Lab 1. Perception and locomotion

(Duration: 1 session)

Introduction

Throughout the course, you will be using the CoppeliaSim robot simulator developed by Coppelia Robotics.[1] In this first lab session, you will become familiar with its main features and learn how to remotely control a differential drive robot from a Python script.

Learning outcomes

By the end of the session, you should be able to:

- Use the CoppeliaSim Python Remote API.
- Acquire sensor readings and issue commands to control a differential drive robot.
- Program a wall-following algorithm to explore a corridor environment.

Preparation (1 point)

Read through the statement and take note of anything you do not understand to ask the instructors at the beginning of the session. As suggested in Section 2, it is also advisable to check the documentation for the simulator functions you will have to use in this assignment.

Since you will be working on your personal laptop, make sure to install Python, Visual Studio Code, and CoppeliaSim in advance or, alternatively, the Ubuntu 20.04 virtual machine available in Moodle. Besides, download the code template and try to answer the following questions. You might need to dust off your object oriented programming skills.

1. When is the `pass` keyword used in Python?
2. Can you instantiate the `Robot` class? Why?
3. How are `Robot`'s attributes initialized from `RobotP3DX`? Can `RobotP3DX` objects access them?

Deliverables

You will have to submit two files through **Moodle** one week after the end of the session:

- **A report in PDF format** that contains the developed code —screen captures are not allowed—, justified answers to the proposed questions (including those of the Preparation section), and analyses of the results. The report does not need to be long, but should demonstrate that you worked through the whole statement.
- **A compressed folder**, in `.zip` or `.7z` format, with all your code files and any additional CoppeliaSim environment that you may have created. This second upload serves two purposes: it will allow the instructor to execute your program in order to assess your algorithms' performance; and, most importantly, it will serve as a backup you can resort to when working on your final project.

[1] https://www.coppeliarobotics.com/

1. Introduction to CoppeliaSim

Let's start with the basics of CoppeliaSim before getting to the coding exercises. When you launch the simulator you should see a screen like the one shown in Figure 1.

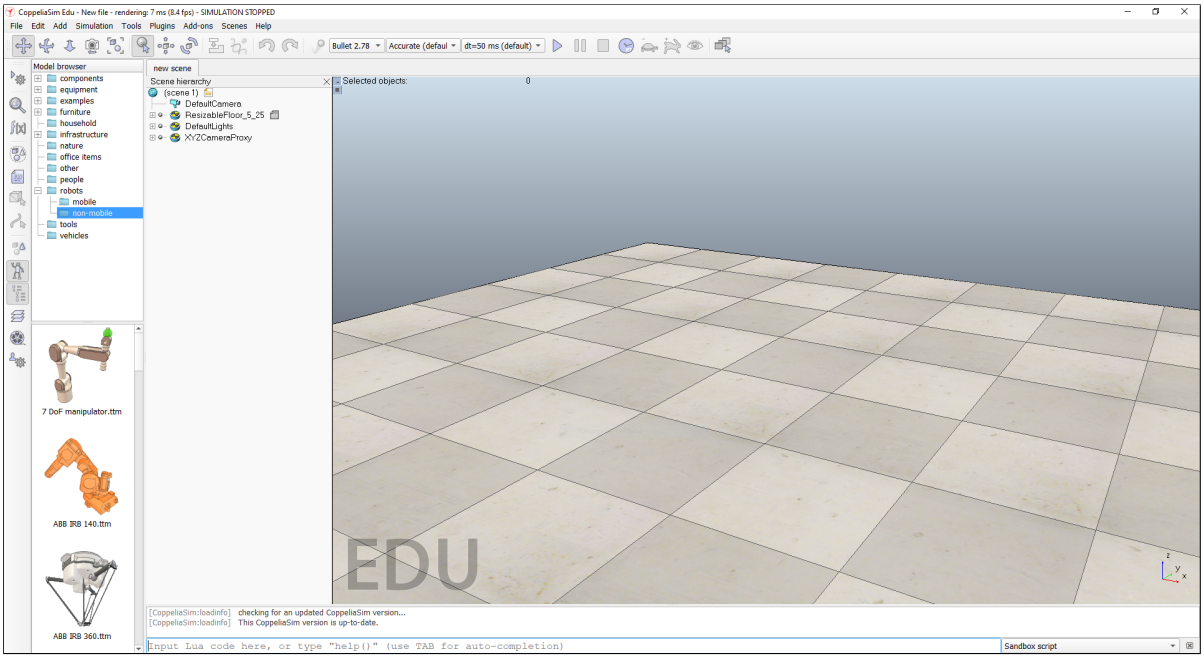


Figure 1. Opening screen of CoppeliaSim.

