EE3305/ME3243
Robotic System Design
Manual of Project 1
Implementation of PID Control in ROS

Teaching Assistant: Guo Haoren haorenguo_06@u.nus.edu

Instructor 1: Dr. Andi Sudjana Putra andi_sp@nus.edu.sg

Instructor 2: A/Prof. Prahlad Vadakkepat prahlad@nus.edu.sg

AY 2023/2024 Semester 1

October 8, 2023

Contents

1 About the Project

1. Students are expected to simulate a Turtlebot3 robot in an empty world with a single pillar. Students are expected to apply a proportional-integral-derivative (PID) control in the simulation using Robot Operating System (ROS).
2. The location of the pillar is according to the last three digits of the matriculation number of each student.
   - A student whose matriculation number ends with digits xyz should set the pillar location at $(3 + x/2,\ 3 + y/2)$. For example, a student whose matriculation number is $A0012345B$ should set the pillar location at $(3 + 3/2 = 4.5, 3 + 4/2 = 5)$.
3. The height and the diameter of the pillar is 0.1 and 0.2, respectively.
4. Students are expected to move the Turtlebot3 robot towards the pillar and then to make it stop at a predetermined distance from the pillar. The distance away from the pillar is according to the last digit of matriculation number of each student.
   - A student whose matriculation number ends with digit xyz should set the predetermined distance as $(0.5 + z/10)$. For example, a student whose matriculation number is A0012345B should set the predetermined distance at $0.5 + 5/10 = 1.0$.
   - The Turtlebot3 should face the pillar.
5. Students are expected to describe the PID algorithm and explain the rationale of the PID tuning. Students are expected to analyse the performance of the PID control.

2 Learning Objectives

At the end of the project, students are expected to demonstrate their ability to:

1. Apply a PID control in a ROS simulation.
2. Tune PID control gains and explain the rationale.
3. Present and analyse the performance of the control system.

3 Submissions (Demo, Report and Code)

A **demo** is required to show students running ROS. In the demo, students may be asked to show his/her original simulation and/or to change the location of the pillar.

Report should be titled "1_PID_[YourName].pdf". The content of the report is as follows:

1. Initial conditions.

   a) Show the initial location of the pillar and the Turtlebot3 of your setting. Use the view that best show their locations.
   b) Calculate the initial distance and orientation of the pole with respect to the Turtlebot3.

2. Implementation of PID control. Discuss how the PID control is implemented in ROS with reference to your code.
   a) Describe how you (1) define the integral term, (2) define the derivative term and (3) define the PID control term.
   b) Describe the purpose of the code to (1) regularize the angular error (`error_angle`) and (2) limit the angular control signal (`trans_angle`).

3. Tuning of PID.
   a) Discuss the tuning process, e.g., which gain is determined first, which gain is determined second, how it is determined and so on.
   b) Discuss how you would characterise the PID control (P, PI, PD or PID). Discuss the merits, demerits, and other points that you want to highlight about your design.

4. Performance of PID control.
   a) Attach the plots of errors vs time (both linear and angular errors) that represents your best design.
   b) Analyse the performance, e.g., overshoot, steady state error and settling time.

5. Conclusions and key learning points.

The **code** submission should contain the zip of the entire **package** (`a345b_pid` in the above example). As you may be sharing a machine, it is possible that the name of the package is the same between you and your pair. However, you should indicate your name and NUSNET ID clearly in the following files:

1. The launch file. In this example, `a345b_pid.launch`.

2. The PID control node, i.e., `BotControl.cpp` file.

3. The simulation and PID parameters, i.e., `config.yaml`.

4 Preparation (for personal machines, not applicable to machines in the lab)

1. Install ROS Noetic.

2. Install Turtlebot3 in ROS Noetic as follows:

```
# husky and turtlebot dependencies
cd ~
sudo apt install ros-noetic-husky-gazebo ros-noetic-turtlebot3-* --yes

# write to .bashrc
echo "export TURTLEBOT3_MODEL=burger">> ~/.bashrc
source ~/.bashrc
```

3. Optional: Install Sublime Text editor using the command:

```
sudo snap install sublime-text --classic
```

5 File Structure of ROS Simulation

The file structure of ROS simulation is presented in Figure 1.

