

Robot Operating System basics

Vaios Papaspyros
Mobile robotics group
EPFL

1 Introduction

The goal of this practical work is to introduce the student to the Robot Operating System (ROS) and some of its basic capabilities. We will go through the basics of ROS by: (1) building a simple 2-wheeled robot with respect to the unified formats used in ROS, and (2) implementing a velocity control algorithm of your choice to move the robot. Throughout the course we will be discussing the architecture of ROS. We will also be discussing the development paradigms adopted by the ROS community. The control algorithm will be developed either in C++ or Python and tested with unknown inputs at the end of the course. Following the completion of the course, you will have to prepare a report summarizing your implementation choices.

2 Overview of ROS

ROS is an open-source, meta-operating system, containing multiple tools and libraries for developing robotics applications. Its main purpose is to provide a common ground for sharing and reusing robotics tools, and thus, allowing the development of more advanced applications on top of the readily available tools. ROS resembles an operating system in terms of services provided, i.e., hardware abstraction, low-level device control, message passing between processes and package management. However, it also shares a lot of characteristics with middleware systems (e.g., loose message coupling, etc) and frameworks (e.g., message callbacks, various general utilities, etc). It imposes very little policy on developers, and thus, allows for easy code reuse (e.g., the use of ROS-agnostic libraries) and testing. In the following sections we will provide a short overview of the basic ROS components.

2.1 ROS: Under the hood

As briefly mentioned in the previous section ROS is a loosely coupled system where each process is called a node and each node is responsible for a single task. The loose coupling is derived from the fact that nodes can in fact communicate with each other using messages passing through logical channels, called topics. Therefore, the nodes have the ability to send or retrieve data to/from other nodes through a publish/subscribe model (Fig. 1).

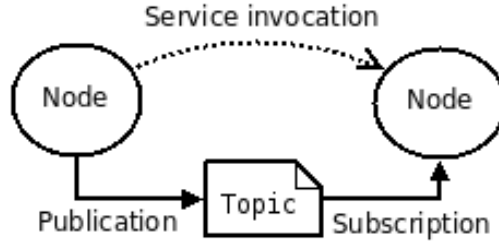
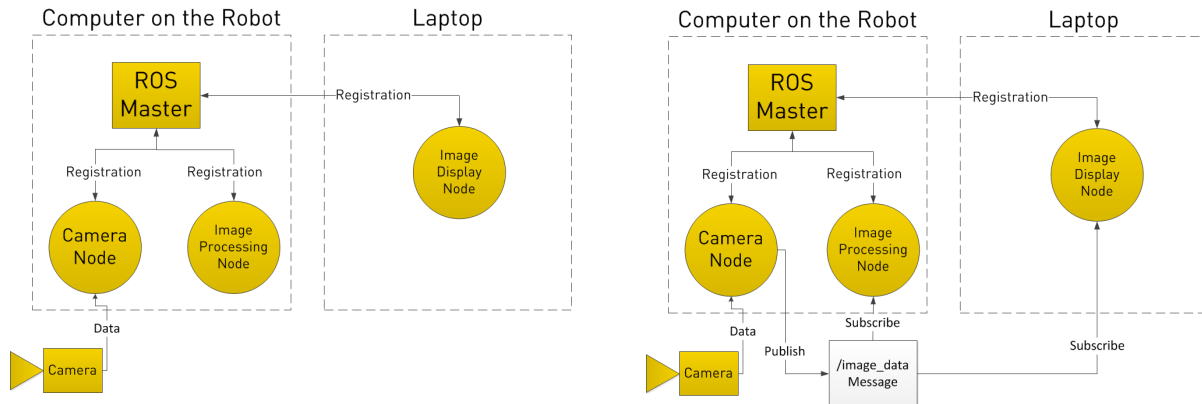


Figure 1: Publisher/Subscriber concept. Individual nodes can communicate through messages published in topics. <http://wiki.ros.org/ROS/Concepts>

This loosely coupled logic is managed by the ROS Master. The Master is responsible for registering new nodes in the system and for facilitating in the lookup of others (see Fig. 2a). Without the Master, it would be impossible for one node to reach another. Although, every node is dependent on the Master (i.e., they need to know how to reach him), it is independent on the physical location of another. This means that nodes can be run in a distributed way, i.e., in different computers and remote locations (see Fig. 2b) as long as they know the location of the Master and have registered their presence with him. Once the registration is complete the two (or more nodes) can directly communicate without passing data through the Master. This design choice opens up a lot of possibilities for distributed computation, multi-robot/swarm systems, etc. For example, think of the problem of formation coordination in a swarm of drones. Each drone would be responsible to control its flight, to avoid obstacles, to capture video or sensory feedback, but the formation itself is controlled from a remote computer that has knowledge of each drone’s position and can fuse data to devise a plan.



(a) A remote node registering its name to the ROS Master. (b) A remote node directly subscribed to another (remote) node's topic.