From the *Model browser* pane on the left of the screen you can select different obstacles and robots to build your environment. Go to robots > mobile > pioneer p3dx.ttm and drag the differential drive mobile robot anywhere on the scene. If you expand the object properties in *Scene hierarchy* you will discover it has two drive wheels and a caster wheel plus 16 overlapping ultrasonic range sensors that cover a 360º field of view (Figure 2).

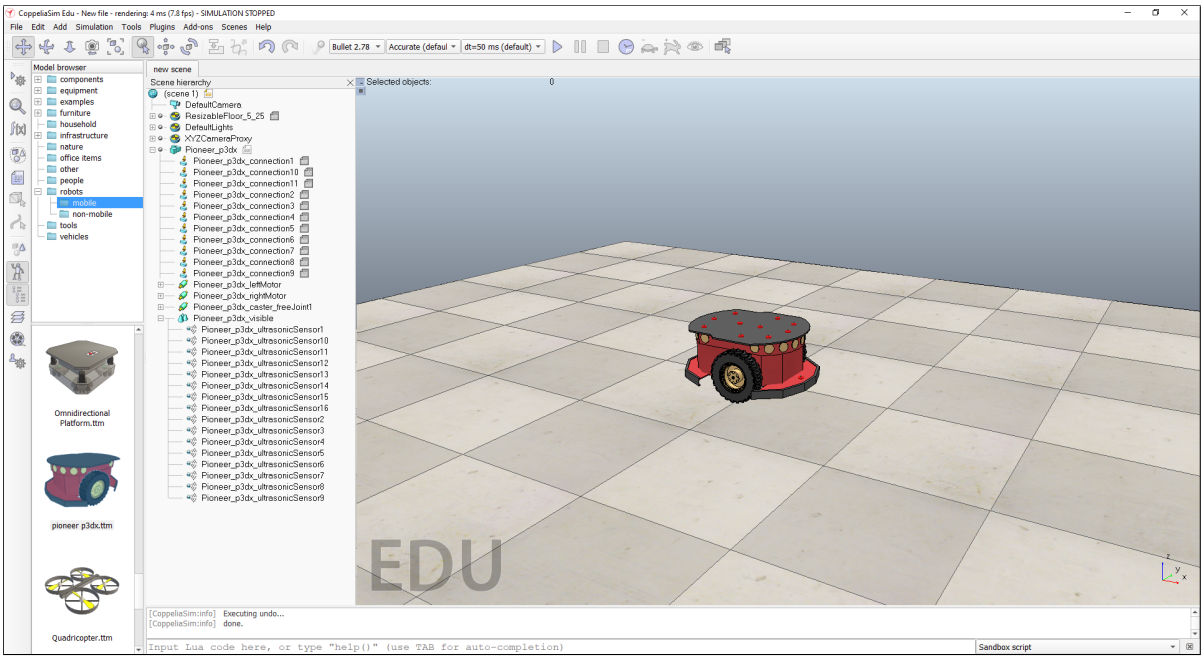
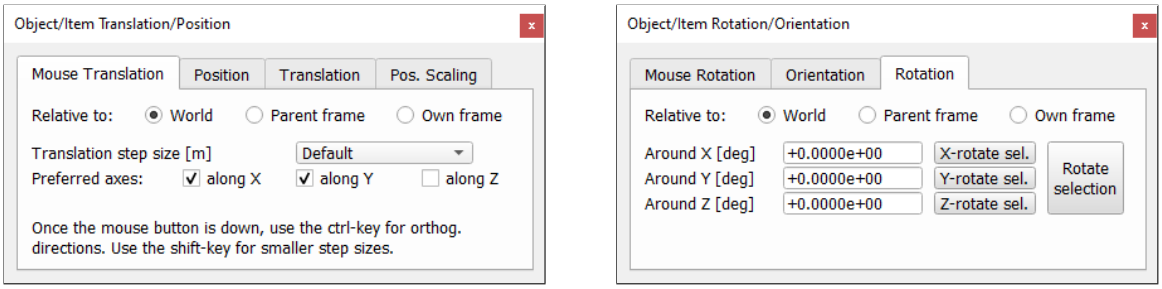


Figure 2. Simulator appearance after adding a Pioneer P3-DX robot.