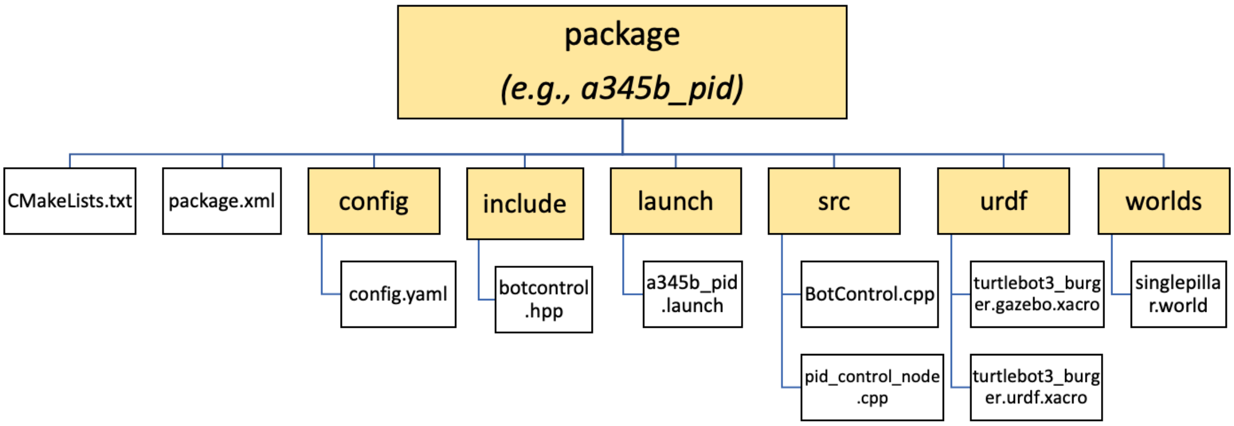

Figure 1: File structure of ROS simulation

The files to download are available at Canvas as follows:

1. `singlepillar.world`
2. `turtlebot3_burger.gazebo.xacro`
3. `turtlebot3_burger.urdf.xacro`
4. `botcontrol.hpp`
5. `templateBotControl.cpp`, where you need to modify and rename to `BotControl.cpp`
6. `pid_control_node.cpp`

6 Steps

1. Create a workspace and build it.
   At the Home directory (symbolised ~), create your own **workspace** with a unique name, such as using a letter and the last 4 characters of matriculation number. (If your matriculation number is $A0012345B$, the workspace name is `a345b_pid_ws`, where `pid` is the title of Project 1 and `ws` is the customary ending for a workspace.) Use the following commands:

```
cd ~
mkdir a345b_pid_ws
cd a345b_pid_ws
mkdir src
cd ~/a345b_pid_ws
catkin_make
```

2. Create a package.
Inside the /src directory of the workspace, create a package using unique name, such as using a letter and the last 4 characters of matriculation number as above. If your matriculation number is $A0012345B$, the package name is a345b_pid. Use command `$ catkin_create_pkg package_name` as follows:

```
# In ~/a345b_pid_ws/src
catkin_create_pkg a345b_pid std_msgs roscpp
```

When the package has been successfully built:
   - A directory whose name is a345b_pid will be generated.
   - Inside the package, two files are generated: CMakeLists.txt and package.xml.

3. Generate a world.
Create a directory called 'worlds' in the package (hint: use command `$ mkdir directory_name`) as follows:

```
# In ~/a345b_pid_ws/src/a345b_pid
mkdir worlds
```

Download the world file singlepillar.world as provided and place it in the worlds directory.

4. Create a launch file to launch the world.
Create a directory called launch in the package. Create a launch file in the launch directory using a unique name, such as using a letter and the last 4 characters of matriculation number as above. (If your matriculation number is $A0012345B$, the launch file name is a345b_pid.launch). Use command `$ touch file_name` to create a file as follows:

```
# In ~/a345b_pid_ws/src/a345b_pid/launch
touch a345b_pid.launch
```

Note: In this project, you will start multiple nodes simultaneously. Launch file is a tool to start multiple nodes as well as setting parameters. A launch file is written in xml as .launch.

In this launch file, you will:
   a) Launch an empty_world from its source in gazebo_ros package under /launch/empty_world.launch.
   b) Launch your own singlepillar.world within empty_world.

