradually from the start pose to the goal pose, while never touching any obstacles in the world and without colliding with the robot links.

In this context, the robot geometry is described via the URDF file. We can also create a description file for the robot environment and use laser or vision sensors of the robot to map its operative space, in order to avoid static and dynamic obstacles during the execution of planned paths.

In the case of the robotic arm, the motion planner should find a trajectory (consisting of joint spaces of each joint) in which the links of the robot should never collide with the environment, avoid self-collision (collision between two robot links), and not violate the joint limits.

MoveIt! can talk to the motion planners through the plugin interface. We can use any motion planner by simply changing the plugin. This method is highly extensible so we can try our own custom motion planners using this interface. The move_group node talks to the motion planner plugin via the ROS action/services. The default planner for the `move_group` node is OMPL (`http://ompl.kavrakilab.org/`).

To start motion planning, we should send a motion planning request to the motion planner which specified our planning requirements. The planning requirement may be setting a new goal pose of the end-effector; for example, for a pick and place operation.

We can set additional kinematic constraints for the motion planners. The following are some inbuilt constraints in MoveIt!:

- **Position constraints**: These restrict the position of a link
- **Orientation constraints**: These restrict the orientation of a link
- **Visibility constraints**: These restrict a point on the link to be visible in an area (view of a sensor)
- **Joint constraints**: These restrict a joint within its joint limits
- **User-specified constraints**: Using these constraints, the user can define his own constraints using the callback functions

Using these constraints, we can send a motion planning request and the planner will generate a suitable trajectory according to the request. The `move_group` node will generate the suitable trajectory from the motion planner which obeys all the constraints. This can be sent to robot joint trajectory controllers.

Motion planning request adapters

The planning request adapters help to pre-process the motion planning request and post-process the motion planning response. One of the uses of pre-processing requests is that it helps to correct if there is a violation in the joint states and, for the post-processing, it can convert the path generated by the planner to a time-parameterized trajectory. The following are some of the default planning request adapters in MoveIt!:

- **FixStartStateBounds**: If a joint state is slightly outside the joint limits, then this adapter can fix the initial joint limits within the limits.
- **FixWorkspaceBounds**: This specifies a workspace for planning with a cube size of 10 m x 10 m x 10 m.
- **FixStartStateCollision**: This adapter samples a new collision free configuration if the existing joint configuration is in collision. It makes a new configuration by changing the current configuration by a small factor called `jiggle_factor`.
- **FixStartStatePathConstraints**: This adapter is used when the initial pose of the robot does not obey the path constraints. In this, it finds a near pose which satisfies the path constraints and uses that pose as the initial state.
- **AddTimeParameterization**: This adapter parameterizes the motion plan by applying the velocity and acceleration constraints.

MoveIt! planning scene

The term "planning scene" is used to represent the world around the robot and store the state of the robot itself. The planning scene monitor inside `move_group` maintains the planning scene representation. The `move_group` node consists of another section called the world geometry monitor, which builds the world geometry from the sensors of the robot and from the user input.

The planning scene monitor reads the joint_states topic from the robot, and the sensor information and world geometry from the world geometry monitor. The world scene monitor reads from the occupancy map monitor, which uses 3D perception to build a 3D representation of the environment, called **Octomap**. Octomaps can be generated from point clouds, which are handled by a point cloud occupancy map update plugin and depth images handled by a depth image occupancy map updater. The following image shows the representation of the planning scene from the MoveIt! official wiki (`http://moveit.ros.org/documentation/concepts/`):

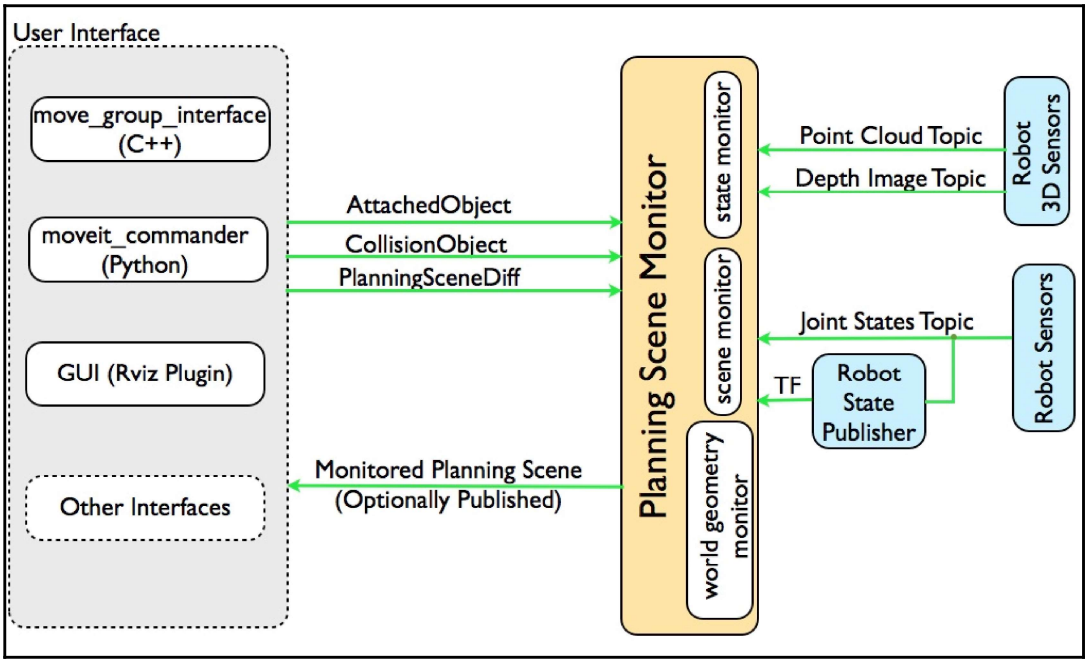

Figure 3: MoveIt! planning scene overview diagram.

MoveIt! kinematics handling

MoveIt! provides a great flexibility to switch the inverse kinematics algorithms using the robot plugins. Users can write their own IK solver as a MoveIt! plugin and switch from the default solver plugin whenever required. The default IK solver in MoveIt! is a numerical jacobian-based solver.

Compared to the analytic solvers, the numerical solver can take time to solve IK. The package called IKFast can be used to generate a C++ code for solving IK using analytical methods, which can be used for different kinds of robot manipulator and perform better if the DOF is less than 6. This C++ code can also be converted into the MoveIt! plugin by using some ROS tool. We will look at this procedure in the upcoming chapters.

Forward kinematics and finding jacobians are already integrated to the MoveIt! RobotState class, so we don't need to use plugins for solving FK.

MoveIt! collision checking

The CollisionWorld object inside MoveIt! is used to find collisions inside a planning scene which is using the **Flexible Collision Library** (**FCL**) package as a backend. MoveIt! supports collision checking for different types of objects, such as meshes, primitive shapes such as boxes, cylinders, cones, spheres, and Octomap.

Collision checking is one of the computationally expensive tasks during motion planning. To reduce this computation, MoveIt! provides a matrix called **ACM** (**Allowed Collision Matrix**). It contains a binary value corresponding to the need to check for a collision between two pairs of bodies. If the value of the matrix is 1, it means collision of the corresponding pair is not needed. We can set the value as 1 where the bodies are always so far that they would never collide with each other. Optimizing ACM can reduce the total computation needed for collision avoidance.

After discussing the basic concepts in MoveIt!, we can now discuss how to interface a robotic arm into MoveIt!. To interface a robot arm in MoveIt!, we need to satisfy the components that we saw in Figure 1. The `move_group` node essentially requires parameters, such as URDF, SRDF, config files, and joint states topics, along with TF from a robot to start with motion planning.

MoveIt! provides a GUI-based tool called Setup Assistant to generate all these elements. The following section describes the procedure to generate a MoveIt! configuration from the Setup Assistant tool.

Generating MoveIt! configuration package using the Setup Assistant tool

The MoveIt! Setup Assistant is a graphical user interface for configuring any robot to MoveIt!. Basically, this tool generates SRDF, configuration files, launch files, and scripts generating from the robot URDF model, which is required to configure the `move_group` node.

The SRDF file contains details about the arm joints, end-effector joints, virtual joints, and the collision link pairs, which are configured during the MoveIt! configuration process using the Setup Assistant tool.

The configuration file contains details about the kinematic solvers, joint limits, controllers, and so on, which are also configured and saved during the configuration process.

Using the generated configuration package of the robot, we can work with motion planning in RViz without the presence of a real robot or simulation interface.

Let's start the configuration wizard, and we can see the step-by-step procedure to build the configuration package of our robotic arm.

Step 1 – Launching the Setup Assistant tool

To start the MoveIt! Setup Assistant tool, we can use the following command:

```
$ roslaunch moveit_setup_assistant setup_assistant.launch
```

This will bring up a window with two choices: **Create New MoveIt! Configuration Package** or **Edit Existing MoveIt! Configuration Package**. Here we are creating a new package, so we need that option. If we have a MoveIt! package already, then we can select the second option.

Click on the **Create New MoveIt! Configuration Package button**, which will display a new screen, as shown next:

Figure 4: MoveIt! Setup Assistant

In this step, the wizard asks for the URDF model of the new robot. To give the URDF file, click on the **Browse** button and navigate to `mastering_ros_robot_description_pkg/urdf/ seven_dof_arm.urdf`. Choose this file and press the **Load** button to load the URDF. We can either give the robot model as pure URDF or xacro; if we give xacro, the tool will convert to RDF internally.

If the robot model is successfully parsed, we can see the robot model on the window, as shown in the following screenshot:

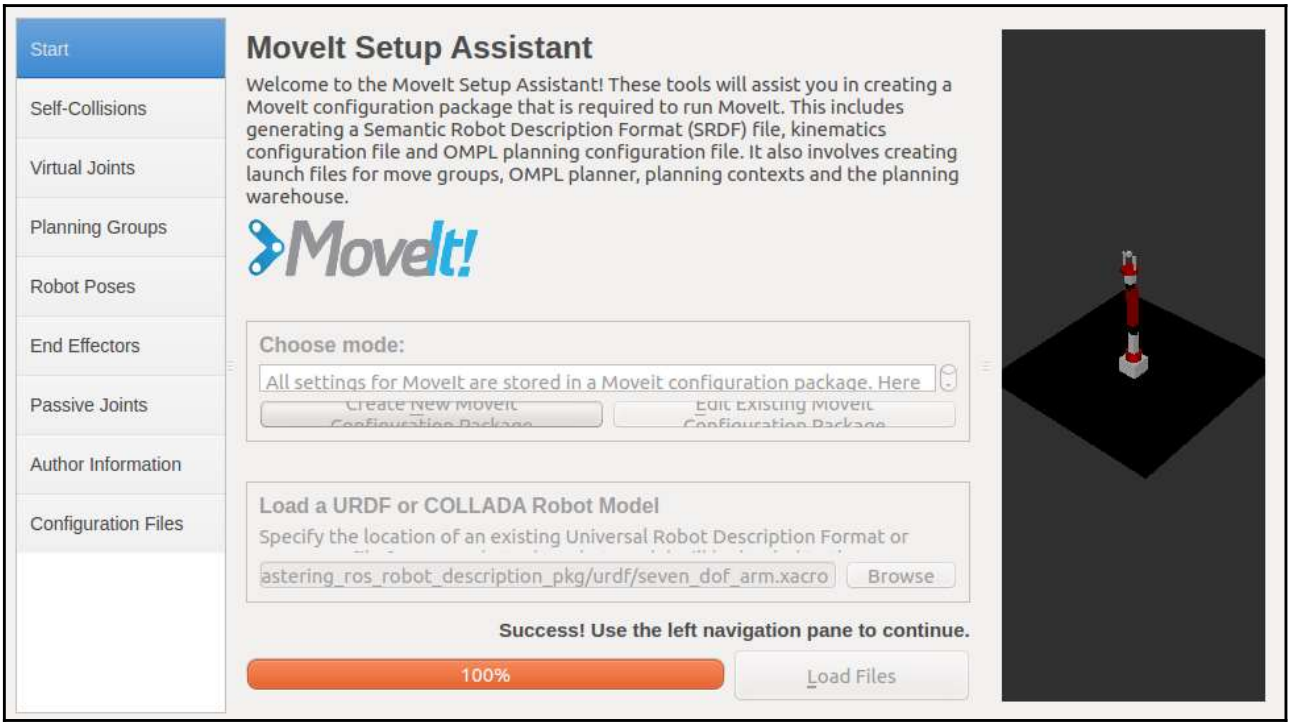

Figure 5: Successfully parsing the robot model in the Setup Assistant tool

Step 2 – Generating the Self-Collision matrix

We can now start to navigate all the panels of the window to properly configure our robot. In the **Self-Collisions** tab, MoveIt! searches for a pair of links on the robot which can be safely disabled from the collision checking. These can reduce the processing time. This tool analyzes each link pair and categorizes the links as always in collision, never in collision, default in collision, adjacent links disabled, and sometimes in collision, and it disables the pair of links which makes any kind of collision. The following image shows the **Self-Collisions** window:

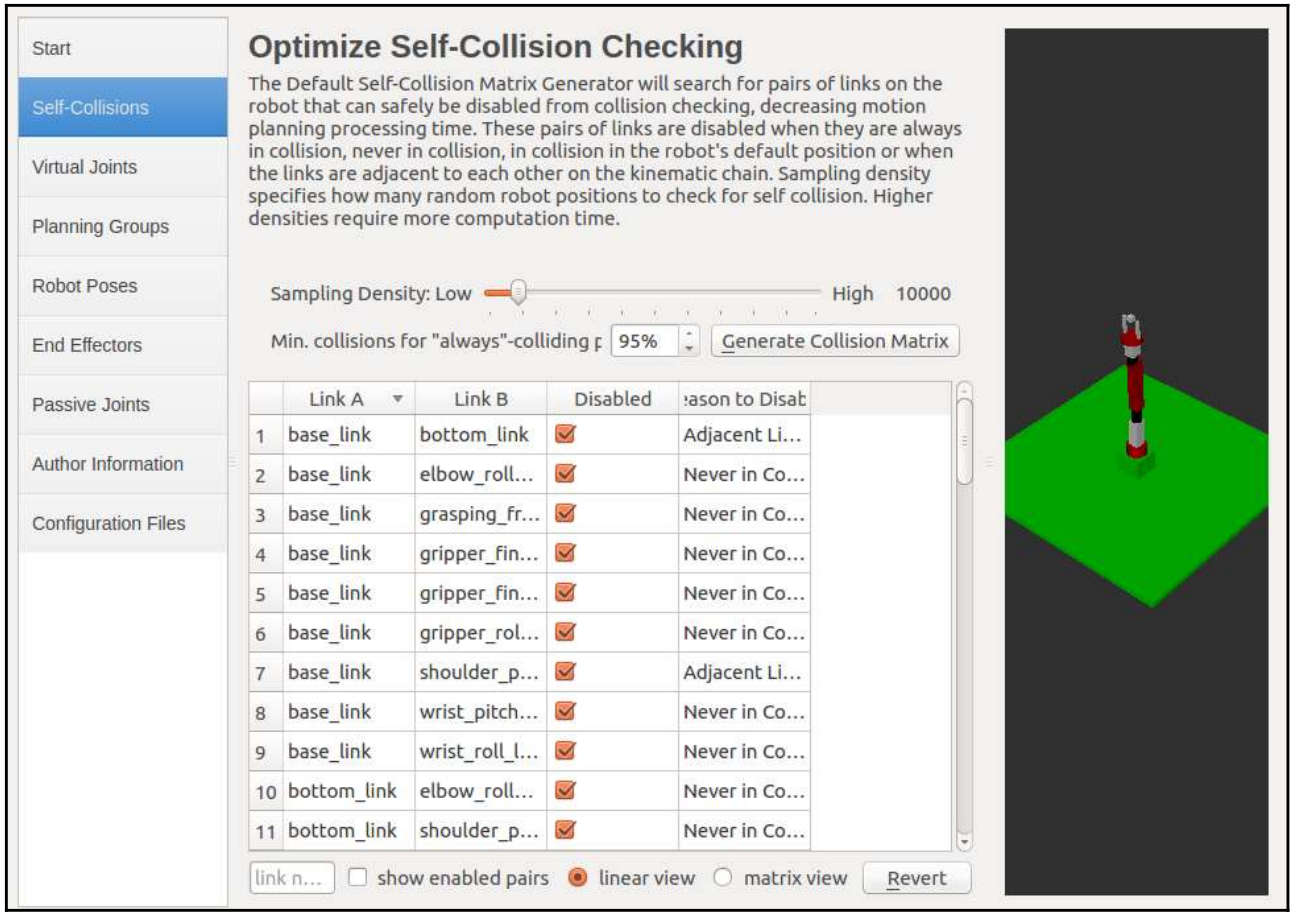

Figure 6: Regenerating the Self-Collision matrix

The sampling density is the number of random positions to check for self-collision. If the density is large, computation will be high but self-collision will be less. The default value is 10,000. We can see the disabled pair of links by pressing the **Regenerate Default Collision Matrix** button; it will take a few seconds to list out the disabled pair of links.

Step 3 – Adding virtual joints

Virtual joints attach the robot to the world. They are not mandatory for a static robot which does not move. We need virtual joints when the base position of the arm is not fixed. For example, if a robot arm is fixed on a mobile robot, we should define a virtual joint with respect to the odometry frame (`odom`).

In the case of our robot, we are not creating virtual joints.

Step 4 – Adding planning groups

A planning group is basically a group of joints/links in a robotic arm which plans together to achieve a goal position of a link or the end effector. We must create two planning groups, one for the arm and one for the gripper.

Click on the **Planning Groups** tab on the left side and click on the **Add Group** button. You will see the following screen, which has the settings of the `arm` group:

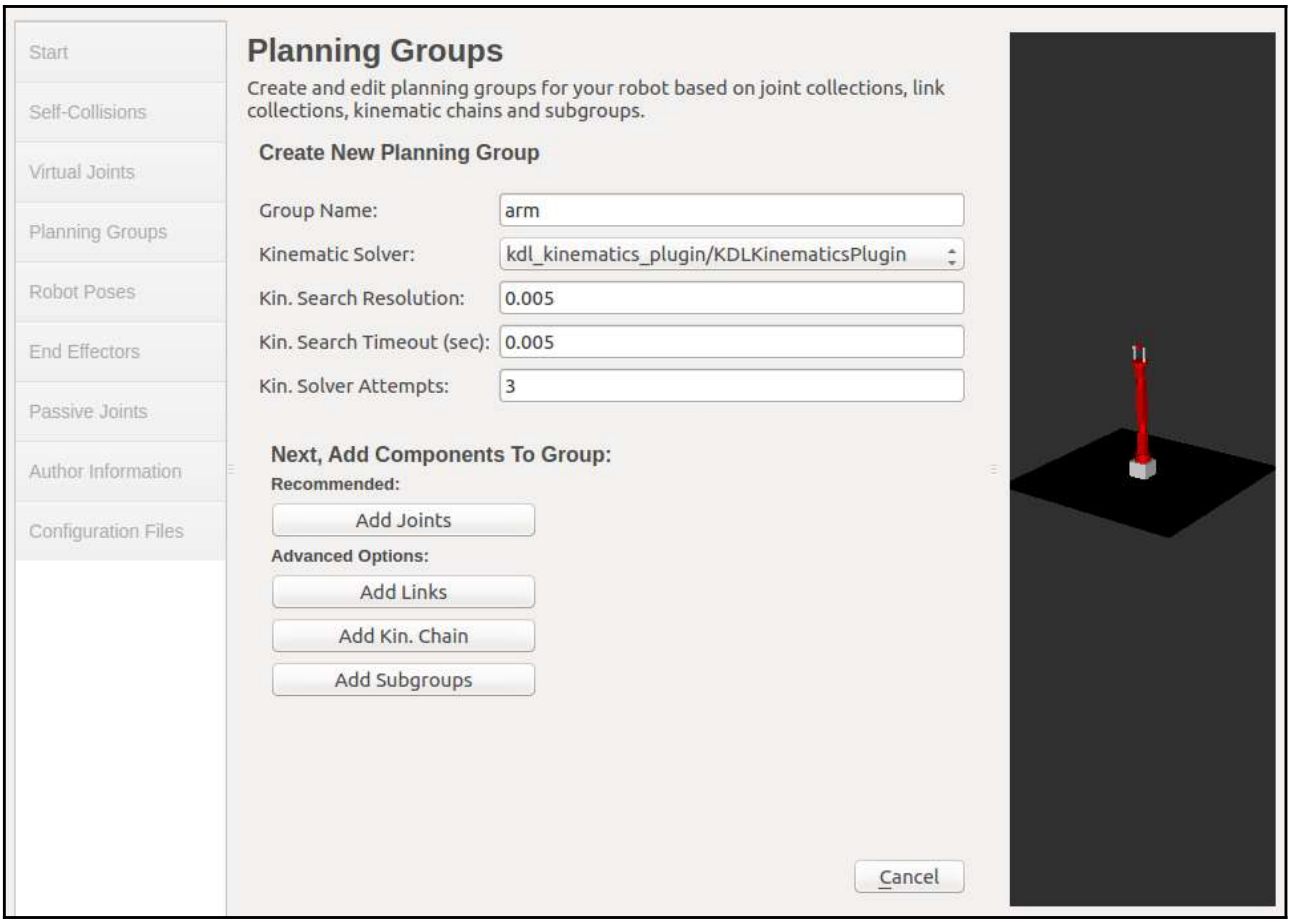

Figure 7: Adding the planning group of the arm

Here, we are giving **Group Name** as arm, and **Kinematic Solver** as `kdl_kinematics_plugin/KDLKinematicsPlugin`, which is the default numerical IK solver with MoveIt!. We can keep the other parameters as the default values. In addition, we can choose different ways to add elements in a planning group. For example, we could specify the joints of the group, add its links, or directly specify a kinematic chain.

Inside the arm group, first we have to add a kinematic chain, starting from `base_link` as the first link to the `grasping_frame`.

Add a group called `gripper` and we don't need to have a kinematic solver for the `gripper` group. Inside this group, we can add the joints and links of the gripper. These settings are shown next:

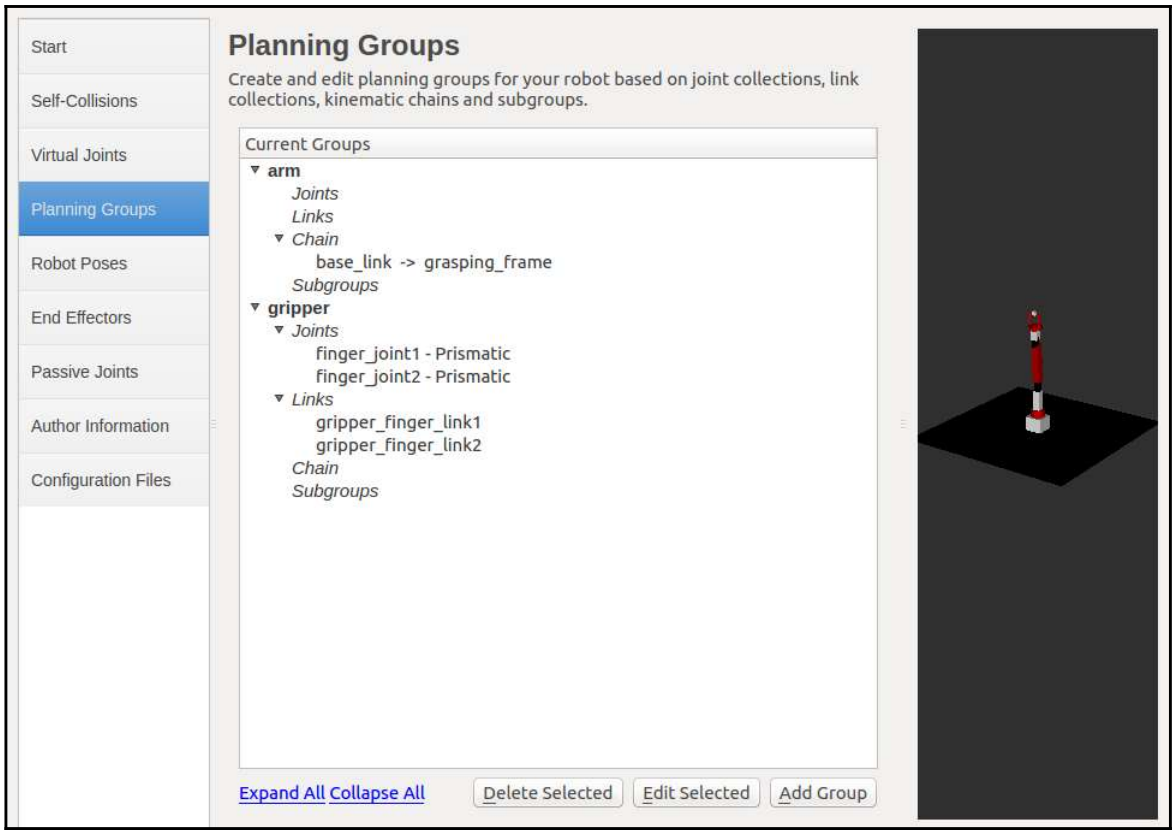

Figure 8: Adding the planning group of the arm and gripper

Step 5 – Adding the robot poses

In this step, we can add certain fixed poses in the robot configuration. For example, we can assign a home position or a pick/place position in this step. The advantage is that, while programming with MoveIt! APIs, we can directly call these poses, which are also called group states. These have many applications in the pick/place and grasping operation. The robot can switch to the fixed poses without any hassle.

Step 6 – Setting up the robot end effector

In this step, we name the robot end effector and assign the end-effector group, the parent link, and the parent group.

We can add any number of end effectors to this robot. In our case, it's a gripper designed for pick and place operations.

