Chapter 5 Building an Autonomous Mobile Robot

In the previous nine chapters, we have learned how to provide position to the good another robot using odometry, dead reckoning and localizing algorithms. With precise sensor measurements and odometry, we can autonomously visiting some fixed waypoints. Some of the sensors used in the autonomous robot are like the encoders, IMU, laser scanner and etc.). Combining these sensors with the algorithms we have learned in the previous chapters, we will take a DIY autonomous mobile robot platform that using ROS. This project is actually the combination of the knowledge we have learned in the previous chapters. Keeping Moving Robot (Step by Step Building a Ros Robot) is the goal of this project. We will mainly go through the following topics: Hardware Assembly, Control the hardware of robot, and Finally Navigation. Hardware Assembly of robot will be introduced all these things since Hardware is the foundation of robot. Control the hardware of robot is the bridge of software and hardware. Finally Navigation is the goal of this project. The following are the main topics we will discuss in this chapter:

- Robot specification and design overview
- Designing and selecting motors and wheels for the robot
- Building a DIY and 3D models of the robot
- Driving the robot model in Gazebo
- Designing and building circuit board hardware
- Interfacing robot hardware with ROS
- Setting up the ROS navigation stack and propagating odometer
- Final run

Robot specification and design overview

Here are the main specifications of the robot that we are going to design in this chapter.

- A maximum payload of 2 Kg
- Body weight of 3 Kg
- A maximum speed of 0.25 m/s
- Ground clearance of 3 cm
- Two hours of continuous operation
- Differential drive configuration
- Circular base footprint
- Autonomous navigation and obstacle avoidance
- Low cost platform

We are going to design a robot that satisfies all these specifications.

Designing and selecting the motors and wheels for the robot

The robot we are going to design should have a payload of 2 kilograms, and from the prevailing specifications, we can first estimate the required motor torque. From the payload and robot body weight, we can estimate the required motor torque.

Computing motor torque requirements

Let's make some basic assumptions first.

The number of wheels is two, meaning two motors driven. The number of wheels is two, meaning the robot weight is on two wheels. We can assume the robot weight is equally distributed of wheel radius is 6.5 cm. The wheel radius is 6.5 cm.

Total weight: 2 Kilo + 3 Kilo = 5 Kilo

Weight of the wheels: 0.1 Kilo + 0.1 Kilo = 0.2 Kilo

Total weight: 5.2 Kilo

Total weight, $W = 5.2 \times 9.8 = 51$ N

Weight on each wheel: $W/2 = 25.5$ N

Assume the coefficient of rolling friction of the robot is 0.02, meaning to start wheel, we need to overcome 0.02 of the weight. So the minimum torque is:

Wheel radius is 6.5 mm thus the weight of the robot is equally distributed on two wheels, meaning each wheel has to support the average weight here.

Wheel radius = 6.5 cm = 0.065 m

Torque, $\tau = (0.02 \times 25.5) \times 0.065$

$\tau = 0.33$ N-m

Torque = 3.4 kg-cm

Assume the coefficient of static friction between rubber and ground is 0.04, meaning to start the robot, we need to overcome 0.04 of the weight. So the maximum torque is:

$\tau = (0.04 \times 25.5) \times 0.065$

$\tau = 0.66$ N-m

Torque = 6.7 kg-cm

Torque = 6.7 kg-cm

$\omega = 0.25 / 0.065 = 3.85$ rad/s

$\omega = 3.85 \times 60 / 2\pi = 36.8$ rpm

$\omega = 36.8 \times 0.065 \times \pi = 7.5$ m/min

$\omega = 7.5$ m/min

$\omega = 0.125$ m/s

$\omega = 12.5$ cm/s

$\omega = 0.125 \times 100 = 12.5$ cm/s

$\omega = 12.5$ cm/s

We can choose a standard 80 rpm DC motor with 6 kg-cm torque.