As you can see, by default the robot is facing right. You can easily change the robot's position and orientation by selecting it in the *Scene hierarchy* and then clicking  or  respectively in the top toolbar. Play around with the different options until you understand how they work (Figure 3). Once you are done, make sure to leave the robot in contact with the floor, not levitating!



(a) **(b)**

Figure 3. Position (a) and orientation (b) dialogs. If the tabs *Mouse translation* or *Mouse rotation* are selected, the object can be freely moved using the mouse.

Now let's add an obstacle. For the sake of this tutorial, we will restrict ourselves to primitive shapes, but feel free to try other alternatives. Go to Add > Primitive shape > Cuboid to insert a 1 m x 0.1 m x 0.5 m obstacle. Then, place the newly created cuboid in front of the robot using the translation and rotation tools. Your scene should resemble Figure 4.

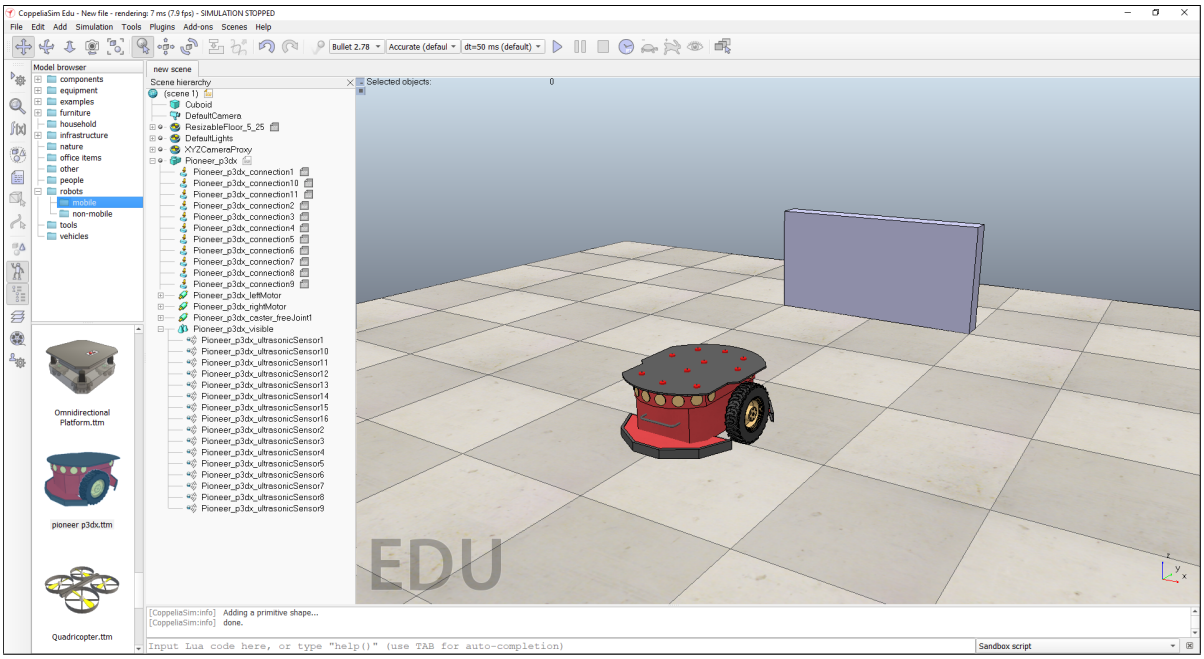


Figure 4. Environment after placing the cuboid.