Click on the **Add End Effector** button and name the end effector as `robot_eef`, the planning group as gripper, which we have already created, the parent link as `grasping_frame`, and the parent group as `arm`:

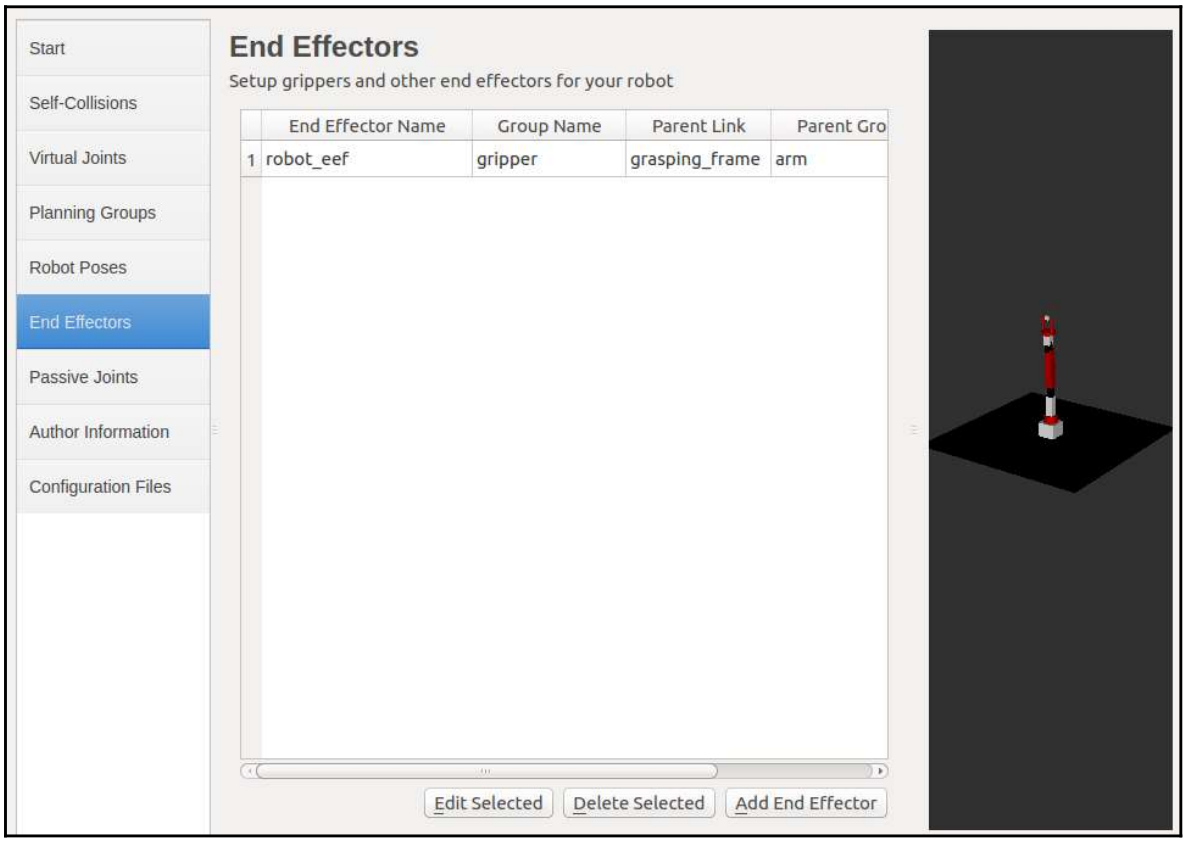

Figure 9: Adding end effectors

Step 7 – Adding passive joints

In this step, we can specify the passive joints in the robot. Passive joints mean that the joints do not have any actuators. Caster wheels are one of the examples of passive joints. The planner will ignore these kinds of joints during motion planning.

Step 8 – Author information

In this step, the author of the robotic model can add personal information, his name and email address, required by catkin to release the model to the ROS community.

Step 9 – Generating configuration files

We are almost done. We are in the final stage, that is, generating the configuration files. In this step, the tool will generate a configuration package which contains the file needed to interface MoveIt!.

Click on the **Browse** button to locate a folder to save the configuration file that is going to be generated by the Setup Assistant tool. Here we can see the files are generating inside a folder called `seven_dof_arm_config`. You can `add_config` or `_generated` along with the robot name for the configuration package.

Click on the **Generate Package** button, and it will generate the files to the given folder.

If the process is successful, we can click on **Exit Setup Assistant**, which will exit us from the tool.

The following screenshot shows the generation process:

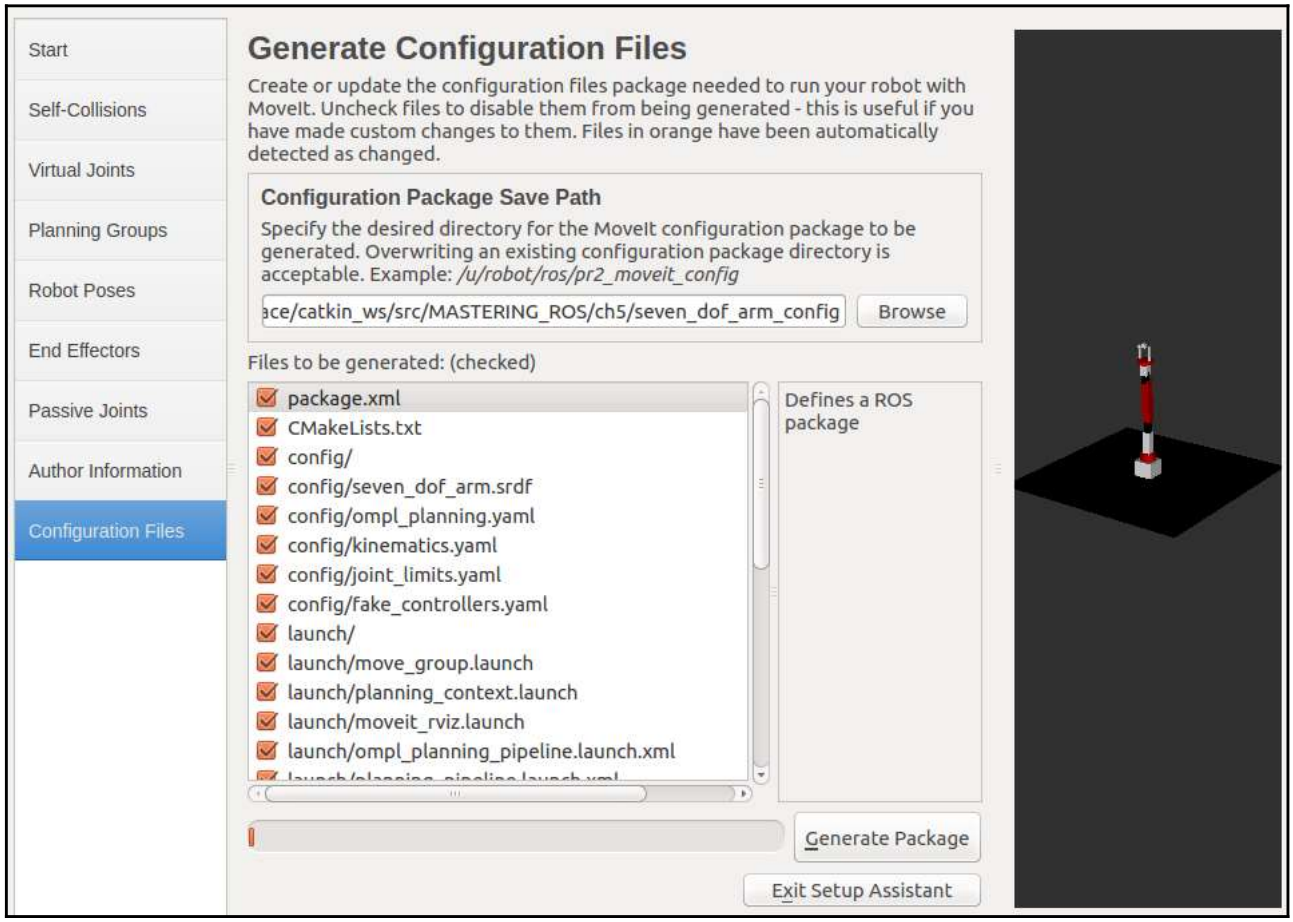

Figure 10: Generating the MoveIt! configuration package

After generating the MoveIt! configuration package, we can copy it into our `catkin` workspace. In the following section, we are going to work with this package. As usual, the model of the robot created can be downloaded from the following GitHub repository or can be obtained from the book's source code:

```
$ git clone https://github.com/jocacace/seven_dof_arm_config
```

Motion planning of robot in RViz using MoveIt! configuration package

MoveIt! provides a plugin for RViz, which allows it to create new planning scenes where **robot works**, **generate motion plans**, and **add new objects**, visualize the planning output and can directly interact with the visualized robot.

The MoveIt! configuration package consists of configuration files and launch files to start motion planning in RViz. There is a demo launch file in the package to explore all the functionalities of this package.

The following is the command to invoke the demo launch file:

```
$ roslaunch seven_dof_arm_config demo.launch
```

If everything works fine, we will get the following screen of RViz being loaded with the `MotionPlanning` plugin provided by MoveIt!:

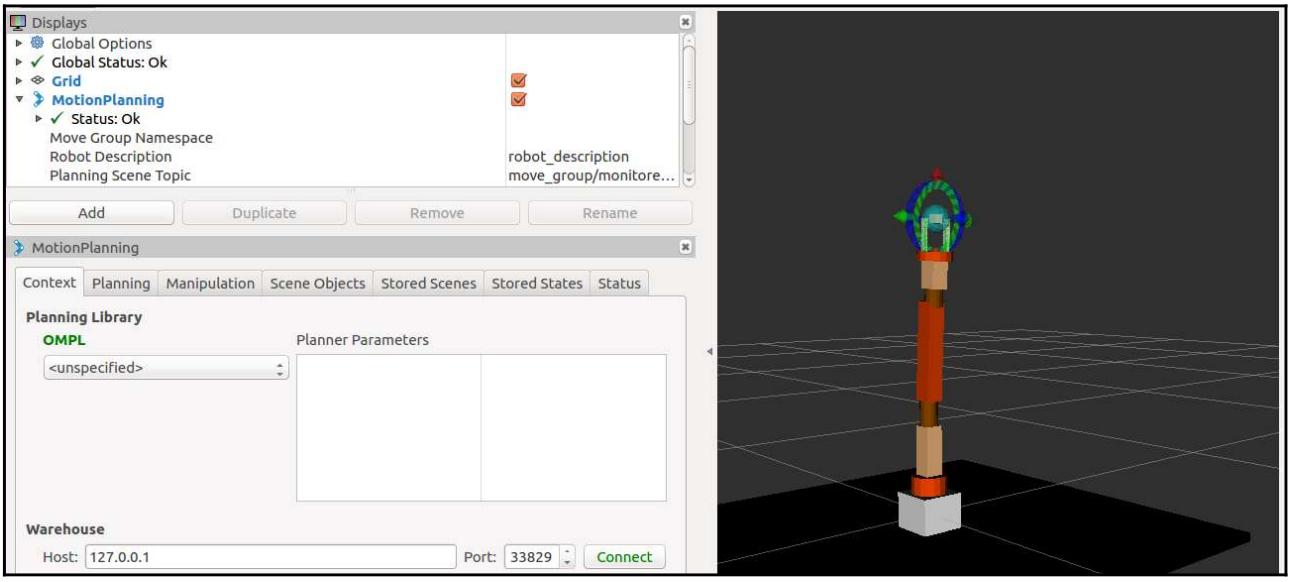

Figure 11: MoveIt! - RViz plugin

Using the RViz Motion Planning plugin

From the preceding figure , we can see that the RViz-Motion Planning plugin is loaded on the left side of the screen. There are several tabs on the **Motion Planning** window, such as **Context**, **Planning**, and so on. The default tab is the **Context** tab and we can see the default **Planning Library** as OMPL, which is shown in green. It indicates that MoveIt! successfully loaded the motion planning library. If it is not loaded, we can't perform motion planning.