Figure 2: ROS Master registration scheme. <https://robohub.org/ros-101-intro-to-the-robot-operating-system/>

So far we have summarized the “plumbing” of ROS, that is, the message passing infrastructure that allows for distributed computation. Apart from the plumbing, ROS is often divided in three more branches, namely, the tools, capabilities and ecosystem. The tools branch concerns all the logging, debugging, visualizing, development, etc, tools that

are compatible and ship with ROS. The capabilities concern higher level concept applications (perception, control, machine intelligence, etc) that are built on top of the plumbing and tools. Last but not least, the ecosystem consists of the strong focus on integration and documentation that aims to make the science and engineering behind multiple robotics applications readily available. In the duration of this course we will emphasize on the plumbing and tools, to some extent on the capabilities. You will also have to interact with the ROS ecosystem in order to acquire API information, documentation, or simply theoretical background about specific applications. You are highly encouraged to do so.

3 Overview of the Practical

- Prior to the 1st session:
 - Lab: –
 - Homework:
 1. Study [Overview of ROS](#)
 2. [ROS installation & preparation](#)
- 1st week:
 - Lab:
 1. [Overview of ROS](#)
 2. [Building a robot model](#)
 3. [Velocity control](#) – part A: interacting with the differential control plugin
 - Homework:
 1. Polish your design
 2. Fully integrate the differential drive
- 2nd week:
 - Lab:
 1. [Velocity control](#) – part B: code subscribers/publishers & implement a velocity control algorithm
 - Homework:
 1. (optional) [Obstacle avoidance](#) – part B: add proximity sensors & implement an obstacle avoidance algorithm
 2. [Report & evaluation](#)

4 ROS installation & preparation

You will need to have a computer with ROS installed **prior** to the first session. It is recommended to have a native installation of ROS (especially if you need it for other courses), but it can be a virtual machine as long as you have allocated enough resources. For this course it is recommended that you use the [ROS Melodic Morenia](#) long-term support (LTS) distribution. We will not cover any distribution specific features, therefore, you can use older distributions as well (e.g., if you already have one installed or your OS is not compatible with the newest version). For the installation, please follow the excellent guide offered in the ROS wiki. You are also encouraged to read the introduction [1] and experiment with the tutorials [2] to have some degree of familiarity before the session. It is also suggested that you pick a code editor of your choice (e.g., Visual code studio or sublime text) to benefit from the code highlighting.

5 Building a robot model

During the first session we will focus on building a small wheeled robot (see examples in Fig. 3) with a design of your choice. This will provide you with fundamental knowledge of how unified robot model formats operate (i.e., what information must be defined, how to define joints, links, etc), how to generalize your models to allow for easier scaling and how to visualize them. We will also be discussing how to retrieve environmental variables when simulating the robot.

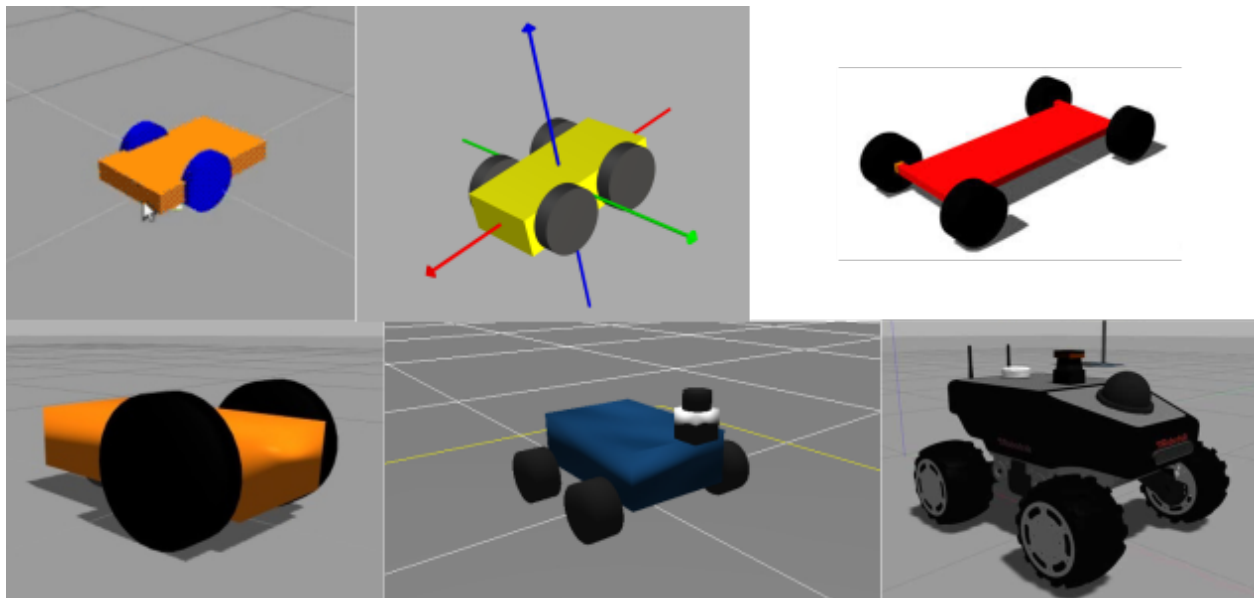


Figure 3: Wheeled robot model examples.