Use command `$ subl file_name` to open the file as follows:

```
# In ~/a345b_pid_ws/src/a345b_pid/launch
subl a345b_pid.launch
```

Type the following code (by changing a345b_pid with your own package):

```xml
<?xml version="1.0"?>
<!--
  EE3305/ME3243
  Name: YOUR NAME
  NUSNET ID: YOUR NUSNET ID
-->

<launch>

<!-- Launch the world in Gazebo. -->
    <arg name="student_pkg_path" value="$(find a345b_pid)"/>
    <arg name="world_name" default="$(arg
    ↪  student_pkg_path)/worlds/singlepillar.world"/>
    <include file="$(find gazebo_ros)/launch/empty_world.launch">
        <arg name="world_name" value="$(arg world_name)"/>
    </include>

</launch>
```

Save the file.

5. Build the package.
   In the workspace, use command `$ catkin_make` to build the package. In the above example, the workspace is `~/a345b_pid_ws` , hence:

```
# In ~/a345b_pid_ws
catkin_make
```

6. Source own workspace.
   Source your own workspace. Go to workspace and use command `$ source ./devel/setup.bash` as follows:

```
# In ~/a345b_pid_ws
source ./devel/setup.bash
```

   To check that sourcing to your workspace has happened, you can go to the workspace and use command `$ echo $ROS_PACKAGE_PATH`. It should point to your own workspace.

7. Launch the program.
   From the workspace, launch all nodes using command `$ roslaunch package_name launch_file`. In this example, the workspace is `~/a345b_ws`, the package is `a345b_pid` and the launch file is `a345b_pid.launch`, hence:

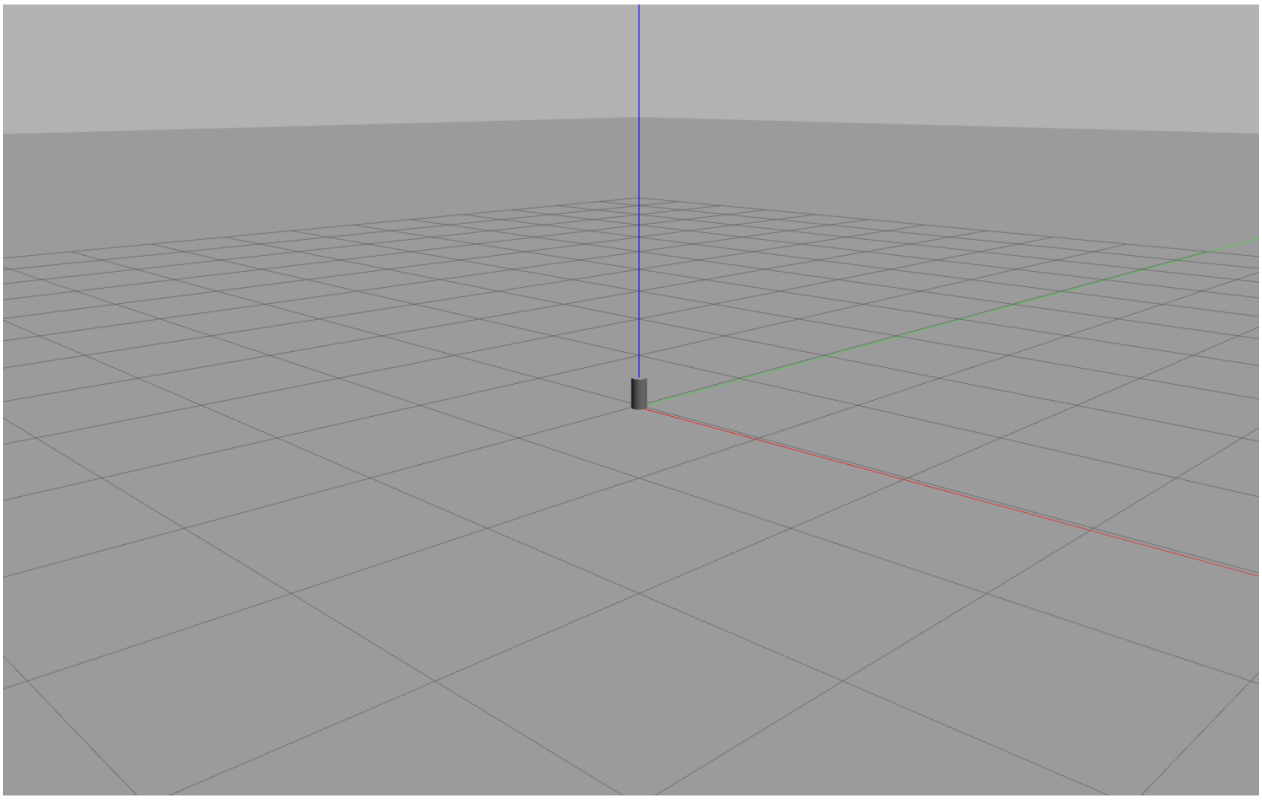

Figure 2: The world with the single pillar at (0,0)

```
# In ~/a345b_pid_ws
roslaunch a345b_pid a345b_pid.launch
```

A window should appear showing the world with a pillar at (0,0), as shown in Figure 2.

8. Change the pillar location.
   The location of the pillar is defined in the world file singlepillar.world and in the config.yaml file (the latter will be defined later). The file singlepillar.world is located in the directory worlds inside the package.

   Open the singlepillar.world as follows:

   ```
   # In ~/a345b_pid_ws/src/a345b_pid/worlds
   subl singlepillar.world
   ```

   The location of the pillar is defined in line 14 by the sentence
   <pose frame=''>X Y Z roll pitch yaw</pose>. Change the pillar location accordingly, i.e., change the X and the Y.

9. Copy the robot model of Turtlebot3.
   Create a directory called urdf in the package (hint: use command $ mkdir directory_name) as follows:

   ```
   # In ~/a345b_pid_ws/src/a345b_pid
   mkdir urdf
   ```

Download the model files `turtlebot3_burger.gazebo.xacro` and `turtlebot3_burger.urdf.xacro` and place them in the `urdf` directory.

10. Include the Turtlebot3 robot in the launch directory.
    Open the launch file inside the directory `launch` as follows:

```
cd ~/a345b_pid_ws/src/a345b_pid/launch
subl a345b_pid.launch
```

Add the code to launch the Turtlebot3 at (0,0) is as follows:

```
... // under the statements that launch world

<!-- spawn turtle -->
    <arg name="model" default="burger" />
    <arg name="x" default="0"/>
    <arg name="y" default="0"/>
    <arg name="yaw" default="0"/>
    <param name="robot_description" command="$(find xacro)/xacro --inorder
    ↪  $(find turtlebot3_description)/urdf/turtlebot3_$(arg
    ↪  model).urdf.xacro" />
    <node pkg="gazebo_ros" type="spawn_model" name="spawn_urdf"  args="-urdf
    ↪  -model turtlebot3_$(arg model) -x $(arg x) -y $(arg y) -Y $(arg yaw)
    ↪  -param robot_description" />

</launch>
```

11. Launch the program.

    a) You need to build the package. (Hint: Step 5 of this manual.)
    b) Make sure you have sourced your own workspace. (Hint: Step 6 of this manual.)
    c) From the workspace, launch all the nodes. (Hint: Step 7 of this manual). A window should appear showing the world and the Turtlebot3 robot.
    d) **Take a screenshot showing the pillar in the correct location (according to your matriculation number) and the Turtlebot3 in the initial position (refer to Figure 3). You will need it for your report.**