Design summary

- Two 80 rpm DC motors
- Motor torque: 10 kg-cm
- Wheel diameter: 6 cm

Building DY and 3D models of the robot body

Chassis design is the main step in designing the robot. We need to ensure the DY drawing of the chassis is accurate and useful in the long run. The specification need to satisfy is: Two motors should be placed in the chassis, the chassis should be able to carry the payload, and the sensors should be placed at appropriate locations. The chassis should be able to carry the payload and the sensors should be placed at the right locations. Keeping the payload in the center of the chassis is a good idea

Figure 7. The 3D render

The 3D render tool uses the saved `.chip_layout_3d_states.txt` file.

Simulating the robot model in Gazebo

After modeling the robot, the next step is to simulate it in a realistic simulation. The simulation is crucial for testing the robot's dynamics before deploying it on the simulation, minimizing real-world damage and ensuring the robot's structural integrity. When we do the actual robot, there can be unexpected issues due to unanticipated perturbations. We can simulate the robot using Gazebo, a robust simulation tool. Gazebo is a powerful open-source simulation tool for modeling and testing robots in realistic environments. It provides a high-fidelity simulation environment with support for various sensors, dynamics, and physics. This allows us to test our robot's behavior before deploying it on real hardware. Gazebo integrates with ROS 2 and supports URDF and SDF formats, making it compatible with our robot model.

Mathematical model of a differential drive robot

In ours, long-term, robot transmitters to the study of differential drive robots and the forces that reflect the motion and robot dynamics to the ground. The robot's motion is governed by the wheel velocities, which determine its linear and angular velocity. The robot's state is defined by its position and orientation in a 2D plane, represented by the state vector $[x, y, \theta]^T$. The kinematics of the robot are governed by the following equations:

Figure 8. Forward kinematics model of a differential drive robot

Here are the unknown variables:

- x is the robot's position in the x-axis
- y is the robot's position in the y-axis
- θ is the robot's orientation (yaw angle)
- v is the robot's linear velocity
- ω is the robot's angular velocity

$$\dot{x} = v\cos\theta$$

$$\dot{y} = v\sin\theta$$

$$\dot{\theta} = \omega$$

Figure 9. Forward kinematics equations

Here are the unknown variables:

- $\dot{x}$ is the robot's velocity in the x-axis
- $\dot{y}$ is the robot's velocity in the y-axis
- $\dot{\theta}$ is the robot's angular velocity

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} v\cos\theta \\ v\sin\theta \\ \omega \end{bmatrix}$$

Figure 10. Forward kinematics equations in matrix form

Here are the unknown variables:

- $\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix}$ is the robot's velocity in the x-axis, y-axis, and angular velocity

Figure 11. Forward kinematics model of a differential drive robot

For more information, see the Differential Wheel Mobile Robot Equation for Differential Drive Robot and Differential Drive Robot Kinematics by Thomas Rodriguez.

So we've seen the kinematics model of the robot. The next step is to simulate this robot.

Importing Chatbot

The robot in the book is controlled by a chatbot, and we are adding this to the customers to a robot. It is called Chatbot. The chatbot runs on a local server, and the robot is also simulating. Chatbot. We are using ngrok to expose the ngrok URL to simulate the chatbot. We are using ngrok's free tier, which gives us a fixed URL. We modified our chatbot to use the ngrok URL. We are using `ngrok.yaml` to define the ngrok URL. Chatbot. Since it is a project that requires a lot of files, we have created a `chatbot_deploy.zip` file containing all the files. We modified our chatbot to use the ngrok URL. We are using `ngrok.yaml` to define the ngrok URL. Chatbot. Since it is a project that requires a lot of files, we have created a `chatbot_deploy.zip` file containing all the files.