What we will cover:

- ROS packages

- xacro and URDF/SDF files
- launch files
- RViz and Gazebo

5.1 Steps to follow

1. Create a ROS workspace if you don't already have one, for example:

```
$ cd <to a directory of your choice or at your home folder>
$ mkdir -p ros_practicals_ws/src
```

2. Create a ROS package with a name of your choice. From now on we will be developing our models and algorithms within this package:

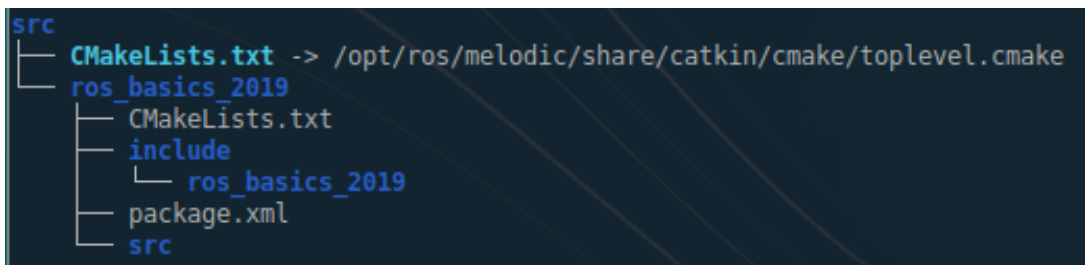
```
$ cd <to your catkin workspace>/src
$ catkin_create_pkg ros_basics_2019 roscpp rospy
```

The above command will create a directory called “ros_basics_2019” that has dependencies to roscpp and rospy. Those two dependencies contain all the necessary packages to build simple programs in C++ and Python. Throughout the course we will have to add more dependencies to use different components of ROS.

If you did not export the ROS source command in your bashrc/zshrc, you might need to issue the following and repeat the previous step:

```
$ source <ros install directory>/setup.sh
```

Your workspace's src folder should now look something like this:



```
src
├── CMakeLists.txt -> /opt/ros/melodic/share/catkin/cmake/toplevel.cmake
├── ros_basics_2019
│   ├── CMakeLists.txt
│   ├── include
│   │   └── ros_basics_2019
│   └── package.xml
└── src
```

If you are having trouble refer to the ROS wiki tutorial for creating packages [3].

3. Before you go further, issue the following command while at workspace's root:

```
$ catkin_make
```

This will invoke the CMakeLists.txt files under the packages contained in the current workspace and will produce two folders with the output of the compilation process (that is, build and devel).

4. Open the **package.xml** file with an editor. Complete your team's information by adding the corresponding XML tags. Near the end of this file, which is called the package manifest, you can see the dependencies we added to the package in step 2. In the future steps you might need to manually add dependencies in order to build and run your code successfully. Notice that these dependencies are not only referring to the building process but the runtime as well.
5. Before we start designing our robot model we need some background on robot description formats. In ROS (and other robotics frameworks), robots are described in unified formats that allow for easier sharing and consistency across the community. One such format is the Unified Robotic Description Format (URDF), which is an XML file format used to describe the elements of a robot. The URDF allows us to encapsulate kinematic and dynamic properties of the robot, as well as the hierarchical relation of how the elements are connected. However, in the URDF it is not possible to specify the pose of the robot, joint loops, friction, and other useful properties. Therefore, modern simulators are pushing the use of yet another XML based format, the Simulation Description Format (SDF), which improves on the URDF weaknesses. ROS uses URDFs by default, however, there are ROS tools for converting from and to the SDF. Once again, you can refer to tutorials of ROS [4] and Gazebo [5].

Another very useful package when working with large XML files is **xacro**. Xacro is practically a scripting mechanism that facilitates the modular design of robot descriptions and the code reuse when defining a URDF model. In ROS, which needs URDF models to operate, a xacro file is used to define macros and allow for more generalized description files before it is converted to a URDF [6].

6. Create a directory named **urdf** in your package folder.
7. Create a **robot_description.urdf.xacro** file under the newly created urdf folder and open the file with an editor (it is suggested that you use an editor that supports XML highlighting).
8. Open your .xacro file. Within this file you will define the materials, properties, shapes, collision objects of a 2-wheeled robot. Construct a robot from simple geometrical shapes and provide estimations about the inertial properties and dimensions of each element. It is recommended that you design a small robot similar to the thymio [7] (see Fig. 4), but with simpler geometrical shapes (feel free to try more complex designs if this interests you).
9. Once you are finished working with an element it is recommended that you convert it to URDF and visualize it to validate your design. To convert your .xacro into a URDF, navigate to the location of your .xacro and issue the following command:



Figure 4: (a) Detailed thymio robot model in RViz, and (b) Simple thymio robot model in RViz.

```
$ rosrunc xacro xacro \
    robot_description.urdf.xacro > robot_description.urdf
```