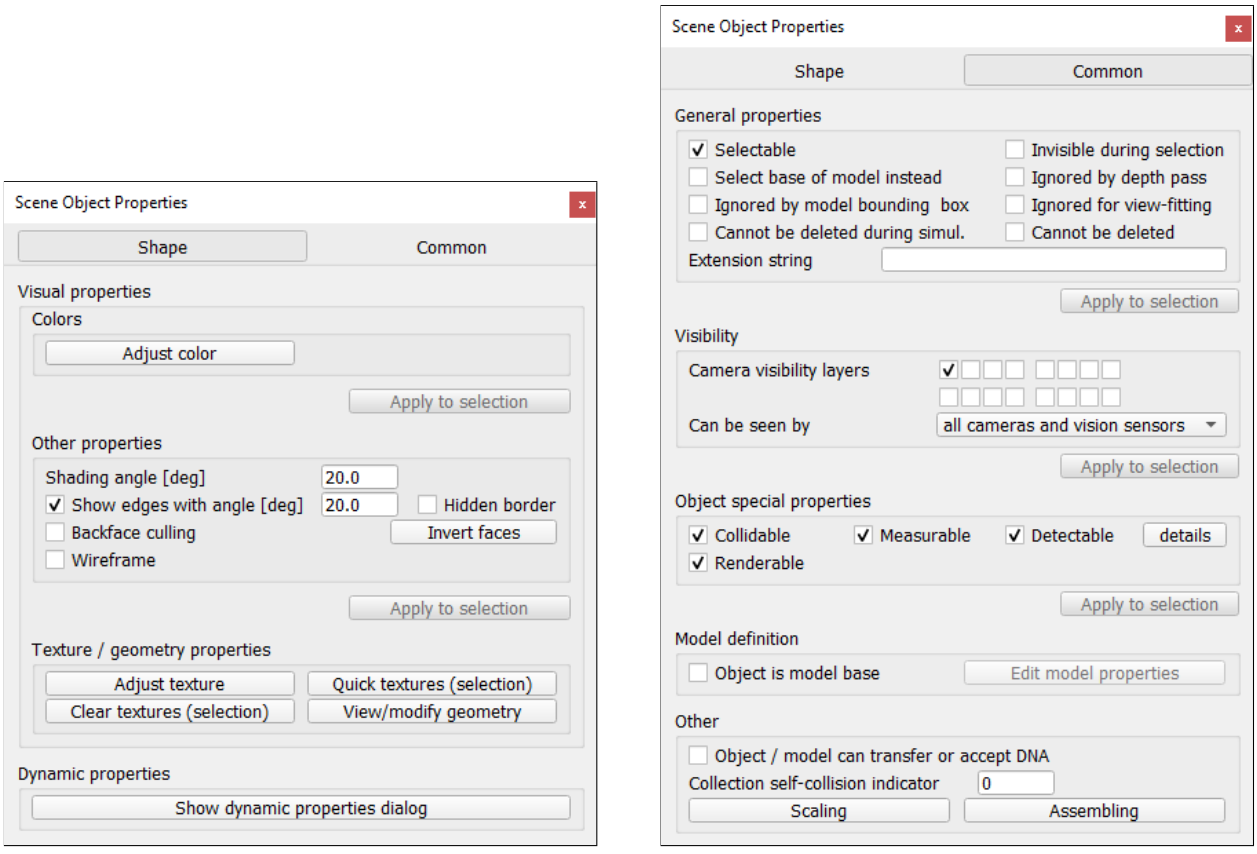
At this stage, if you click  , the robot should start moving forward until it walks into the obstacle. However, it does not come to a halt, but rather it begins pushing the wall until, eventually, both of them fall off the edge of the environment! Wait a minute, why does the robot move? Weren't we going to control it from Python? The reason is that CoppeliaSim can be used as a standalone robot simulator by programming embedded Lua scripts[2] from within the application. Nevertheless, since there are some

[2] https://www.coppeliarobotics.com/helpFiles/en/scripts.htm

limitations to what you can do with these scripts, and given the fact that Lua is scarcely employed by the robotic community, we are going to use Python to code the robot's behavior. Although C/C++, Java, and MATLAB are also supported, Python has been the language of choice because prototyping rather complex algorithms is easier —at the expense of increased computation times compared to other alternatives— thanks to the plethora of available packages. Once the algorithms are tested and their parameters tuned, you typically implement them in C++.

Before disabling the default script that is commanding the robot, let's first adjust a few properties of the obstacle. Check that the simulation is stopped, otherwise you will not be able to modify any parameter. In the *Scene hierarchy*, double click on the icon next to the cuboid object. A dialog with two tabs (Figure 5) should appear.