Importing Gazebo simulation into Chatbot

We are making this, and we are navigating parts, we need to import the Gazebo simulation into Chatbot. Gazebo simulation is a robot simulation tool that is block with software models of robots, sensors, and environments for differential drive robots. Gazebo plugins. You can find this code in `gazebo_diff_drive.zip`.

```
chatbot_ros_msgs
chatbot_ros_bringup
chatbot_ros_chatter
chatbot_ros_llama_msgs
chatbot_ros_llama_bringup
chatbot_ros_llama_chatter
chatbot_ros_llama_tools
chatbot_ros_llama_vision
chatbot_ros_llama_navigation
chatbot_ros_llama_manipulation
chatbot_ros_llama_gazebo
chatbot_ros_llama_bringup_sim
chatbot_ros_llama_chatter_sim
chatbot_ros_llama_navigation_sim
chatbot_ros_llama_manipulation_sim
chatbot_ros_llama_vision_sim
chatbot_ros_llama_tools_sim
chatbot_ros_llama_msgs_sim
chatbot_ros_llama_gazebo_sim
chatbot_ros_gazebo_sim
chatbot_ros_gazebo_msgs
```

In this plugin, we are providing the designed robot of the chatbot, including its wheels, sensor frames, and sensor separation. The differential drive plugin is `libgazebo_ros_diff_drive.so`, which is used in the Gazebo simulation for Chatbot.

After creating this variables, we need to create a depth camera plugin and a lidar plugin. Gazebo provides these plugins, `libgazebo_ros_camera.so` and `libgazebo_ros_ray_sensor.so`. We create these files in `gazebo_plugins.zip`.

```
chatbot_ros_gazebo_plugins
chatbot_ros_gazebo_plugins/CMakeLists.txt
chatbot_ros_gazebo_plugins/package.xml
chatbot_ros_gazebo_plugins/plugins/diff_drive.gazebo.xacro
chatbot_ros_gazebo_plugins/plugins/camera.gazebo.xacro
chatbot_ros_gazebo_plugins/plugins/lidar.gazebo.xacro
```

In this depth camera and lidar, we set a smooth resolution and range for the camera and lidar.

Note

Gazebo Plugins

You can find the Gazebo code using the following command: `$ git clone https://github.com/gazebo-release/gazebo_ros_pkgs`

Running the simulation

Now, we are ready to run our robot, we need to satisfy some dependencies. We need to install `ros2-control` and `ros2-controllers` packages. Now, we can run the simulation.

```
$ sudo apt-get install ros-humble-ros2-control ros-humble-ros2-controllers
$ ros2 launch chatbot_ros_bringup sim.launch.py
```

You can also launch RVIZ packages such as `chatbot_ros_bringup/rviz/nav.rviz` and `chatbot_ros_bringup/rviz/slam.rviz` as per your requirements. Now, we can see the robot model in RVIZ. We can teleoperate and launch the `chatbot_ros_chatter` node.

Figure 12. Simulation of Chatbot in Gazebo

You can move the robot around using a teleop twist. You can use the following command.

```
$ ros2 run teleop_twist_keyboard teleop_twist_keyboard
```

Figure 13. Keyboard teleop

Now, we are ready to start our navigation.

Mapping and localization

We are using slam toolbox for mapping and localization of the simulated robot. Mapping is a process of creating a map of the environment using sensor data, while localization is the process of determining the robot's position within the map. We are using the `slam_toolbox` package for mapping and localization. We can install the `slam_toolbox` package using the following command.

```
$ sudo apt-get install ros-humble-slam-toolbox
```

Now, we are ready to start the slam toolbox. We can use the following command.

```
$ ros2 launch chatbot_ros_bringup mapping.launch.py use_sim_time:=true
```

Figure 14. Slam toolbox in action

To save mapping the environment, we need to run the following command. This file will start the grouping node and save the map in `maps/map_name`.

```
$ ros2 run chatbot_ros_bringup save_map.sh map_name
```

After launching grouping, we need to start the navigation. We are launching the navigation using the following command. We are using the following command will start the navigation stack with the saved map.

```
$ ros2 launch chatbot_ros_bringup navigation.launch.py map:=maps/map_name.yaml use_sim_time:=true
```

Now, we can start the teleop node and move around the world. We will need to keep the teleop running.