If your XML is correct, then this process should not yield any warnings on your terminal. Now that you have your robot model, you need to get the ROS Master up and running to be able to use the visualization tools. Open up a terminal and issue start the Master:

```
$ roscore
```

Before you start the ROS Visualization (RViz) program to see your model, you still need to inform ROS of a few things. Firstly, you need to export the necessary paths in your Linux environment, so that ROS can find your code, models, etc. For this, navigate to the root of your catkin workspace and issue the following command:

```
$ source devel/setup.sh
```

Subsequently, you need to let the parameter server [8] know your model's specifications. The parameter server can be thought of as a multi-variate dictionary that is accessible through the network. ROS nodes can use this server to store or retrieve parameters at runtime. However, the parameter server is not designed for high performance and is, therefore, mainly used for storing and retrieving configuration parameters. RViz will look for the **robot_description** parameter upon startup. With the Master running, you can check if this parameter is defined:

```
$ rosparam list
```

Unless you were experimenting with your model prior to issuing this command, the `robot_description` parameter should not be listed in the parameter server. Let us now set a value for this parameter that matches our robot model:

```
$ rosparam set robot_description \
    "`cat <path to urdf folder>/robot_description.urdf`"
```

You can quickly validate that the parameter was set by listing the parameters of the server or by asking for this specific parameter:

```
$ rosparam get robot_description
```

Now you are ready to launch RViz:

```
$ rviz
```

When you launch RViz you will be presented with an empty world, so go ahead and add your robot model by clicking on **Add** → **Robot Model**.

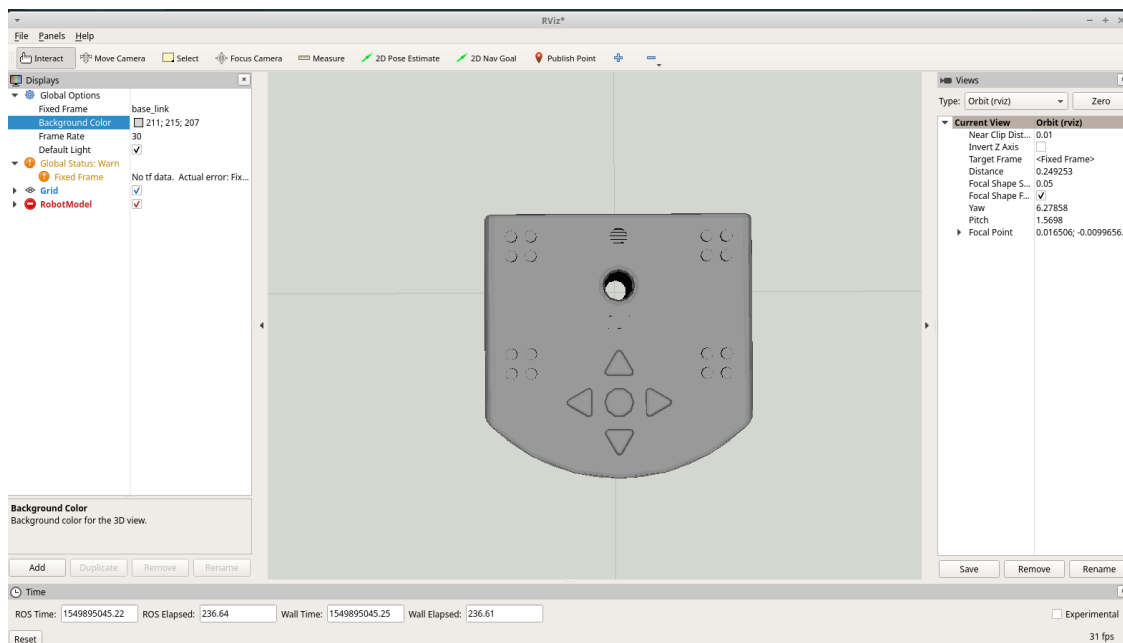


Figure 5: Visualizing a robot with RViz.

The above steps can be summarized in a single script, called a launch file. ROS launch files are a very common and convenient way to start multiple nodes and the Master and even setting parameters without having to issue commands one-by-one every time you may need to run your application. Let us write a launch file to start RViz and load the robot model. First you need to create yet another folder under your package with the name **launch**, as follows:

```
# while in your package root directory
$ mkdir launch
```


Under the **launch** directory, create a new script with the name **robot_description.launch** and open it in any code editor and paste the following information:

```
<launch>
  <param name="robot_description" command="$(find xacro)/xacro '$(
    find ros_basics_2019)/urdf/robot_description.urdf.xacro'"/>

  <!-- for fake joint values -->
  <node name="joint_state_publisher" pkg="joint_state_publisher"
    type="joint_state_publisher">
    <param name="use_gui" value="False"/>
  </node>

  <node name="robot_state_publisher" pkg="robot_state_publisher"
    type="state_publisher"/>

  <!-- launch the rviz node -->
  <node name="rviz" pkg="rviz" type="rviz" />
</launch>
```