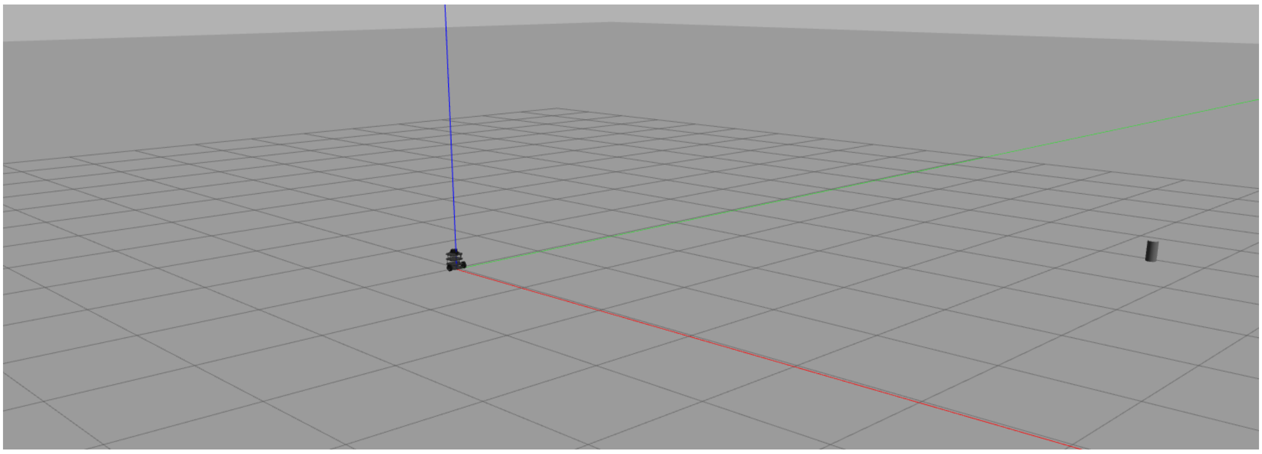

Figure 3: The world with single pillar at (4.5,5) and Turtlebot3 at (0,0)

12. Import the definition of the variables to drive the Turtlebot3 robot.
    Look for a directory called `include` in the package.

    Download the file `botcontrol.hpp` as provided and place it in the `include` directory.

    This file defines how the Turtlebot3 robot receives inputs and provides outputs. Open the file `botcontrol.hpp` and study the file. You will find:

    a) The packages used in the simulation are included, namely:
       - `sensor_msgs`
       - `nav_msgs`
       - `geometry_msgs`
       - `std_msgs`
       - `gazebo_msgs`

    b) Variables are classified into private variables and public variables.

    c) Private variables include:
       - The variables used for localisation and for PID control.
       - The topics that the simulations subscribes and the topics that it publishes are defined (refer to the private variables in the program).

    d) Public variables include:
       - The variables used to tune the PID control
       - The variables to assign the targeted distance of the Turtlebot3 robot from the pillar and the location of the pillar
       - Please note these public variables as you will need to define and tune them in a file `config.yaml` later.

13. Import a node to run the PID control.
    Inside the package, look for a directory `src` to store your node to run the PID control. To create the node, you will download 2 files to the directory `src` as follows:

    a) `templateBotControlPID.cpp`

    b) `pid_control_node.cpp`

Rename templateBotControlPID.cpp into BotControl.cpp. The file BotControl.cpp defines the PID control. You need to insert your own code to BotControl.cpp to apply PID control.

Open the renamed file BotControl.cpp. (Hint: use command subl). You will find:

a) The file botcontrol.hpp is included.
b) **Insert your name and NUSNET ID in the space provided at the start of the code.**
c) The subscribers and the publishers have been included (refer to botcontrol.hpp).
d) The variables of the PID controls have been initialised. Note that the initial values are zero.
e) The PID control algorithm is mostly defined in void BotControl::pidAlgorithm(). You can study it while referring to the Lecture Notes to refer to what each variable means.
f) Under void BotControl::pidAlgorithm(), find the lines // ENTER YOUR CODE HERE and // END OF YOUR CODE HERE. **This is where you should enter the code to implement the PID control.**

pid_control_node.cpp runs the PID algorithm in botcontrol.cpp. Open the file to study it. Note that the name of the node is assigned as pid_control_node.

14. Create a config file.
Inside the package, create a directory called config (hint: use command $ mkdir) as follows:

```
# In ~/a345b_pid_ws/src/a345b_pid
mkdir config
```

Under the directory config, create a file config.yaml (hint: use command $ touch) as follows:

```
# In ~/a345b_pid_ws/src/a345b_pid/config
touch config.yaml
```

Open the file config.yaml (hint: use command $ subl). This is the file where you tune the PID control. Write as follows:

a) Write your name and NUSNET ID.
b) Key in all public parameters (refer to Step 12) of the PID control.

The syntax is as follows:

```yaml
# EE3305/ME3243
# Name: YOUR NAME
# NUSNET ID: YOUR NUSNET ID

Kp_f: 0
Ki_f: 0
# enter the rest of the variables (11 variables total)
...
```

dt can be assigned 0.1 throughout the simulation. target_angle can be assigned 0 throughout the simulation. target_distance, pillar_x and pillar_y are assigned according to the values assigned to you.

As a start, assign the PID gains zero.