Figure 15. The map visualized in RVIZ

After building the map, we are ready to start using the following command.

```
$ ros2 run chatbot_ros_chatter map_client.py
```

This will save the map in the `maps/map_name.yaml`.

Congratulations you have successfully created a map of the environment. The next step is to use this map and implement navigation using the world. We would like to use the Nav2 stack for navigation. Continuing this with the ROS 2 navigation stack, we need to create a costmap representing the world around the world.

Navigation and localization

Once of the Tutorials we have used the Nav2 navigation stack for navigation. We are using the following command.

```
$ ros2 launch chatbot_ros_bringup navigation.launch.py map:=maps/map_name.yaml use_sim_time:=true
```


Figure 16. Nav2 navigation

This will save the map in the `maps/map_name.yaml`.

Congratulations you have successfully created a map of the environment. The next step is to use this map and implement navigation using the world. We would like to use the Nav2 stack for navigation. Continuing this with the ROS 2 navigation stack, we need to create a costmap representing the world around the world.

Navigation and localization

Once of the Tutorials we have used the Nav2 navigation stack for navigation. We are using the following command.

```
$ ros2 launch chatbot_ros_bringup navigation.launch.py map:=maps/map_name.yaml use_sim_time:=true
```

Start transition using the button or command:

```
ros2 lifecycle set /talker 1
```

```
ros2 lifecycle set /talker 4
```

This will load the talker node and start it. Note, the example talker node, which is a publisher, starting the following command:

```
ros2 topic echo /chatter
```

```
$ LIFECYCLE = 1 ros2 lifecycle set /talker shutdown
```

Now, we can start managing the lifecycle of the talker node using the client as the RQt lifecycle tool and click on the map to start the talker node and see the position. You can also try autonomously move from the Rviz2 GUI using the 2D Goal Pose button.

Figure 15: Flooding

Designing and tuning actual robot hardware

Let's build the actual hardware of the robot. We need actual motors, encoders, power supply and additional micro sensors to perform SLAM and AMCL. Here is the list:

No.	Component Name	Cost
1	30 : gear motor with encoder	$15
2	Motor driver	$15
3	Tires 2 125 mm 1 Pair	$10
4	Ball caster	$5
5	5V/Sensor power	$10
6	MPU 6050 (Gyro/Acc)	$10
7	Jetson Nano Developer Kit	$100
8	Battery	$15
9	Chassis	$15
10	Lidar Sensor	$100

Let's build the actual hardware part of the robot.

Motor and motor driver

The motors are controlled using motor driver. We need to write a low-level program of the robot to adjust the speed of the robot. We need to write a low-level program to change the speed of the motor based on the position of the motors. We are using Jetson Nano, so we need to write a low-level program that runs on the Kit in the table.

Motor encoders

Motor encoders are sensors attached to motors to measure the position of the motor. Usually, we use Quadrature encoders to measure the position of the motor. We can use the encoders to calculate the position of the robot.

IMU sensor

To our ROS unit for odometry, we need to measure the robot's position and orientation in 3D. We use Inertial IMU. It can essentially run the robot's position and orientation.

Interfacing sensors and motors with the Launchpad

In this section, we will see how to interface motors and sensors with the Launchpad. The Launchpad can be used to interface motor drivers and sensors. Using Launchpad, we can have a dedicated processor for the motor control and sensor data. Using Launchpad, we can have real-time control of motors and sensors. Using Launchpad, we can have a dedicated controller with sensors.

Ultrasonic sensor

The ultrasonic sensor is used to detect nearby obstacles. If any obstacle is in the robot's path, it is useful to stop the robot. We can use the ultrasonic sensor to detect the obstacle in front of the robot. We can use the ultrasonic sensor to detect the obstacle and stop the robot to have zero delay.