Then launch this file by issuing the following command:

```
$ roslaunch ros_basics_2019 robot_description.launch
```

10. Instead of RViz you may use the [Gazebo simulator](#). Go ahead and create a new launch file (e.g., robot_description_gazebo.launch) and add the following:

```
<launch>
  <param name="robot_description" command="$(find xacro)/xacro '$(
    find ros_basics_2019)/urdf/robot_description.urdf.xacro'"/>
  <arg name="x" default="0"/>
  <arg name="y" default="0"/>
  <arg name="z" default="0.05"/>

  <!-- Start Gazebo with an empty world loaded -->
  <include file="$(find gazebo_ros)/launch/empty_world.launch">
    <arg name="paused" default="false"/>
  </include>

  <!-- Spawn your robot model -->
  <node name="mybot_spawn" pkg="gazebo_ros" type="spawn_model" output
    ="screen"
    args="-urdf -param robot_description -model <robot name> -x
      $(arg x) -y $(arg y) -z $(arg z)" />
</launch>
```

Notice how in both launch files we have integrated the xacro to URDF conversion in a single command. You can also pass execution parameters to your nodes, such as configuration of the simulation world, variables, etc.

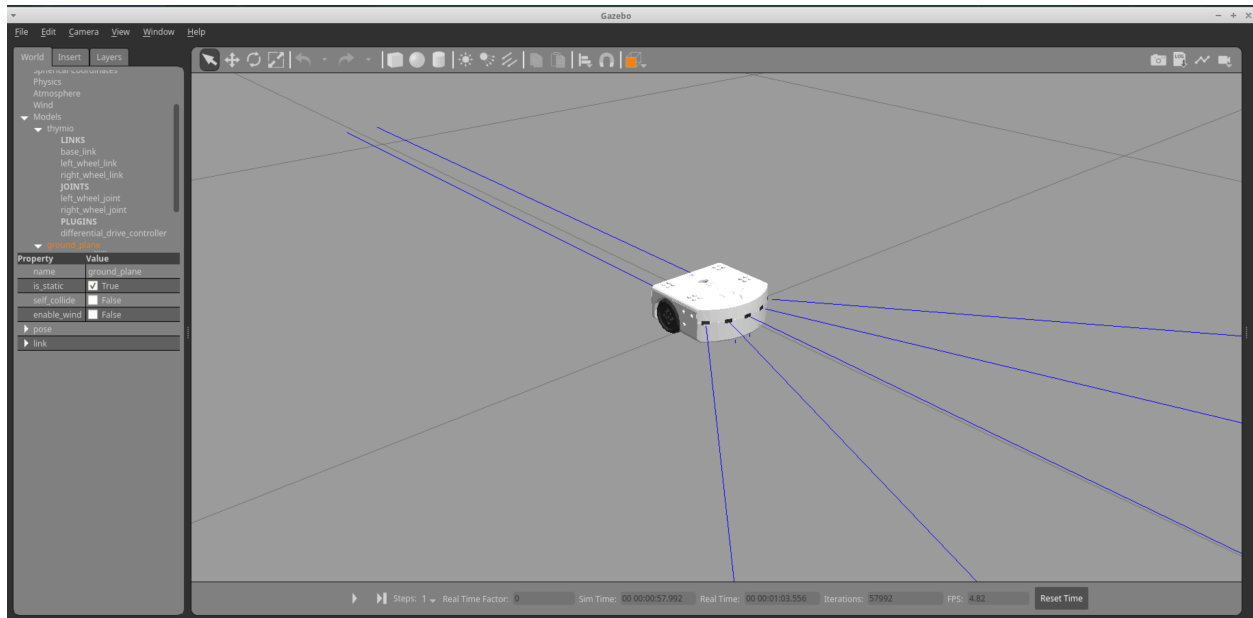


Figure 6: Visualizing a robot with Gazebo.

11. Experiment with the previous steps until you are confident that you have a functional robot model.

6 Velocity control

By this point you should have a simple robot model, that so far can not move. It is possible to implement the control of the robot in various ways, but in this practical course we will make use of Gazebo's [differential drive plugin](#). The plugin requires only a minor addition to your robot description, but is only compatible with Gazebo. Once you have managed to get the differential drive working, you are going to implement a velocity control algorithm to follow trajectories given to the robot.