Next is the **Planning** tab. This is one of the frequently used tabs used to assign the **Start State**, **Goal State**, **Plan** a path, and **Execute** the path. Shown next is the GUI of the **Planning** tab:

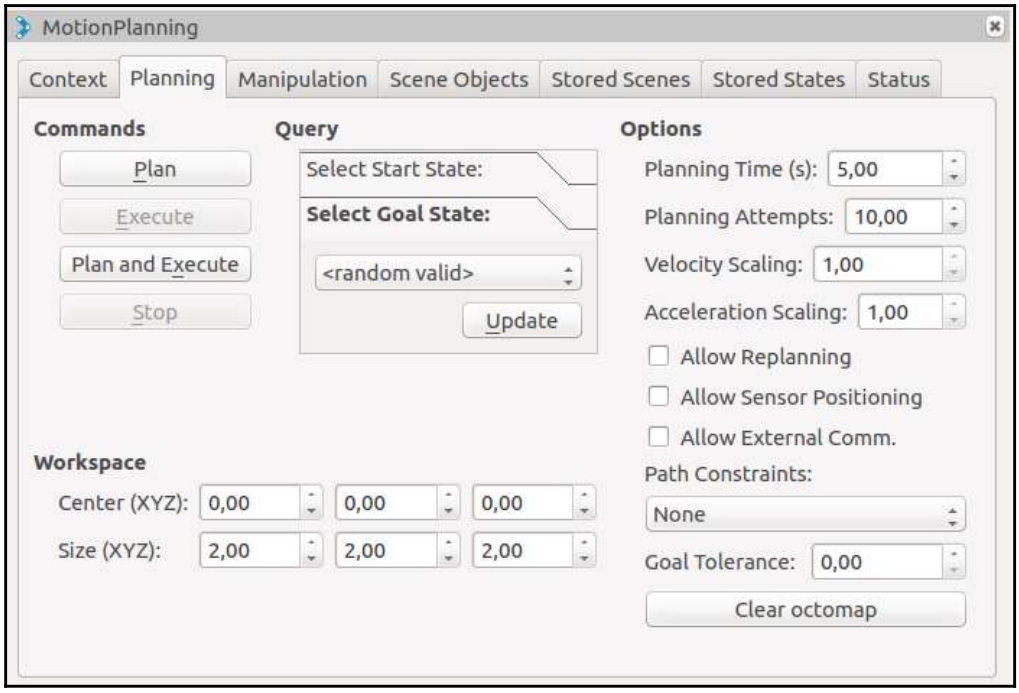

Figure 12: MoveIt! -RViz Planning tab

We can assign the start state and the goal state of the robot under the **Query** panel. Using the **Plan** button, we can plan the path from the start to the goal state, and if the planning is successful, we can execute it. By default, execution is done on fake controllers. We can change these controllers into trajectory controllers for executing the planned trajectory in Gazebo or the real robot.

We can set the starting and the goal position of the robot end effector by using the interactive marker attached on the arm gripper. We can translate and rotate the marker pose, and, if there is a planning solution, we can see an arm in orange color. In some situations, the arm will not move even the end-effector marker pose moves, and if the arm does not come to the marker position, we can assume that there is no IK solution in that pose. We may need more DOF to reach there or there might be some collision between the links.

The following screenshots show a valid goal pose and an invalid goal pose:

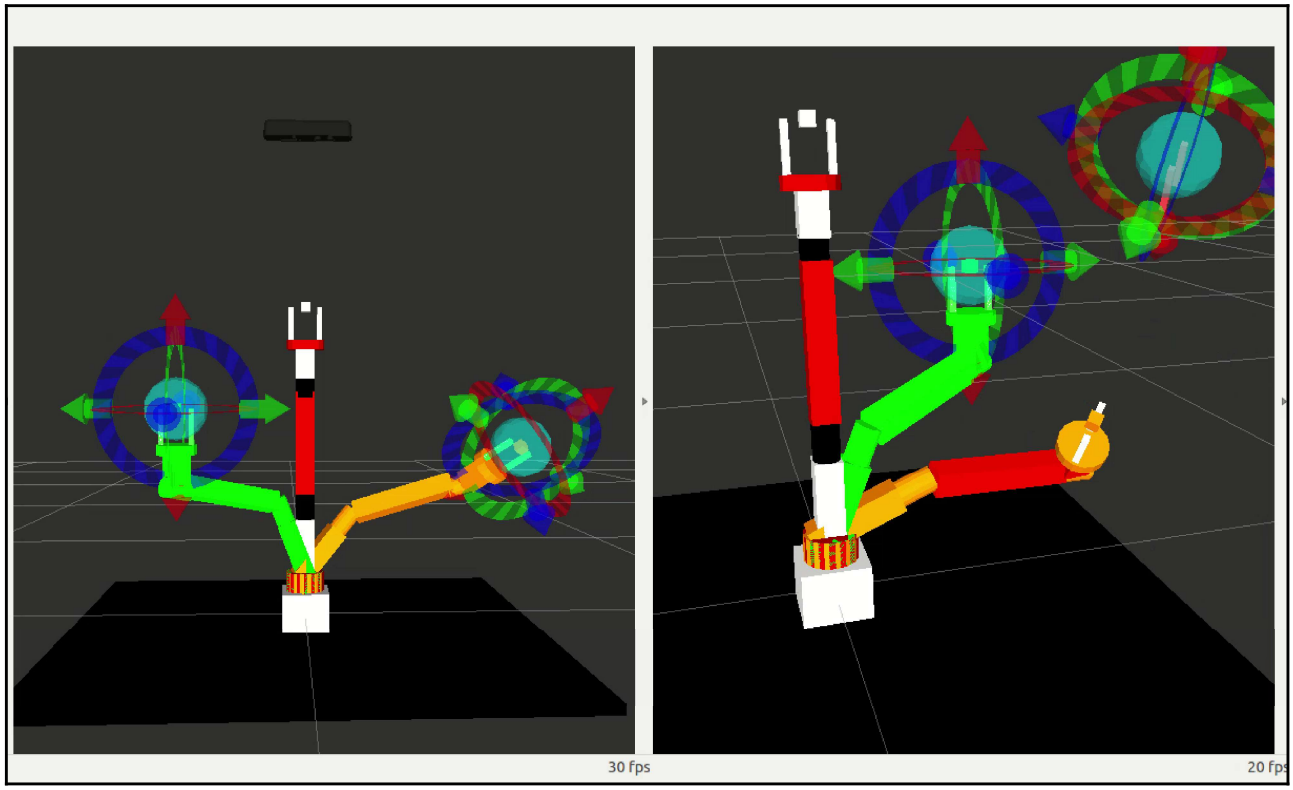

Figure 13: A valid pose and an invalid pose of the robot in RViz

The green colored arm represents the starting position of the arm, and the orange color represents the goal position. In the first figure, if we press the **Plan** button, MoveIt! plans a path from start to goal. In the second image, we can observe two things. First, one of the links of the orange arm is red, which means that the goal pose is in a self-collided state. Secondly, look at the end- effector marker; it is far from the actual end effector and it has also turned red.

We can also work with some quick motion planning using random valid options in the start state and the goal state. If we select the goal state as random valid and press the **Update** button, it will generate a random valid goal pose. Click on the **Plan** button and we can see the motion planning.

We can customize the RViz visualization using the various options in the `MotionPlanning` plugin. Shown next are some of the options of this plugin:

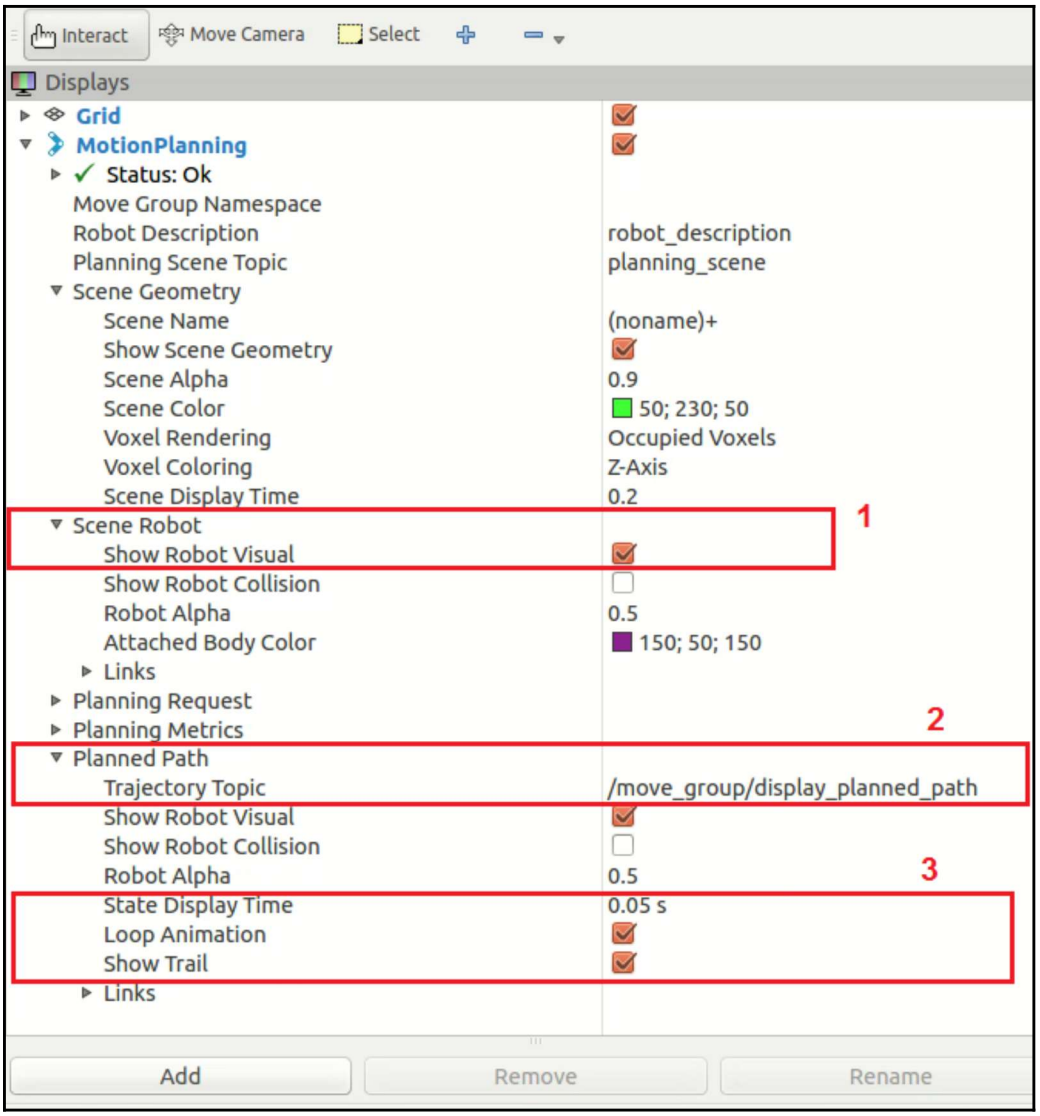

Figure 14: Settings of the MotionPlanning plugin on RViz

The first marked area is **Scene Robot**, which will show the robot model; if it is unchecked, we won't see any robot model. The second marked area is the **Trajectory Topic**, in which RViz gets the visualization trajectory. If we want to animate the motion planning and want to display the motion trails, we should enable this option.

One of the other sections in the plugin settings is shown in the following image:

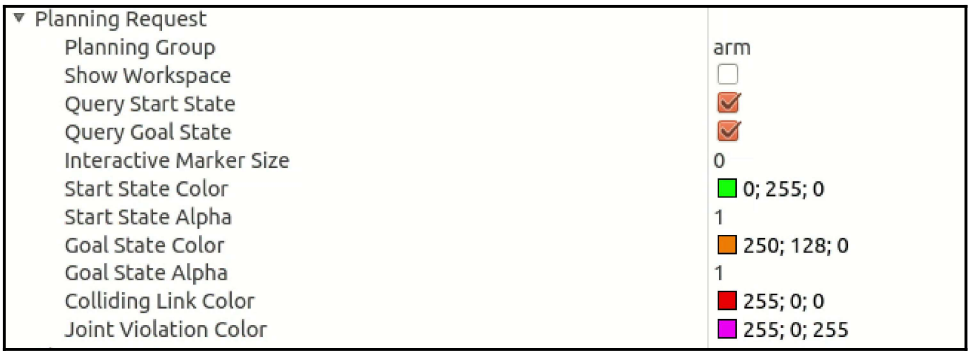

Figure 15: Planning Request setting in MotionPlanning plugin

In the preceding figure, we can see the **Query Start State** and the **Query Goal State** options. These options can visualize the start pose and the goal pose of the arm, which we saw in Figure 13. **Show Workspace** visualizes the cubic workspace (world geometry) around the robot. The visualization can help to debug our motion-planning algorithm and understand the robot motion behavior in detail.

In the next section, we will see how to interface the MoveIt! configuration package to Gazebo. This will execute the trajectory generated by MoveIt! in Gazebo.

Interfacing the MoveIt! configuration package to Gazebo

We have already worked with the Gazebo simulation of this arm and attached controllers to it. For interfacing the arm in MoveIt! to Gazebo, we need a trajectory controller which has the `FollowJointTrajectoryAction` interface, as we mentioned in the MoveIt! architecture.

The following is the procedure to interface MoveIt! to Gazebo.

Step 1 – Writing the controller configuration file for MoveIt!

The first step is to create a configuration file for talking with the trajectory controllers in Gazebo from MoveIt!. The controller configuration file called `controllers.yaml` has to be created inside the `config` folder of the `seven_dof_arm_config` package.

Given next is an example of the `controllers.yaml` definition:

```
controller_manager_ns: controller_manager
controller_list:
  - name: seven_dof_arm/seven_dof_arm_joint_controller
    action_ns: follow_joint_trajectory
    type: FollowJointTrajectory
    default: true
    joints:
      - shoulder_pan_joint
      - shoulder_pitch_joint
      - elbow_roll_joint
      - elbow_pitch_joint
      - wrist_roll_joint
      - wrist_pitch_joint
      - gripper_roll_joint

  - name: seven_dof_arm/gripper_controller
    action_ns: follow_joint_trajectory
    type: FollowJointTrajectory
    default: true
    joints:
      - finger_joint1
      - finger_joint2
```

The controller configuration file contains the definition of the two controller interfaces; one is for arm and the other is for gripper. The type of action used in the controllers is `FollowJointTrajectory`, and the action namespace is `follow_joint_trajectory`. We have to list out the joints under each group. The `default: true` indicates that it will use the default controller, which is the primary controller in MoveIt! for communicating with the set of joints.