Cost-effective robot sensor

The ultrasonic sensor, we will use the HC-SR04 sensor. It is a cost-effective robot sensor. It can measure the distance between the robot and the obstacle. It can measure the distance between the robot and the obstacle. Obtaining data from the sensor is very simple. We can use the HC-SR04 sensor to have zero delay.

Cost-effective robot sensor

The ultrasonic sensor, we will use the HC-SR04 sensor. It is a cost-effective robot sensor. It can measure the distance between the robot and the obstacle. It can measure the distance between the robot and the obstacle. Obtaining data from the sensor is very simple. We can use the HC-SR04 sensor to have zero delay.

Figure 16: Interconnection between Jetson Nano and Launchpad

The communication between Jetson Nano and Launchpad can be carried out using Serial Port. Jetson Nano can send the data to the Launchpad using Serial Port. Launchpad can send the data to Jetson Nano using Serial Port. Jetson Nano can send the data to Launchpad using Serial Port. Once all the ROS based data processing is completed, the Jetson Nano can send the data to Launchpad. Launchpad can be assigned with interfacing to motor driver and sensors.

Figure 17: Interconnection between Jetson Nano and Launchpad

The communication between Jetson Nano and Launchpad can be carried out using Serial Port. Jetson Nano can send the data to Launchpad using Serial Port. Launchpad can send the data to Jetson Nano using Serial Port. Jetson Nano can send the data to Launchpad using Serial Port. Once all the ROS based data processing is completed, the Jetson Nano can send the data to Launchpad. Launchpad can be assigned with interfacing to motor driver and sensors.

Figure 18: Interconnection between Jetson Nano and Launchpad

Programming the Tiva C Launchpad

The programming of the Tiva C Launchpad is done using Energia IDE, which is the customized version of the Arduino IDE for the Launchpad. Follow the Tiva Launchpad guide with Arduino, you can choose the Tiva Launchpad board and the serial port.

Figure 19: Energia IDE

The communication is advised in the Chapter 3_Control_Motor_Encoder_Using_Tiva_C_Launchpad. Follow the Tiva C guide, which is a kind of some important commands for the Launchpad.

Here are features: Pins of the Tiva C Launchpad, Serial Port, Interrupts, Timers, Analog to digital converter, Digital to analog converter, Pulse Width Modulation.

The Microcontroller library is used to handle serial data from the PC.

Figure 20: Serial Port

In the following code, the first line is the object of the Serial Port. The second line is the serial port baud rate. The second one is the object of the Serial Port. The second one is the serial port baud rate.

```
HardwareSerial Serial(0);
int baud = 9600;
HardwareSerial Serial2(2);
int baud2 = 9600;
```

The following is the main function of the code. It will read sensor values and publish the sensor values back to the main stack. The speed of the motors are handled from the main function.

```
void loop()
{
  //Read from Serial
  if (Serial.available()>0)
  {
    //Read message
    ReadSerialPort();
    ParseMessage();
    ClearVariables();
  }
  //Read encoders
  noInterrupts();
  update_Encoder();
  interrupts();
  //Update motor speed
  update_motor_speed();
  //Send sensor data back
  sendSensorData();
}
```

We can compile the code and upload it into the launchpad using Energia. If the upload is successful, you can open the serial monitor using the Ctrl+Shift+M. Type 1 and hit enter to start the sensor data. Type 0 and hit enter to change the parameters using following command:

```
$ LEFT_SPEED 100 RIGHT_SPEED 100
```

```
$ SETUP_PID 10 100 1 0.01
```

We can communicate with the based using the following:

```
$ LEFT_SPEED 100 RIGHT_SPEED 100
```

```
$ SETUP_PID 10 100 1 0.01
```

If everything is successful, you will get a format like these:

Figure 21: The serial port values from the launchpad

The message that you are seeing can be decoded like this. The first character is the message type. It represents, from what the letters mean.