What we will cover:

- Gazebo's differential drive plugin
- The `cmd_vel` topic
- Subscribers/Publishers

6.1 Steps to follow

1. Before moving forward you are encouraged to study [9] and the corresponding section about differential drive. Once you are feeling confident about differential drive, go ahead and add the following to your robot model (within the `<robot>` tags):

```
<gazebo>
  <plugin name="differential_drive_controller" filename="
    libgazebo_ros_diff_drive.so">
    <alwaysOn>true</alwaysOn>
    <updateRate>20</updateRate>
    <leftJoint>YOUR_LEFT_WHEEL_JOINT</leftJoint>
    <rightJoint>YOUR_RIGHT_WHEEL_JOINT</rightJoint>
    <wheelSeparation>XXX</wheelSeparation>
    <wheelDiameter>YYY</wheelDiameter>
    <wheelTorque>ZZZ</wheelTorque>
    <commandTopic>cmd_vel</commandTopic>
    <odometryTopic>odom</odometryTopic>
    <odometryFrame>odom</odometryFrame>
    <robotBaseFrame>YOUR_BASE_FRAME</robotBaseFrame>
  </plugin>
</gazebo>
```

You will need to adjust the values according to your design. Notice the tag `<commandTopic>`. This tag instructs the robot to accept velocity commands from the topic `cmd_vel` and then Gazebo's differential drive plugin will distribute the velocities in the wheels so that your robot moves along the requested vector.

2. After integrating the differential drive, launch Gazebo and spawn your robot (like we did in 5). Make sure your simulation is not paused and then publish a simple velocity command to test your differential drive. To do so, you need to publish a message to the `cmd_vel`, which accepts messages of the type `geometry_msgs/Twist` (see http://docs.ros.org/api/geometry_msgs/html/msg/Twist.html):

```
$ rostopic pub /cmd_vel geometry_msgs/Twist "linear:
  x: 1.0
  y: 0.0
  z: 0.0
angular:
  x: 0.0
  y: 0.0
  z: 0.0"
```

The above command instructs the robot to move on the x-axis with a constant rate of 1.0 m/s. If your model is correct you should see your robot moving along this axis.

3. Alternatively to publishing velocity commands with Twist messages, you can use the `teleop_twist_keyboard` node to send commands through your keyboard. First, make sure you have this package installed or install it. For ROS Melodic Morenia:

```
$ sudo apt-get install ros-melodic-teleop-twist-keyboard
```

Then, you need to start the `teleop_twist_keyboard` node:

```
$ rosrun teleop_twist_keyboard teleop_twist_keyboard.py
```

Follow the instructions provided in the shell to control your robot.

4. Once you are familiar with the use of Twist messages and your robot's differential drive, you will need to build a trajectory following and velocity control algorithm. Create a **`my_robot_control.{cpp or py}`** file under your package's `src` folder. Notice that if you write your code in C++ you will need to also implement a `CMakeLists.txt` file ([you will find a sample at the end of the publisher/subscriber tutorial](#)). In this file you will have to develop your algorithm. Your program (node), should at least subscribe to a topic named **`goal_pos`** where we will be sending the goal positions for the robot. This topic should contain messages of your own custom type or of the type `geometry_msgs/Point`. Your node **must** be able to accept goal positions at **any rate** (regardless if you reached your previous goal or not) and follow every point of the trajectory sequentially. You may also subscribe to other topics (e.g., `odom`) that give you information about the robot's pose, position, etc. This knowledge is necessary for the robot to follow the trajectory correctly.

Your algorithm must include the following:

- Subscribe to the `odom` topic to get information about the pose of the robot. This topic contains messages of the type [nav_msgs/Odometry.msg](#). Make sure to study the contents of this message type and their corresponding units.
- A listener that stores incoming goal positions in a list.
- A control loop that is responsible for setting the appropriate velocities to reach the next available goal position. Once the goal is reached (at least within a sensible threshold), the goal is removed from the list and the next waypoint will be processed. If all waypoints are reached, then the robot will not move. Once you have calculated the desired velocity for one timestep, you will publish a `geometry_msgs/Twist` message to the `cmd_vel` topic.
- Your control loop should implement at least a basic PID controller. That is, the angular and linear velocity of your robot should be regulated according to the robot's pose compared to the waypoint. In our case the robot has only two degrees of freedom. It can rotate (around z-axis) and move (along the x-axis).

For this exercise you need to be careful regarding the synchronization of your code:

- The control loop and the subscriber callback are running **asynchronously**. This means you might face race conditions. Therefore, make sure to be “defensive” in your code (e.g., make use of mutexes).
5. Create a launch file that starts your node(s) (you can also start Gazebo or any other tools you want to use).
 6. In your report you will have to describe your algorithm and provide graphs of acceleration/deceleration for random points that you give to the robot. Try at least one point for: small, medium and big distances (proportional to your robot’s size). You will also need to state any parameters that you use (e.g., gains, thresholds, etc) to control the robot motion.

7 Obstacle avoidance

So far you have constructed a robot model that you can move through Gazebo’s differential drive plugin. However, your robot is still lacking one important feature, that is, sensors. Modern robots are equipped with multiple sensors that allow them to perceive aspects of their environment and take intelligent actions. This is a fundamental of the development of autonomous systems. Therefore, in this section we will be adding one or more proximity sensors to our robot and implementing an obstacle avoidance algorithm.