Step 2 – Creating the controller launch files

Next, we have to create a new launch file called `seven_dof_arm_moveit_controller_manager.launch`, which can start the trajectory controllers. The name of the file starts with the robot name, which is added with `_moveit_controller_manager`.

The following is the `seven_dof_arm_config/launch/seven_dof_arm_moveit_controller_manager.launch` launch file definition:

```
<launch>
  <!-- Set the param that trajectory_execution_manager needs to find the
controller plugin -->
  <arg name="moveit_controller_manager"
default="moveit_simple_controller_manager/MoveItSimpleControllerManager" />
  <param name="moveit_controller_manager" value="$(arg
moveit_controller_manager)"/>

  <!-- load controller_list -->
  <arg name="use_controller_manager" default="true" />
  <param name="use_controller_manager" value="$(arg
use_controller_manager)" />

  <!-- Load joint controller configurations from YAML file to parameter
server -->
  <rosparam file="$(find seven_dof_arm_config)/config/controllers.yaml"/>
</launch>
```

This launch file starts the `MoveItSimpleControllerManager` and loads the joint trajectory controllers defined inside `controllers.yaml`.

Step 3 – Creating the controller configuration file for Gazebo

After creating the MoveIt! files, we have to create the Gazebo controller configuration file and the launch file.

Create a new file called `trajectory_control.yaml`, which contains the list of the Gazebo ROS controllers that need to be loaded along with Gazebo.

You will get this file from the `the seven_dof_arm_gazebo` package created in `Chapter 4`, *Simulating Robots Using ROS and Gazebo* in the `/config` folder.

The following is the definition of this file:

```
seven_dof_arm:
  seven_dof_arm_joint_controller:
    type: "position_controllers/JointTrajectoryController"
    joints:
      - shoulder_pan_joint
      - shoulder_pitch_joint
      - elbow_roll_joint
      - elbow_pitch_joint
      - wrist_roll_joint
      - wrist_pitch_joint
      - gripper_roll_joint

    gains:
      shoulder_pan_joint:   {p: 1000.0, i: 0.0, d: 0.1, i_clamp: 0.0}
      shoulder_pitch_joint: {p: 1000.0, i: 0.0, d: 0.1, i_clamp: 0.0}
      elbow_roll_joint:  {p: 1000.0, i: 0.0, d: 0.1, i_clamp: 0.0}
      elbow_pitch_joint:       {p: 1000.0, i: 0.0, d: 0.1, i_clamp: 0.0}
      wrist_roll_joint:    {p: 1000.0, i: 0.0, d: 0.1, i_clamp: 0.0}
      wrist_pitch_joint:      {p: 1000.0, i: 0.0, d: 0.1, i_clamp: 0.0}
      gripper_roll_joint:    {p: 1000.0, i: 0.0, d: 0.1, i_clamp: 0.0}


  gripper_controller:
    type: "position_controllers/JointTrajectoryController"
    joints:
      - finger_joint1
      - finger_joint2
    gains:
      finger_joint1:  {p: 50.0, d: 1.0, i: 0.01, i_clamp: 1.0}
      finger_joint2:  {p: 50.0, d: 1.0, i: 0.01, i_clamp: 1.0}
```

Here, we created a `position_controllers/JointTrajectoryController`, which has an action interface of `FollowJointTrajectory` for both the `arm` and the `gripper`. We also defined the PID gain associated with each joint, which can provide a smooth motion.

Step 4 – Creating the launch file for Gazebo trajectory controllers

After creating the configuration file, we can load the controllers along with Gazebo. We have to create a launch file which launches Gazebo, the trajectory controllers, and the MoveIt! interface in a single command.

The launch file `seven_dof_arm_bringup_moveit.launch` contains the definition to launch all these commands:

```
<launch>
  <!-- Launch Gazebo  -->
  <include file="$(find
seven_dof_arm_gazebo)/launch/seven_dof_arm_world.launch" />

  <!-- ros_control seven dof arm launch file -->
  <include file="$(find
seven_dof_arm_gazebo)/launch/seven_dof_arm_gazebo_states.launch" />

  <!-- ros_control trajectory control dof arm launch file -->
  <include file="$(find
seven_dof_arm_gazebo)/launch/seven_dof_arm_trajectory_controller.launch" />

  <!-- moveit launch file -->
  <include file="$(find
seven_dof_arm_config)/launch/moveit_planning_execution.launch" />

   <node name="joint_state_publisher" pkg="joint_state_publisher"
type="joint_state_publisher">
         <param name="/use_gui" value="false"/>
         <rosparam
param="/source_list">[/move_group/fake_controller_joint_states]</rosparam>
   </node>
</launch>
```

This launch file spawns the robot model in Gazebo, publishes the joint states, attaches the position controller, attaches the trajectory controller, and, finally, launches `moveit_planning_execution.launch` inside the MoveIt! package for starting the MoveIt! nodes along with RViz. We may need to load the `MotionPlanning` plugin in RViz if it is not loaded by default.

We can start motion planning inside RViz and execute in Gazebo using the following single command:

```
$ roslaunch seven_dof_arm_gazebo seven_dof_arm_bringup_moveit.launch
```

Note that, before properly launching the planning scene, we should use the following command to install some packages needed by MoveIt! to use ROS controllers:

```
$ sudo apt-get install ros-kinetic-joint-state-controller ros-kinetic-
position-controllers ros-kinetic-joint-trajectory-controller
```

After we have installed the preceding packages, we can launch the planning scene. This will launch RViz and Gazebo, and we can do motion planning inside RViz. After motion planning, click on the **Execute** button to send the trajectory to the Gazebo controllers:

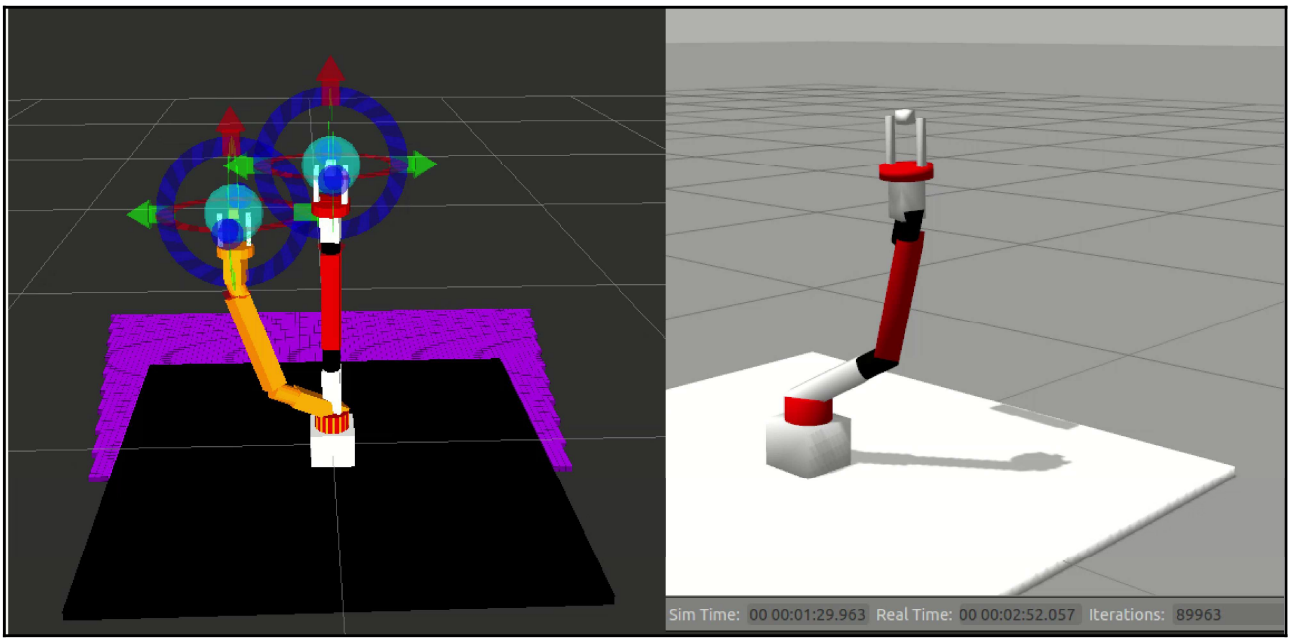

Figure 16: Gazebo trajectory controllers executing the trajectory from MoveIt!

Step 5 – Debugging the Gazebo- MoveIt! interface

In this section, we will discuss some of the common issues and debugging techniques in this interface.

If the trajectory is not executing on Gazebo, first list the topics:

```
$ rostopic list
```

If the Gazebo controllers are started properly, we will get the following joint trajectory topics in the list:

```
/seven_dof_arm/gripper_controller/command
/seven_dof_arm/gripper_controller/follow_joint_trajectory/cancel
/seven_dof_arm/gripper_controller/follow_joint_trajectory/feedback
/seven_dof_arm/gripper_controller/follow_joint_trajectory/goal
/seven_dof_arm/gripper_controller/follow_joint_trajectory/result
/seven_dof_arm/gripper_controller/follow_joint_trajectory/status
/seven_dof_arm/gripper_controller/state
/seven_dof_arm/joint_states
/seven_dof_arm/seven_dof_arm_joint_controller/command
/seven_dof_arm/seven_dof_arm_joint_controller/follow_joint_trajectory/cancel
/seven_dof_arm/seven_dof_arm_joint_controller/follow_joint_trajectory/feedback
/seven_dof_arm/seven_dof_arm_joint_controller/follow_joint_trajectory/goal
/seven_dof_arm/seven_dof_arm_joint_controller/follow_joint_trajectory/result
/seven_dof_arm/seven_dof_arm_joint_controller/follow_joint_trajectory/status
/seven_dof_arm/seven_dof_arm_joint_controller/state
/tf
/tf_static
/trajectory_execution_event
```

Figure 17: Topics from the Gazebo-ROS trajectory controllers

We can see `follow_joint_trajectory` for the gripper and the `arm` group. If the controllers are not ready, the trajectory will not execute in Gazebo.

Also, check the terminal message while starting the launch file:

```
 [1505806707.153599116, 0.343000000]: Added FollowJointTrajectory controller for seven_dof_ar
ller
 [1505806707.153740538, 0.343000000]: Returned 2 controllers in list
 [1505806707.205783246, 0.347000000]: Trajectory execution is managing controllers
 'move_group/ApplyPlanningSceneService'...
 'move_group/ClearOctomapService'...
 'move_group/MoveGroupCartesianPathService'...
 'move_group/MoveGroupExecuteTrajectoryAction'...
 'move_group/MoveGroupGetPlanningSceneService'...
 'move_group/MoveGroupKinematicsService'...
 'move_group/MoveGroupMoveAction'...
 'move_group/MoveGroupPickPlaceAction'...
 'move_group/MoveGroupPlanService'...
 'move_group/MoveGroupQueryPlannersService'...
 'move_group/MoveGroupStateValidationService'...                                        1
 [1505806835.903571251, 36.978000000]: arm[RRTkConfigDefault]: Starting planning with 1 state
astructure
 [1505806835.994742622, 36.997000000]: arm[RRTkConfigDefault]: Created 21 states
 [1505806836.036028021, 37.004000000]: arm[RRTkConfigDefault]: Created 38 states          2
 [1505806836.038435520, 37.005000000]: ParallelPlan::solve(): Solution found by one or more t
41 seconds
```

Figure 18: The Terminal message showing successful trajectory execution

In the Figure 18, the first section shows that the `MoveItSimpleControllerManager` was able to connect with the Gazebo controller and if it couldn't connect to controller, it shows that it can't connect to the controller. The second section shows a successful motion planning. If the motion planning is not successful, MoveIt! will not send the trajectory to Gazebo.

In the next section, we will discuss the ROS Navigation stack and look at the requirements needed to interface the Navigation stack to the Gazebo simulation.

Understanding the ROS Navigation stack

The main aim of the ROS Navigation package is to move a robot from the start position to the goal position, without making any collision with the environment. The ROS Navigation package comes with an implementation of several navigation-related algorithms which can easily help implement autonomous navigation in the mobile robots.

The user only needs to feed the goal position of the robot and the robot odometry data from sensors such as wheel encoders, IMU, and GPS, along with other sensor data streams, such as laser scanner data or 3D point cloud from sensors such as **Kinect**. The output of the Navigation package will be the velocity commands that will drive the robot to the given goal position.

The Navigation stack contains the implementation of the standard algorithms, such as SLAM, A *(star), Dijkstra, amcl, and so on, which can directly be used in our application.