15. Modify the CMakeLists.txt file.
 In the package, open CMakeLists.txt file as follows:

```
# In ~/a345b_pid_ws/src/a345b_pid
subl CMakeLists.txt
```

Most of its statements are commented, i.e., inactive. You will uncomment statements that you need.

First, find find_package . Uncomment it (by removing the # sign) and add the following packages:

a) roscpp (as C++ is used)
b) Packages included in BotControl.hpp (refer to Step 11), namely sensor_msgs, nav_msgs, geometry_msgs, std_msgs and gazebo_msgs.

The resulting statements should be as follows:

```
find_package(catkin REQUIRED COMPONENTS
        roscpp
        sensor_msgs
        nav_msgs
        geometry_msgs
        std_msgs
        gazebo_msgs
)
```

Second, find catkin_package and add the same dependency packages as in find_package. The resulting statements should be as follows:

```
catkin_package(
        INCLUDE_DIRS include
        CATKIN_DEPENDS
                roscpp
                sensor_msgs
                nav_msgs
                geometry_msgs
                std_msgs
                gazebo_msgs
)
```

Third, as you have include directory in the package, uncomment include_directories. The resulting statements should be as follows:

```
include_directories(
        include
        ${catkin_INCLUDE_DIRS}
)
```

Fourth, uncomment add_executable and include the files that define the node in the src directory, namely pid_control_nodes.cpp and BotControl.cpp. The resulting statements should be as follows:

```
add_executable(
        ${PROJECT_NAME}
                src/pid_control_node.cpp
                src/BotControl.cpp
)
```

Fifth and lastly, uncomment `target_link_LIBRARIES`. The resulting statements should be as follows:

```
target_link_libraries(${PROJECT_NAME}
        ${catkin_LIBRARIES}
)
```

The `CMakeLists.txt` file should look as follows (excluding comments and unnecessary lines):

```
cmake_minimum_required(VERSION X.X.X)
project(a345b_pid)

find_package(catkin REQUIRED COMPONENTS
        roscpp
        sensor_msgs
        nav_msgs
        geometry_msgs
        std_msgs
        gazebo_msgs
)

catkin_package(
   INCLUDE_DIRS include
   CATKIN_DEPENDS
      roscpp
      sensor_msgs
      nav_msgs
      geometry_msgs
      std_msgs
      gazebo_msgs
)

include_directories(
  include
  ${catkin_INCLUDE_DIRS}
)

add_executable(
   ${PROJECT_NAME}
      src/pid_control_node.cpp
      src/BotControl.cpp
)

target_link_libraries(${PROJECT_NAME}
   ${catkin_LIBRARIES}
)
```

16. Modify the `package.xml` file.
   In the package, open `package.xml` file as follows:

```
# In ~/a345b_pid_ws/src/a345b_pid
subl package.xml
```

   Add <build_depend>, <build_export_depend> and <exec_depend> for each packages included in `CMakeLists.txt` (refer to Step 15), namely `sensor_msgs`, `nav_msgs`, `geometry_msgs`, `std_msgs` and `gazebo_msgs`. The `package.xml` file should look as follows (excluding comments and unnecessary lines):

```xml
<?xml version="1.0"?>
<package format="2">
  <name>a345b_pid</name>
  <version>0.0.0</version>
  <description>The a345b_pid package</description>

  <maintainer email="maluser@todo.todo">maluser</maintainer>
  <license>TODO</license>

  <buildtool_depend>catkin</buildtool_depend>

  <build_depend>roscpp</build_depend>
  <build_depend>sensor_msgs</build_depend>
  <build_depend>nav_msgs</build_depend>
  <build_depend>geometry_msgs</build_depend>
  <build_depend>std_msgs</build_depend>
  <build_depend>gazebo_msgs</build_depend>

  <build_export_depend>roscpp</build_export_depend>
  <build_export_depend>sensor_msgs</build_export_depend>
  <build_export_depend>nav_msgs</build_export_depend>
  <build_export_depend>geometry_msgs</build_export_depend>
  <build_export_depend>std_msgs</build_export_depend>
  <build_export_depend>gazebo_msgs</build_export_depend>

  <exec_depend>roscpp</exec_depend>
  <exec_depend>sensor_msgs</exec_depend>
  <exec_depend>nav_msgs</exec_depend>
  <exec_depend>geometry_msgs</exec_depend>
  <exec_depend>std_msgs</exec_depend>
  <exec_depend>gazebo_msgs</exec_depend>

  <export>
    <!-- Other tools can request additional information be placed here -->
  </export>
</package>
```