What we will cover:

- Gazebo’s laser sensor plugin
- Subscribers/Publishers

7.1 Steps to follow

1. First, you will need to add sensor links to your robot (add as many as you like, but keep in mind that it will be more difficult to write an efficient obstacle avoidance algorithm when dealing with a lot of sensors). Define a small rectangle (or any shape you like) as a sensor case and position it in your model. At this point your sensors are part of the robot structure but do not have any actual functionality.
2. To simulate the sensor functionality you will need to add additional references to your URDF and bind them to the objects you created in step 1. Gazebo (like most simulators) offers a wide range of sensors that you can integrate in your robot. One such sensor, that will be useful for our obstacle avoidance section, is the laser sensor. Once again, in order to access Gazebo’s sensors, you will need to load its [sensor plugin](#). Your code should look something like this:

```
<gazebo reference="YOUR_SENSOR_LINK">
  <sensor type="ray" name="laser_right">
    <pose>0 0 0 0 0 0</pose>
    <ray>
```

```

        <!-- ray parameters -->
    </ray>
    <plugin name="laser" filename="libgazebo_ros_laser.so" >
        <topicName>YOUR_ROBOT_LASER/scan</topicName>
        <frameName>YOUR_ROBOT_LASER_LINK</frameName>
    </plugin>
    <always_on>1</always_on>
    <update_rate>10</update_rate>
    <visualize>true</visualize>
</sensor>
</gazebo>

```

Notice that the plugin will publish measurements in a custom topic. Those will be used both for visualization and for your obstacle avoidance algorithm. We suggest that you set a small sample of rays to simplify your case (beware that this will make the robot more susceptible to collisions).

3. You can easily validate that your sensor is working once you have the plugin setup and publishing data. Launch Gazebo and make sure that you have visual confirmation that your plugin is running (see Fig. 7).

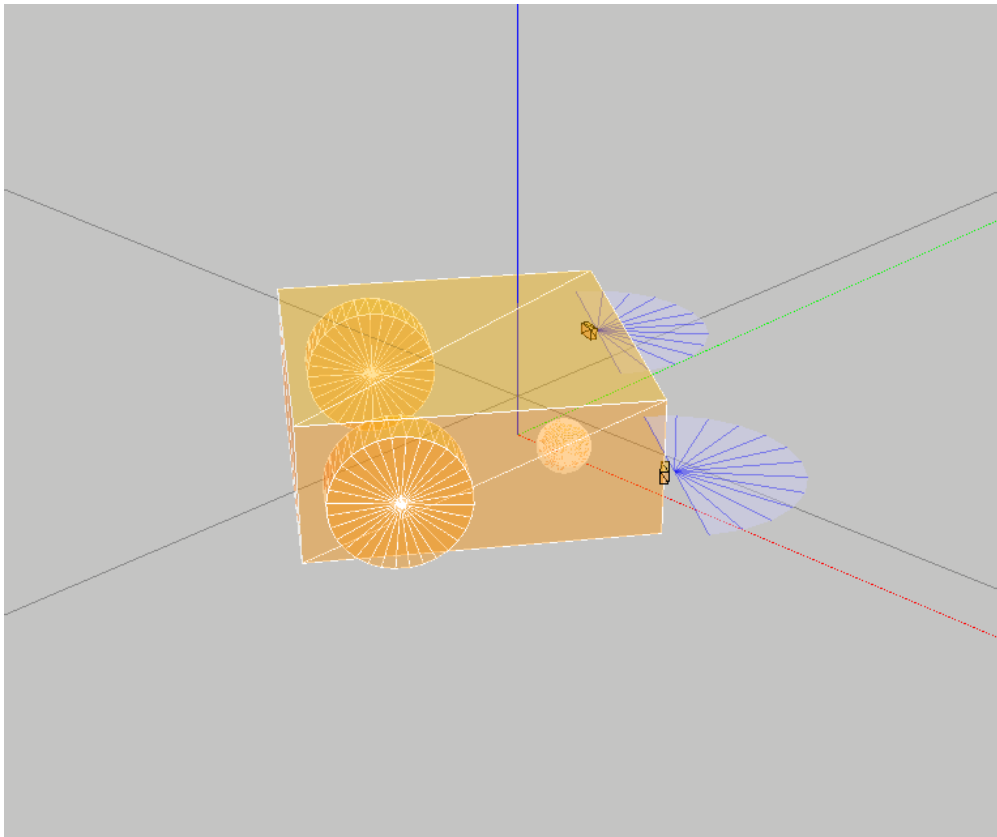


Figure 7: Thymio-like robot with two laser sensors running.