ROS Navigation hardware requirements

The ROS Navigation stack is designed as generic. There are some hardware requirements that should be satisfied by the robot. The following are the requirements:

- The Navigation package will work better in differential drive and holonomic (total DOF of robot equals to controllable DOF of robots). Also, the mobile robot should be controlled by sending velocity commands in the form of: `x: velocity`, `y: velocity` (linear velocity), and `theta: velocity` (angular velocity).
- The robot should be equipped with a vision (*rgb-d*) or laser sensor to build the map of the environment.
- The Navigation stack will perform better for square and circular shaped mobile bases. It will work on an arbitrary shape, but performance is not guaranteed.

The following are the basic building blocks of the Navigational stack taken from the ROS website (`http://wiki.ros.org/navigation/Tutorials/RobotSetup`). We can see the purposes of each block and how to configure the Navigation stack for a custom robot:

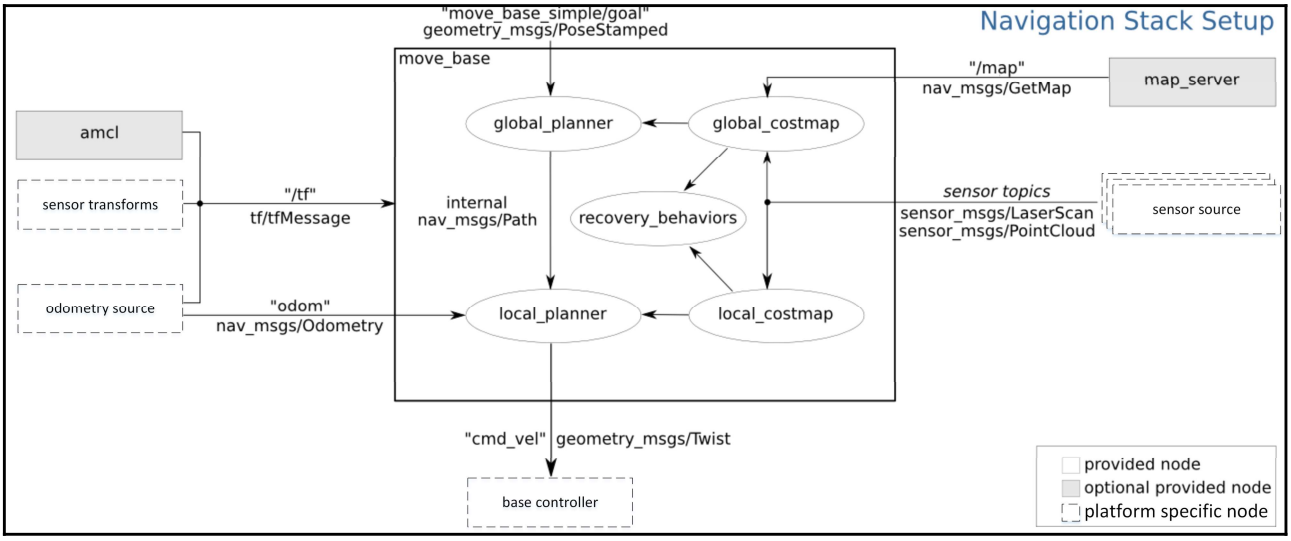

Figure 19: Navigation stack setup diagram

According to the Navigation setup diagram, for configuring the Navigation package for a custom robot, we must provide functional blocks that interface to the Navigation stack. The following are the explanations of all the blocks which are provided as input to the Navigational stack:

- **Odometry source**: Odometry data of a robot gives the robot position with respect to its starting position. The main odometry sources are wheel encoders, IMU, and 2D/3D cameras (visual odometry). The odom value should publish to the Navigation stack, which has a message type of `nav_msgs/Odometry`. The `odom` message can hold the position and the velocity of the robot. Odometry data is a mandatory input to the Navigational stack.
- **Sensor source**: We have to provide laser scan data or point cloud data to the Navigation stack for mapping the robot environment. This data, along with odometry, combines to build the global and local cost map of the robot. The main sensors used here are Laser Range finders or Kinect 3D sensors. The data should be of type `sensor_msgs/LaserScan` or `sensor_msgs/PointCloud`.

- **sensor transforms/tf**: The robot should publish the relationship between the robot coordinate frame using ROS tf.
- **base_controller**: The main function of the base controller is to convert the output of the Navigation stack, which is a twist (`geometry_msgs/Twist`) message, and convert it into corresponding motor velocities of the robot.

The optional nodes of the Navigation stack are `amcl` and map server, which allow localization of the robot and help to save/load the robot map.

Working with Navigation packages

Before working with the Navigation stack, we were discussing MoveIt! and the `move_group` node. In the Navigation stack, also, there is a node like the `move_group` node, called the `move_base` node. From Figure 19, it is clear that the `move_base` node takes input from sensors, joint states, TF, and odometry, which is very similar to the `move_group` node that we saw in MoveIt!.

Let's see more about the `move_base` node.

Understanding the move_base node

The `move_base` node is from a package called `move_base`. The main function of this package is to move a robot from its current position to a goal position with the help of other navigation nodes. The `move_base` node inside this package links the global-planner and the local-planner for the path planning, connecting to the rotate-recovery package if the robot is stuck in some obstacle and connecting global costmap and local costmap for getting the map.

The `move_base` node is basically an implementation of `SimpleActionServer`, which takes a goal pose with message type (`geometry_msgs/PoseStamped`). We can send a goal position to this node using a `SimpleActionClient` node.

The `move_base` node subscribes the goal from a topic called move`_base_simple/goal`, which is the input of the Navigation stack, as shown in the previous diagram.

When this node receives a goal pose, it links to components such as `global_planner`, `local_planner`, `recovery_behavior`, `global_costmap`, and `local_costmap`, generates the output, which is the command velocity (geometry_msgs/Twist), and sends it to the base controller for moving the robot for achieving the goal pose.

The following is the list of all the packages which are linked by the `move_base` node:

- **global-planner**: This package provides libraries and nodes for planning the optimum path from the current position of the robot to the goal position, with respect to the robot map. This package has the implementation of path-finding algorithms, such as A*, Dijkstra, and so on, for finding the shortest path from the current robot position to the goal position.
- **local-planner**: The main function of this package is to navigate the robot in a section of the global path planned using the global planner. The local planner will take the odometry and sensor reading, and send an appropriate velocity command to the robot controller for completing a segment of the global path plan. The base local planner package is the implementation of the trajectory rollout and dynamic window algorithms.
- **rotate-recovery**: This package helps the robot to recover from a local obstacle by performing a 360 degree rotation.
- **clear-costmap-recovery**: This package is also for recovering from a local obstacle by clearing the costmap by reverting the current costmap used by the Navigation stack to the static map.
- **costmap-2D**: The main use of this package is to map the robot environment. The robot can only plan a path with respect to a map. In ROS, we create 2D or 3D occupancy grid maps, which is a representation of the environment in a grid of cells. Each cell has a probability value that indicates whether the cell is occupied or not. The costmap-2D package can build the grid map of the environment by subscribing sensor values of the laser scan or point cloud and also the odometry values. There are global cost maps for global navigation and local cost maps for local navigation.

The following are the other packages which are interfaced to the `move_base` node:

- **map-server**: The map-server package allows us to save and load the map generated by the costmap-2D package.
- **AMCL**: AMCL (Adaptive Monte Carlo Localization) is a method to localize the robot in a map. This approach uses a particle filter to track the pose of the robot with respect to the map, with the help of probability theory. In the ROS system, AMCL accepts a `sensor_msgs/LaserScan` to create the map.
- **gmapping**: The gmapping package is an implementation of an algorithm called **Fast SLAM**, which takes the laser scan data and odometry to build a 2D occupancy grid map.

After discussing each functional block of the Navigation stack, let's see how it really works.

Working of Navigation stack

In the previous section, we saw the functionalities of each block in the ROS Navigation stack. Let's check how the entire system works. The robot should publish a proper odometry value, TF information, and sensor data from the laser, and have a base controller and map of the surroundings.

If all these requirements are satisfied, we can start working with the Navigation package.

Localizing on the map

The first step the robot is going to perform is localizing itself on the map. The AMCL package will help to localize the robot on the map.

Sending a goal and path planning

After getting the current position of the robot, we can send a goal position to the move_base node. The move_base node will send this goal position to a global planner, which will plan a path from the current robot position to the goal position.

This plan is with respect to the global costmap, which is feeding from the map server. The global planner will send this path to the local planner, which executes each segment of the global plan.

The local planner gets the odometry and the sensor value from the move_base node and finds a collision-free local plan for the robot. The local planner is associated with the local costmap, which can monitor the obstacle(s) around the robot.

Collision recovery behavior

The global and local costmap are tied with the laser scan data. If the robot is stuck somewhere, the Navigation package will trigger the recovery behavior nodes, such as the clear costmap recovery or rotate recovery nodes.

Sending the command velocity

The local planner generates the command velocity in the form of a twist message that contains linear and angular velocity (`geometry_msgs/Twist`), to the robot base controller. The robot base controller converts the twist message to the equivalent motor speed.

Installing the ROS Navigation stack

The ROS desktop full installation will not install the ROS Navigation stack. We must install the Navigation stack separately, using the following command:

```
$ sudo apt-get install ros-kinetic-navigation
```

After installing the Navigation package, let's start learning how to build a map of the robot environment. The robot we are using here is the differential wheeled robot that we discussed in the previous chapter. This robot satisfies all the three requirements of the Navigation stack.

Building a map using SLAM

The ROS Gmapping package is a wrapper of the open source implementation of SLAM, called OpenSLAM (`https://www.openslam.org/gmapping.html`). The package contains a node called `slam_gmapping`, which is the implementation of SLAM and helps to create a 2D occupancy grid map from the laser scan data and the mobile robot pose.

The basic hardware requirement for doing SLAM is a laser scanner which is horizontally mounted on the top of the robot, and the robot odometry data. In this robot, we have already satisfied these requirements. We can generate the 2D map of the environment, using the `gmapping` package through the following procedure.

Before operating with Gmapping, we need to install it using the following command:

```
$ sudo apt-get install ros-kinetic-gmapping
```

Creating a launch file for gmapping

The main task while creating a launch file for the `gmapping` process is to set the parameters for the `slam_gmapping` node and the `move_base` node. The `slam_gmapping` node is the core node inside the ROS Gmapping package. The `slam_gmapping` node subscribes the laser data (`sensor_msgs/LaserScan`) and the TF data, and publishes the occupancy grid map data as output (`nav_msgs/OccupancyGrid`). This node is highly configurable and we can fine tune the parameters to improve the mapping accuracy. The parameters are mentioned at `http://wiki.ros.org/gmapping`.

The next node we have to configure is the `move_base` node. The main parameters we need to configure are the global and local costmap parameters, the local planner, and the `move_base` parameters. The parameters list is very lengthy. We are representing these parameters in several YAML files. Each parameter is included in the `param` folder inside the `diff_wheeled_robot_gazebo` package.

The following is the `gmapping.launch` file used in this robot. The launch file is placed in the `diff_wheeled_robot_gazebo/launch` folder:

```
<launch>
  <arg name="scan_topic" default="scan" />

<!-- Defining parameters for slam_gmapping node -->

  <node pkg="gmapping" type="slam_gmapping" name="slam_gmapping"
output="screen">
    <param name="base_frame" value="base_footprint"/>
    <param name="odom_frame" value="odom"/>
    <param name="map_update_interval" value="5.0"/>
    <param name="maxUrange" value="6.0"/>
    <param name="maxRange" value="8.0"/>
    <param name="sigma" value="0.05"/>
    <param name="kernelSize" value="1"/>
    <param name="lstep" value="0.05"/>
    <param name="astep" value="0.05"/>
    <param name="iterations" value="5"/>
    <param name="lsigma" value="0.075"/>
    <param name="ogain" value="3.0"/>
    <param name="lskip" value="0"/>
    <param name="minimumScore" value="100"/>
    <param name="srr" value="0.01"/>
    <param name="srt" value="0.02"/>
    <param name="str" value="0.01"/>
    <param name="stt" value="0.02"/>
    <param name="linearUpdate" value="0.5"/>
    <param name="angularUpdate" value="0.436"/>
    <param name="temporalUpdate" value="-1.0"/>
    <param name="resampleThreshold" value="0.5"/>
    <param name="particles" value="80"/>
    <param name="xmin" value="-1.0"/>
    <param name="ymin" value="-1.0"/>
    <param name="xmax" value="1.0"/>
    <param name="ymax" value="1.0"/>

    <param name="delta" value="0.05"/>
    <param name="llsamplerange" value="0.01"/>
```

```
    <param name="llsamplestep" value="0.01"/>
    <param name="lasamplerange" value="0.005"/>
    <param name="lasamplestep" value="0.005"/>
    <remap from="scan" to="$(arg scan_topic)"/>
  </node>

<!-- Defining parameters for move_base node -->

  <node pkg="move_base" type="move_base" respawn="false" name="move_base"
output="screen">
    <rosparam file="$(find
diff_wheeled_robot_gazebo)/param/costmap_common_params.yaml" command="load"
ns="global_costmap" />
    <rosparam file="$(find
diff_wheeled_robot_gazebo)/param/costmap_common_params.yaml" command="load"
ns="local_costmap" />
    <rosparam file="$(find
diff_wheeled_robot_gazebo)/param/local_costmap_params.yaml" command="load"
/>
    <rosparam file="$(find
diff_wheeled_robot_gazebo)/param/global_costmap_params.yaml" command="load"
/>
    <rosparam file="$(find
diff_wheeled_robot_gazebo)/param/base_local_planner_params.yaml"
command="load" />
    <rosparam file="$(find
diff_wheeled_robot_gazebo)/param/dwa_local_planner_params.yaml"
command="load" />
    <rosparam file="$(find
diff_wheeled_robot_gazebo)/param/move_base_params.yaml" command="load" />

  </node>

</launch>
```

Running SLAM on the differential drive robot

We can build the ROS package called `diff_wheeled_robot_gazebo` and can run the `gmapping.launch` file for building the map. The following are the commands to start with the mapping procedure.