Letter	Meaning or Message
m	Motor speed
f	Flow
e	Encoder
p	PID variables
s	Sensor
t	Motor speed
i	PID actual

The serial messages are separated by spaces and tabs. The end of the line is an end of message.

Figure 22: Serial port values

Note

Source code for the Tiva C Launchpad interfacing can be found here: https://github.com/prasenjit-net/slam-robot

Interfacing robot hardware with ROS

In this section, we will see how to interface motors and sensors with the Launchpad. The attached controller can be used to interface motor drivers and sensors. Using Launchpad, we can have a dedicated controller. The ROS robot controller will communicate with the Launchpad. To Launchpad, we can send the ROS navigation stack. The ROS robot controller will receive the sensor data from the Launchpad. The ROS robot controller will send the motor speed to the Launchpad. The robot controller receives the sensor data from the Launchpad and calculates the distance traveled by the robot. The ROS robot controller will send the motor speed to the Launchpad.

The Clearpath robot controller is a ROS based robot controller. It is a very good robot controller. You can check our robot ready controller here: https://github.com/prasenjit-net/slam-robot/tree/master/src/robot_control

- robot_control_main.py: This is the ROS driver implements the PID controller. It subscribes to the robot target velocity and publishes the robot velocity.
- Twist_Twist_Stamp.py: This is the ROS driver implements the PID controller. It subscribes to the robot target velocity and publishes the robot velocity.
- Twist_Twist_Stamp_Serial.py: This node subscribes to Twist messages and publishes to serial port. It subscribes to serial port and publishes to Twist messages.
- Twist_Twist_Stamp_Serial_Encoder.py: This node subscribes to Twist messages and publishes to serial port. It subscribes to serial port and publishes to Twist messages.
- Encoder_Odom.py: This node subscribes to encoder data and publishes to Odom messages.
- Twist_Twist_Stamp_Serial_Encoder_Odom.py: This node subscribes to Twist messages and publishes to serial port. It subscribes to serial port and publishes to Twist messages.
- Twist_Twist_Stamp_Serial_Encoder_Odom_Sensor.py: This node subscribes to Twist messages and publishes to serial port. It subscribes to serial port and publishes to Twist messages.
- Twist_Twist_Stamp_Serial_Encoder_Odom_Sensor_Motor.py: This node subscribes to Twist messages and publishes to serial port. It subscribes to serial port and publishes to Twist messages.
- Twist_Twist_Stamp_Serial_Encoder_Odom_Sensor_Motor_Imu.py: This node subscribes to Twist messages and publishes to serial port. It subscribes to serial port and publishes to Twist messages.
- Twist_Twist_Stamp_Serial_Encoder_Odom_Sensor_Motor_Imu_Tf.py: This node subscribes to Twist messages and publishes to serial port. It subscribes to serial port and publishes to Twist messages.
- Twist_Twist_Stamp_Serial_Encoder_Odom_Sensor_Motor_Imu_Tf_Ultrasonic.py: This node subscribes to Twist messages and publishes to serial port. It subscribes to serial port and publishes to Twist messages.
- Twist_Twist_Stamp_Serial_Encoder_Odom_Sensor_Motor_Imu_Tf_Ultrasonic_Ekf.py: This node subscribes to Twist messages and publishes to serial port. It subscribes to serial port and publishes to Twist messages.

calculates the distance traveled by the robot. It then publishes as the odometry and **transformation (TF)** topic.