4. You can also add a few random obstacles in front of your robot and launch RViz where you can **Add** → **By topic** and select your sensors' topics. Now you should be able to see the ray collisions (see Fig 8)

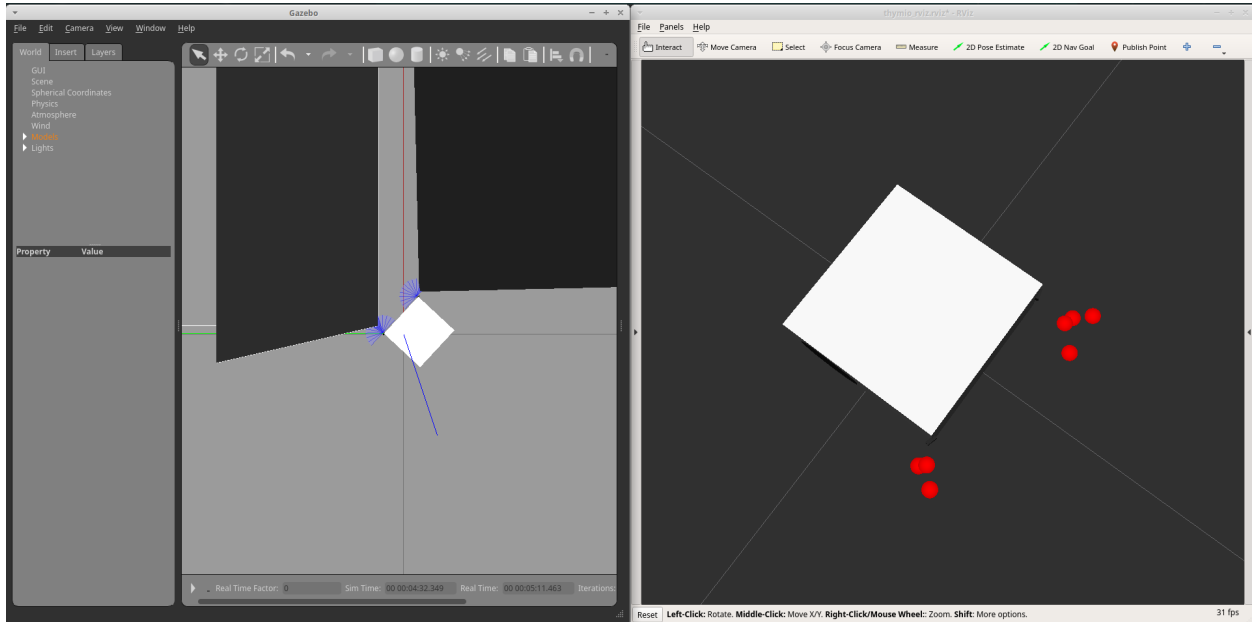


Figure 8: Thymio-like robot facing obstacles (left) and sensing them with its laser sensors (right).

5. Once you are sure that your sensors are working properly you will need to implement an obstacle avoidance algorithm. For this task you are free to choose which algorithm to implement. It could be a simple Braitenberg obstacle avoidance (see more on the Braitenberg vehicle in [9]) or a custom of your own design. Notice that more sophisticated algorithms will be graded with a higher mark.
6. To test your algorithm you can use Gazebo's worlds or build one. Your algorithm will also be tested and evaluated in an unknown map.

8 Report & evaluation

You will have one week after the practical work to submit your report with the following suggested structure:

- Introduction: state the problem, give sufficient background information, outline the scope of the report.
- Robot model: give a description of your robot design choices and justify them where necessary.
- Velocity control: give a description of your control algorithm implementation, provide figures or measurements if available.

- Discussion: discuss and conclude your work.
- Suggestions (optional): it is the first time that this practical is taking place, please tell us if and how it can be improved.
- Appendix A: add your robot model code.
- Appendix B: add your control algorithm code
- Appendix C: provide instructions for launching examples of your work and mention your ROS distribution if you are not using Melodic Morenia.

The evaluation will be based on the following table:

	Comments	Weight	Mark
Accomplishments <ul style="list-style-type: none"> • Goals reached • Quality of results • Q&A (discussion) 		45%	
Methodology <ul style="list-style-type: none"> • Systematic approach • Understanding of the subject • Personal contribution 		20%	
Working style <ul style="list-style-type: none"> • Autonomy • Communication with assistants • Timeliness 		10%	
Report <ul style="list-style-type: none"> • Structure (hierarchy, structure, etc) • Completeness • Clarity (general from, graphs, captions, etc) 		25%	

Table 1: Evaluation criteria & weights.

References

- [1] *ROS Introduction*, available at <http://wiki.ros.org/ROS/Introduction>.
- [2] *ROS Tutorials*, available at <http://wiki.ros.org/ROS/Tutorials>.
- [3] *Creating a ROS Package*, available at <http://wiki.ros.org/ROS/Tutorials/CreatingPackage>.
- [4] *urdf package*, available at <http://wiki.ros.org/urdf>.
- [5] *Tutorial: Using a URDF in Gazebo*, available at http://gazebosim.org/tutorials/?tut=ros_urdf.
- [6] *Using Xacro to Clean Up a URDF File*, available at <http://wiki.ros.org/urdf/Tutorials/Using%20Xacro%20to%20Clean%20Up%20a%20URDF%20File>.
- [7] *Thymio specifications*, available at <https://www.thymio.org/en:thymiospecifications>.
- [8] *Parameter Server*, available at <http://wiki.ros.org/Parameter%20Server>.
- [9] Ben-Ari, M. and Mondada, F., *Elements of Robotics*, (Springer International Publishing, 2018).