Start the robot simulation by using the Willow Garage world:

```
$ roslaunch diff_wheeled_robot_gazebo diff_wheeled_gazebo_full.launch
```

Start the `gmapping` launch file with the following command:

```
$ roslaunch diff_wheeled_robot_gazebo gmapping.launch
```

If the `gmapping` launch file is working fine, we will get the following kind of output on the Terminal:

```
[ INFO] [1505810240.049575967, 15.340000000]: Loading from pre-hydro parameter style
[ INFO] [1505810240.168699314, 15.381000000]: Using plugin "static_layer"
[ INFO] [1505810240.384469019, 15.449000000]: Requesting the map...
[ INFO] [1505810240.663457937, 15.552000000]: Resizing costmap to 288 X 608 at 0.050000 m/pix
[ INFO] [1505810240.871384865, 15.650000000]: Received a 288 X 608 map at 0.050000 m/pix
[ INFO] [1505810240.897210021, 15.656000000]: Using plugin "obstacle_layer"
[ INFO] [1505810240.913185546, 15.660000000]:     Subscribed to Topics: scan bump
[ INFO] [1505810241.183408917, 15.714000000]: Using plugin "inflation_layer"
[ INFO] [1505810241.592248141, 15.851000000]: Loading from pre-hydro parameter style
[ INFO] [1505810241.730240828, 15.900000000]: Using plugin "obstacle_layer"
[ INFO] [1505810241.978042290, 16.015000000]:     Subscribed to Topics: scan bump
[ INFO] [1505810242.124180243, 16.057000000]: Using plugin "inflation_layer"
[ INFO] [1505810242.504991688, 16.191000000]: Created local_planner dwa_local_planner/DWAPlannerROS
[ INFO] [1505810242.518319734, 16.198000000]: Sim period is set to 0.20
[ INFO] [1505810244.343111055, 16.967000000]: Recovery behavior will clear layer obstacles
[ INFO] [1505810244.546680028, 17.020000000]: Recovery behavior will clear layer obstacles
[ INFO] [1505810244.697982461, 17.046000000]: odom received!
```

Figure 20: Terminal messages during gmapping

Start the keyboard teleoperation for manually navigating the robot around the environment. The robot can map its environment only if it covers the entire area:

```
$ roslaunch diff_wheeled_robot_control keyboard_teleop.launch
```

The current Gazebo view of the robot and the robot environment is shown next. The environment is with obstacles around the robot:

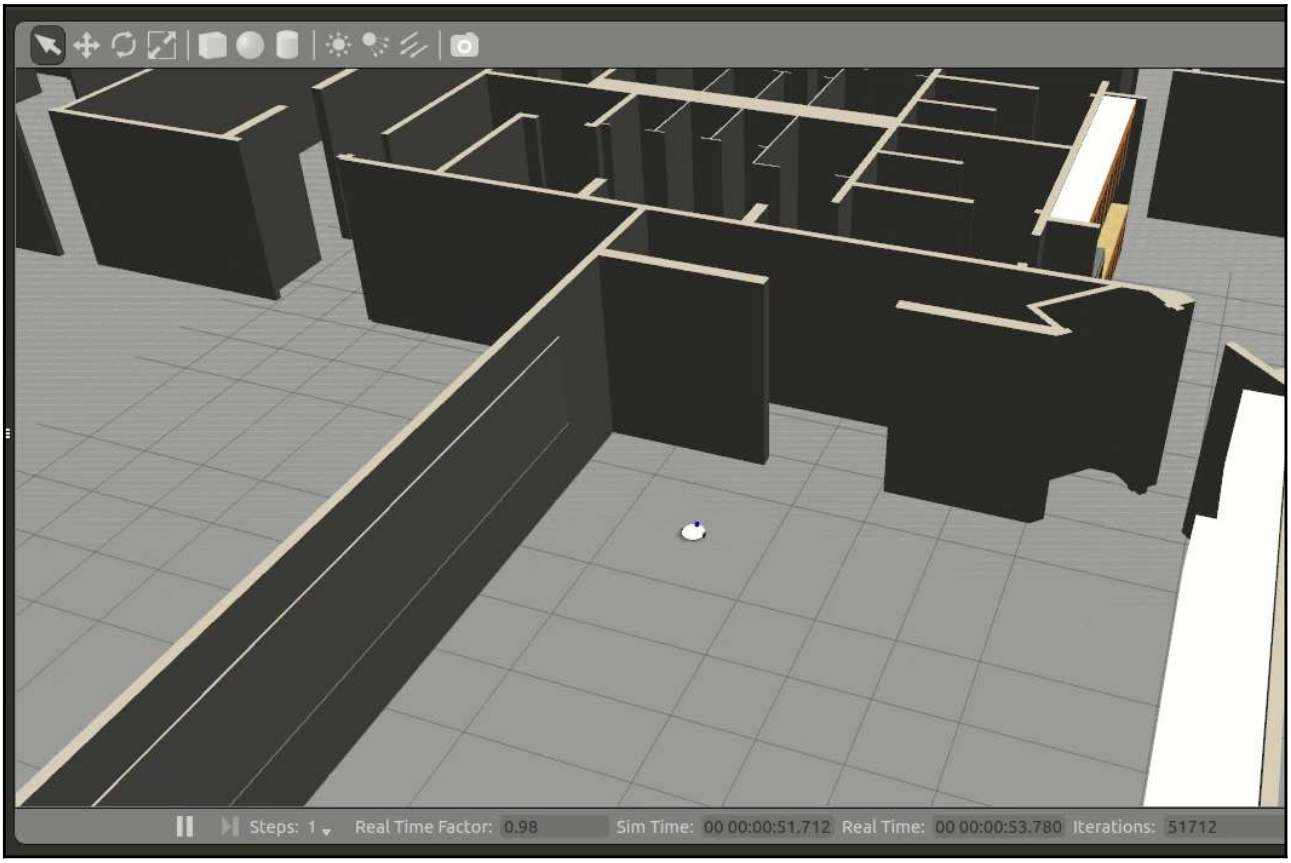

Figure 21: Simulation of the robot using the Willow Garage world

We can launch RViz and add a display type called **Map** and the topic name as `/map`.

We can start moving the robot inside the world by using key board teleoperation, and we can see a map building according to the environment. The following image shows the completed map of the environment shown in RViz:

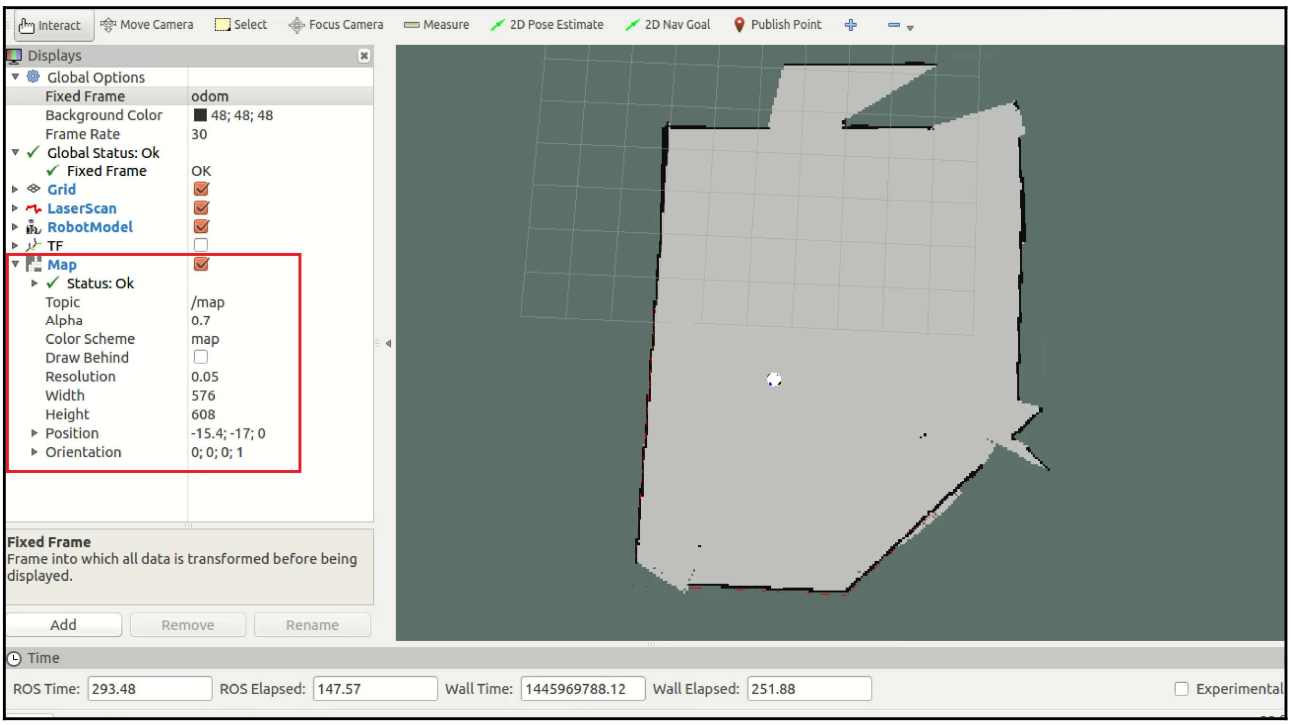

Figure 22: Completed map of the room in RViz

We can save the built map using the following command. This command will listen to the map topic and save into the image. The map server package does this operation:

```
$ rosrun map_server map_saver -f willo
```

Here `willo` is the name of the map file. The map file is stored as two files: one is the YAML file, which contains the map metadata and the image name, and second is the image, which has the encoded data of the occupancy grid map. The following is the screenshot of the preceding command, running without any errors:

```
jcacace@robot:~$ rosrun map_server map_saver -f willo
[ INFO] [1505810794.895750258]: Waiting for the map
[ INFO] [1505810795.117276658, 21.621000000]: Received a 288 X 608 map @ 0.050 m/pix
[ INFO] [1505810795.119888038, 21.621000000]: Writing map occupancy data to willo.pgm
[ INFO] [1505810795.138065942, 21.632000000]: Writing map occupancy data to willo.yaml
[ INFO] [1505810795.138632329, 21.632000000]: Done
```

Figure 23: Terminal screenshot while saving a map

The saved encoded image of the map is shown next. If the robot gives accurate robot odometry data, we will get this kind of precise map similar to the environment. The accurate map improves the navigation accuracy through efficient path planning:

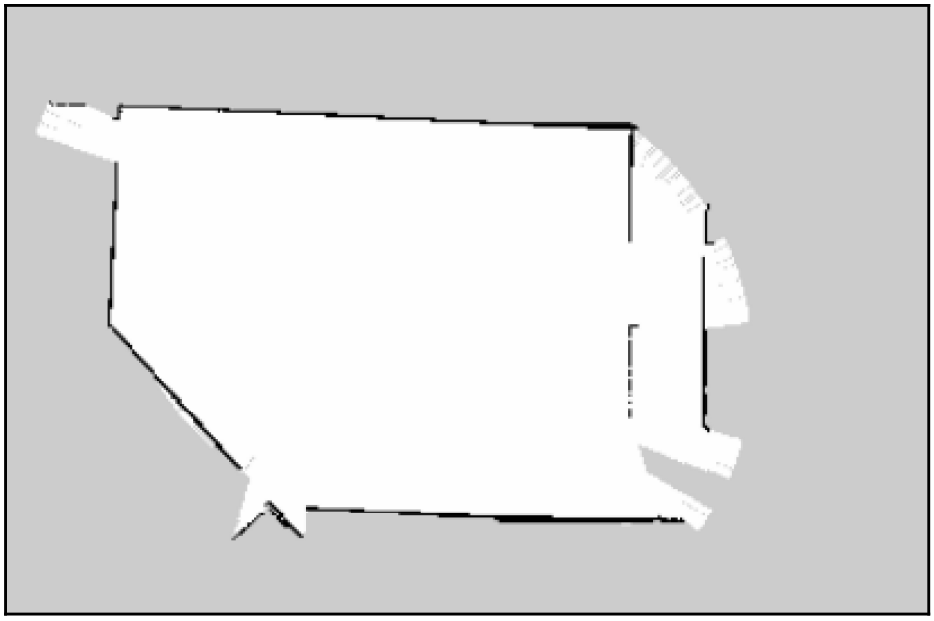

Figure 24: The saved map

The next procedure is to localize and navigate in this static map.