Here is the graph showing the communication between the nodes:

Figure 19: Communication among ROS driver nodes

Here is the list of ROS launch files that we need in order to work with the actual robot. All launch files are placed in the `chefbot_bringup/launch` folder:

- `robot_standalone.launch` This will launch the ROS driver nodes of Chefbot.
- `model_robot.launch` This launch file loads the URDF file of Chefbot.
- `view_robot.launch` This will display the robot model on Rviz.
- `keyboard_teleop.launch` This will start the keyboard teleop node, which can drive the robot using a keyboard.
- `laserscan.launch` This will launch OpenNI to enable depth camera drivers. There may changes to this launch file according to the sensor.
- `gmapping_demo.launch` This will launch the `gmapping` nodes, which will help us map the robot environment.
- `amcl_demo.launch` This will launch the AMCL nodes, which help us localize the robot on the map.
- `view_navigation.launch` This will visualize the map and robot, which helps us command the robot to move to the destination on the map.

Note

Gribbles Astro camera ROS driver:
http://wiki.ros.org/astra_camera
https://github.com/orbbec/ros_astra_camera

Running Chefbot ROS driver nodes

The following is the block diagram of the connection. Make sure that you are all with connecting the devices. Make sure you have connected all sensors and the Launchpad board to your PC before running the driver.

Figure 20: Block diagram of the Chefbot

If we want to launch all driver nodes at the robot, you can simply do it using the following command. Don't forget to change the serial port permission.

```
$ roslaunch
chefbot_bringup
robot_standalone.launch
```

If everything working fine, you will get the following ROS topics:

Figure 21: The Chefbot driver topics

You can also visualize the ROS computational graph using `rqt_graph`. Here is the visualization of `rqt_graph`, showing the communication between all nodes:

Figure 22: The computation graph view of Chefbot driver nodes

Gmapping and localization in Chefbot

After launching the ROS driver, we can bringup the robot using keyboard teleop. We can use the following command to start keyboard teleoperation:

```
$ roslaunch
chefbot_bringup
keyboard_teleop.launch
```

If we want to map the robot environment, we can start the gmapping launch file like we did in the simulation:

```
$ roslaunch
chefbot_bringup
gmapping_demo.launch
```

You can visualize the map building in Rviz using the following command:

```
$ roslaunch
chefbot_bringup
view_navigation.launch
```

You can build the map by teleoperating the robot around the room. After mapping, save the map as we did in the simulation:

```
$ rosrun map_server
map_saver -f <test_map
```

After getting the map, launch AMCL nodes to perform final navigation. You have to restart all the launch files and start again.

Let's look at the commands to launch the AMCL nodes.

First, start the ROS driver nodes using the following command:

```
$ roslaunch
chefbot_bringup
robot_standalone.launch
```

Now start the AMCL nodes:

```
$ roslaunch
chefbot_bringup
amcl_demo.launch
map_file:=<test_map.yaml
```

Then start Rviz to command the robot on the map:

```
$ roslaunch
chefbot_bringup
view_navigation.launch
```

You will see Rviz showing something like the following screenshot, in which you can command the robot and the robot can run autonomously:

Figure 23: Localization and navigation with Chefbot

The following diagram shows the actual robot hardware. As per our design, we can see circular plate and hollow tubes to add additional layers to the robot. You can also see the **Intel NUC** and **Kinect** camera for robot navigation:

Figure 24: The actual Chefbot prototype

Questions

- How to convert encoder data to estimate the robot's position?
- What is the role of SLAM in robot navigation?
- What is AMCL and why is it used?
- What is the importance of the ROS navigation stack?

Summary

In this chapter, we designed and built an autonomous mobile robot from scratch. The design of the robot started with its specification. From the specification, we designed various parameters of the robot, such as motor torque and speed. After finding out the robot specifications, we selected the robot chassis and simulated the robot and actuators to the embedded board. We wrote the firmware of the embedded board. The board can communicate with the PC on which the ROS is running. The ROS driver node receives the data from the robot and interfaces with the gmapping and AMCL packages to perform autonomous navigation.

In the next chapter, we will see how to create a self-driving car and interface to Robot Operating System.

Previous Chapter
Next Chapter