17. Modify the launch file.
Open the launch file inside the directory `launch` as follows:

```
# In ~/a345b_pid_ws/src/a345b_pid/launch
subl a345b_pid.launch
```

Add <node> tag to launch the PID algorithm node as follows:

```xml
<node pkg="a345b_pid" type="a345b_pid" name="pid_control_node"
  output="screen"/>
```

where:

- pkg refers to the name of the package

- `type` refers to the name of the executable
- `name` refers to the name of the node

Add the variables to tune the PID control from `config.yaml` as follows:

```
<rosparam command="load" file="$(arg student_pkg_path)/config/config.yaml"/>
```

18. Build the package.
Refer to Step 5.

19. Source own workspace.
Refer to Step 6.

20. Launch.
Refer to Step 7.

    A window should appear showing the world with a pillar and the Turtlebot3 robot, very similar to Figure 3. The Turtlebot3 has not yet moved because all gains are zero.

21. Tune the PID control and run the simulation to arrive at the pillar location.
At this stage, you will need to tune the gains of PID control in `config.yaml` to achieve as good performance as you can. You can check the error plot to assess the performance.

    To obtain the error plot, use the command `$ rqt_plot` as follows:

    - Launch the simulation. Once the simulation is launched, pause it by clicking the "Pause/Play" button at the bottom left of the Gazebo window (refer to Figure 4).
    - Open a new terminal. Use the command `$ rqt_plot`. A new plot window appears.
    - At the top left corner, you can determine the plot that you want to display. Start with `error_forward` plot by typing `error_forward` in the "Topic" field, representing the distance error.
    - Click the "Edit axis" button (refer to Figure 5) and set both X-axes and Y-axes to present the chart clearly. For a start, you can set X-axes between `0` and `60`, and Y-axes between `-10` to `10`.
    - Once ready, resume the simulation by clicking the "Pause/Play" button. As the simulation resumes, the plot will follow.
    - For the `error_angle` plot, repeat the above steps. You may want to change the Y-axis configuration.

Figure 4: Play/Pause button in Gazebo environment

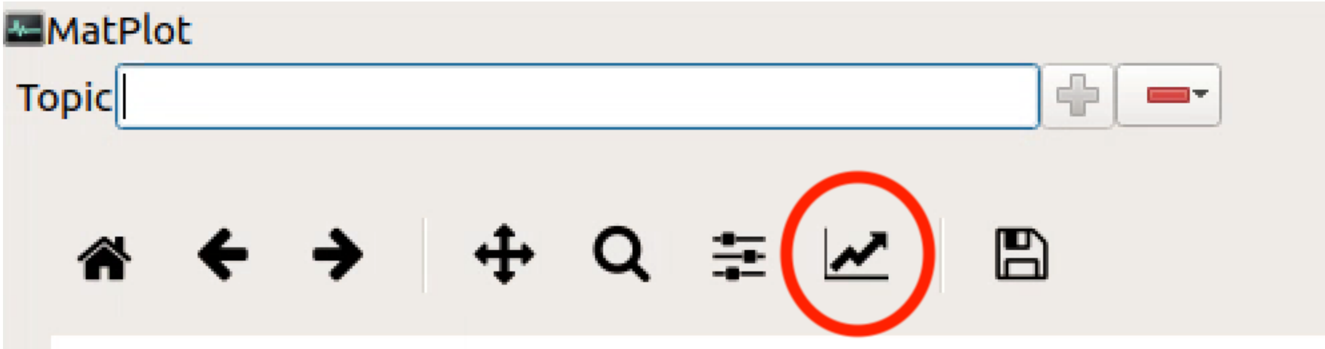

Figure 5: "Edit axis" button in Gazebo environment

22. Capture the plot.
    After you have achieved a satisfactory result, capture the plot of errors vs time (both linear and angular errors).

7 Enrichment

You may systematically explore various PID gains and observe the performance of Turtlebot3.

You may explore the ROS structure. Open a new terminal while the program is running. In the new terminal, use command `$ rqt_graph` to get an overview of the nodes and topics of the simulation.