Implementing autonomous navigation using amcl and a static map

The ROS amcl package provides nodes for localizing the robot on a static map. The amcl node subscribes the laser scan data, laser scan based maps, and the TF information from the robot. The amcl node estimates the pose of the robot on the map and publishes its estimated position with respect to the map.

If we create a static map from the laser scan data, the robot can autonomously navigate from any pose of the map using amcl and the move_base nodes. The first step is to create a launch file for starting the amcl node. The amcl node is highly customizable; we can configure it with a lot of parameters. The list of parameters are available at the ROS package site (`http://wiki.ros.org/amcl`).

Creating an amcl launch file

A typical `amcl` launch file is given next. The amcl node is configured inside t `amcl.launch.xml` file, which is in the `diff_wheeled_robot_gazebo/launch/include` package. The `move_base` node is also configured separately in the `move_base.launch.xml` file. The map file we created in the `gmapping` process is loaded here, using the `map_server` node:

```
<launch>

  <!-- Map server -->
  <arg name="map_file" default="$(find
diff_wheeled_robot_gazebo)/maps/test1.yaml"/>
  <node name="map_server" pkg="map_server" type="map_server" args="$(arg
map_file)" />


  <include file="$(find
diff_wheeled_robot_gazebo)/launch/includes/amcl.launch.xml">

    <arg name="initial_pose_x" value="0"/>
    <arg name="initial_pose_y" value="0"/>
    <arg name="initial_pose_a" value="0"/>

  </include>

  <include file="$(find
diff_wheeled_robot_gazebo)/launch/includes/move_base.launch.xml"/>
```

```
</launch>
```

The following is the code snippet of `amcl.launch.xml`. This file is a bit lengthy, as we have to configure a lot of parameters for the `amcl` node:

```
<launch>
  <arg name="use_map_topic"  default="false"/>
  <arg name="scan_topic"     default="scan"/>
  <arg name="initial_pose_x" default="0.0"/>
  <arg name="initial_pose_y" default="0.0"/>
  <arg name="initial_pose_a" default="0.0"/>

  <node pkg="amcl" type="amcl" name="amcl">
    <param name="use_map_topic"             value="$(arg use_map_topic)"/>
    <!-- Publish scans from best pose at a max of 10 Hz -->
    <param name="odom_model_type"           value="diff"/>
    <param name="odom_alpha5"               value="0.1"/>
    <param name="gui_publish_rate"          value="10.0"/>
    <param name="laser_max_beams"             value="60"/>
    <param name="laser_max_range"           value="12.0"/>
```

After creating this launch file, we can start the amcl node, using the following procedure:

Start the simulation of the robot in Gazebo:

```
$ roslaunch diff_wheeled_robot_gazebo diff_wheeled_gazebo_full.launch
```

Start the `amcl` launch file, using the following command:

```
$ roslaunch diff_wheeled_robot_gazebo amcl.launch
```

If the `amcl` launch file is correctly loaded, the Terminal shows the following message:

```
[ INFO] [1505821904.100025792, 139.365000000]: Using plugin "static_layer"
[ INFO] [1505821904.277281445, 139.434000000]: Requesting the map...
[ INFO] [1505821904.489128458, 139.541000000]: Resizing costmap to 512 X 480 at 0.050000 m/pix
[ INFO] [1505821904.667453907, 139.643000000]: Received a 512 X 480 map at 0.050000 m/pix
[ INFO] [1505821904.675176680, 139.648000000]: Using plugin "obstacle_layer"
[ INFO] [1505821904.681719452, 139.648000000]:     Subscribed to Topics: scan bump
[ INFO] [1505821904.813327088, 139.699000000]: Using plugin "inflation_layer"
[ INFO] [1505821905.081866940, 139.802000000]: Using plugin "obstacle_layer"
[ INFO] [1505821905.194340020, 139.871000000]:     Subscribed to Topics: scan bump
[ INFO] [1505821905.323469494, 139.903000000]: Using plugin "inflation_layer"
[ INFO] [1505821905.674954354, 140.036000000]: Created local_planner dwa_local_planner/DWAPlannerROS
[ INFO] [1505821905.689447045, 140.040000000]: Sim period is set to 0.20
[ INFO] [1505821907.560275254, 141.046000000]: Recovery behavior will clear layer obstacles
[ INFO] [1505821907.785016235, 141.138000000]: Recovery behavior will clear layer obstacles
[ INFO] [1505821907.949123108, 141.197000000]: odom received!
```

Figure 25: Terminal screenshot while executing amcl

If amcl is working fine, we can start commanding the robot to go into a position on the map using RViz, as shown in the following figure. In the figure, the arrow indicates the goal position. We have to enable LaserScan, Map, and Path visualizing plugins in RViz for viewing the laser scan, the global/local costmap, and the global/local paths. Using the **2D NavGoal** button in RViz, we can command the robot to go to a desired position.

The robot will plan a path to that point and give velocity commands to the robot controller to reach that point:

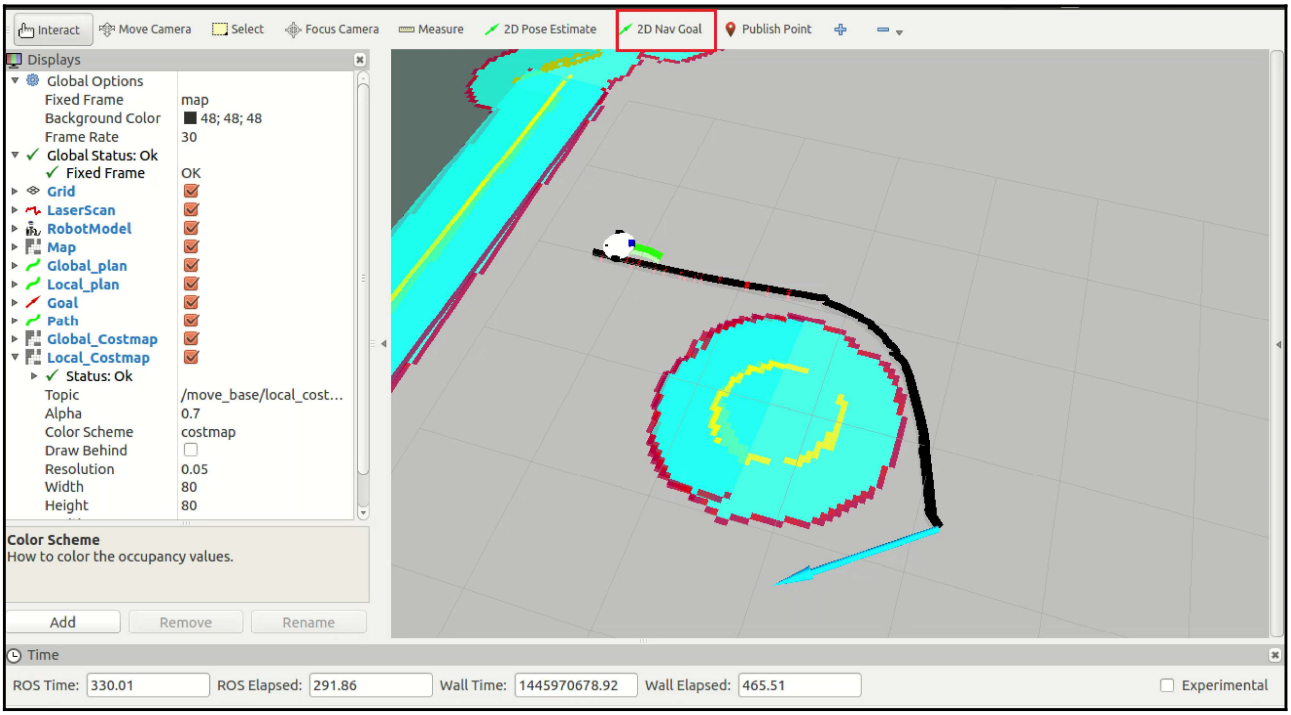

Figure 26: Autonomous navigation using amcl and the map

In the preceding image, we can see that we have placed a random obstacle in the robot's path, and that the robot has planned a path to avoid the obstacle.

We can view the amcl particle cloud around the robot by adding a **Pose Array** on RViz and the topic is /particle_cloud. The following image shows the amcl particle around the robot:

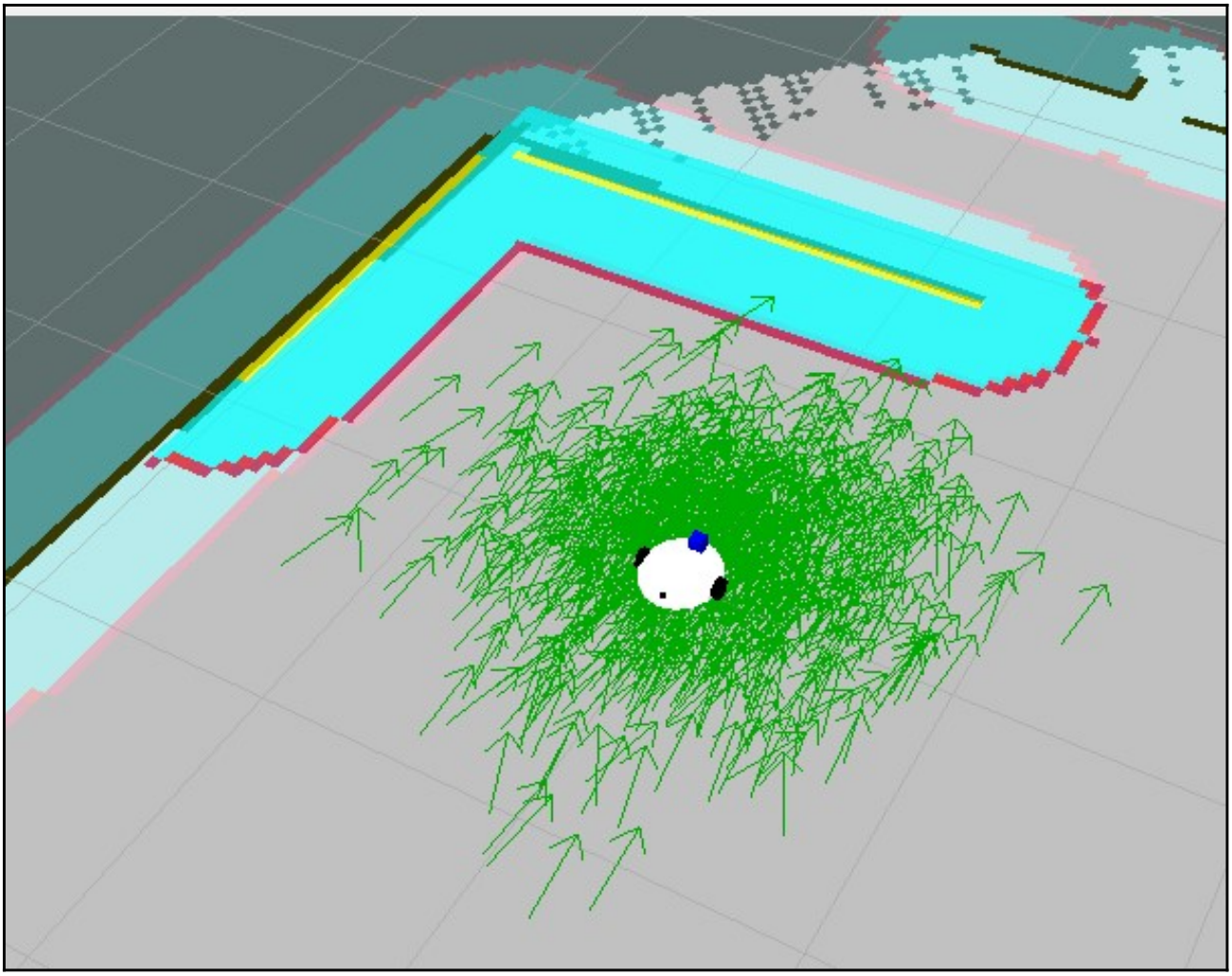

Figure 27: The amcl particle cloud and odometry

Questions

- What is the main purpose of MoveIt! packages?
- What is the importance of the `move_group` node in MoveIt!?
- What is the purpose of the move_base node in the Navigation stack?
- What are the functions of the SLAM and amcl packages?

Summary

This chapter offered a brief overview of MoveIt! and the Navigation stack of ROS, and demonstrated its capabilities using Gazebo simulation of a robotic arm mobile base. The chapter started with a MoveIt! overview and discussed detailed concepts about MoveIt!. After discussing MoveIt!, we interfaced MoveIt! and Gazebo. After interfacing, we executed the trajectory from MoveIt! on Gazebo.

The next section was about the ROS Navigation stack. We discussed its concepts and workings as well. After discussing the concepts, we tried to interface our robot in Gazebo to the Navigation stack and build a map using SLAM. After doing SLAM, we performed autonomous navigation using amcl and the static map.

In the next chapter, we will discuss pluginlib, nodelets, and controllers.