- The *Shape* tab allows modifying visual (e.g., color, dimension) and dynamic (e.g., mass) properties. For instance, try to make the wall red and 2 m long, and increase its weight enough to prevent the robot from moving it in case of collision. Run the simulation again to verify that the wall stands still.
- The *Common* tab deals with the object's response to external events. Make sure that everything looks like in Figure 5(b) to allow all types of sensors to properly detect it.



(a) **(b)**

Figure 5. Shape (a) and Common (b) object property dialogs.

Since the default script, which implements a simple obstacle avoidance algorithm, is still active, if you execute the simulation once more, the robot will not crash into the wall this time, but will still bungee jump without rope... To remove it, right click on the script icon next to the robot and navigate to Edit ⟩ Remove ⟩ Associated child script (Figure 6).

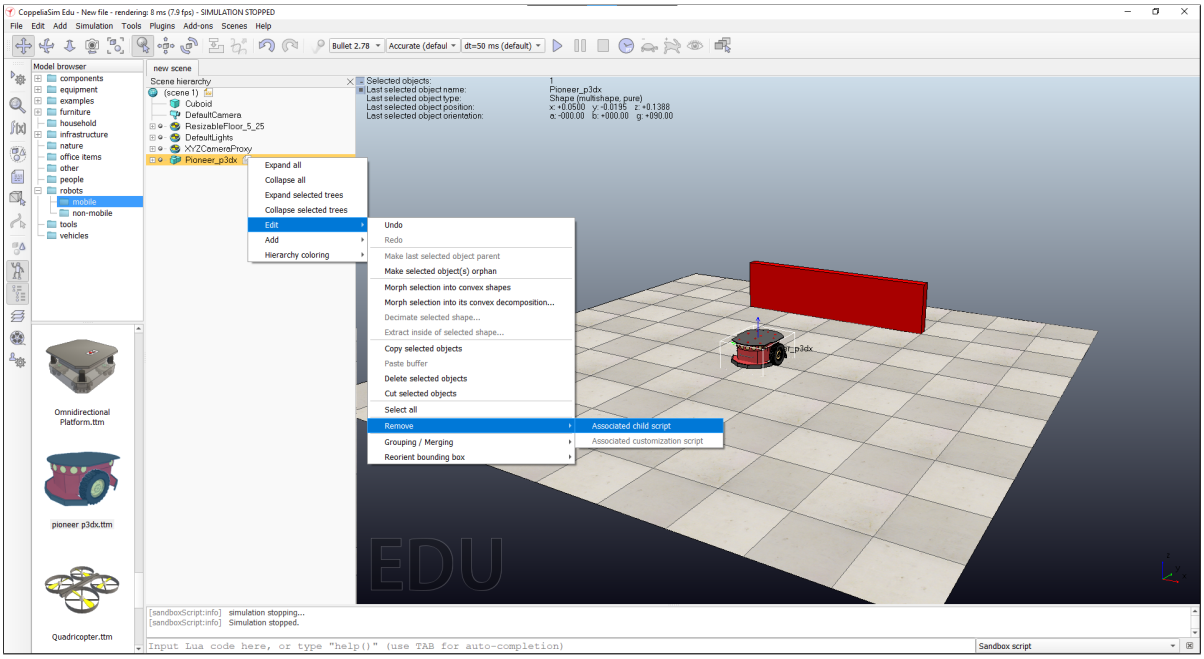


Figure 6. How to remove a robot's default child script.

2. Wandering around (1.5 points)

It is finally time to switch to Python. Download the code template from Moodle —if you had not done it already—, extract it, and open it with Visual Studio Code. `main.py` is the entry point to the program and performs all the necessary API calls[3] to connect to a CoppeliaSim instance running on your machine —remember, there is no place like 127.0.0.1— on port 19997.

You will notice that in `robot_p3dx.py` there are several methods that contain `TODO` comments. Your first assignment is to complete the body of `_init_motors`[4] and `move` to match the expected behavior described in the corresponding docstrings. Nevertheless, you may change the interface if you deem it necessary.

The API functions to acquire the motor handles and set the motor speeds are shown in Code 1. For simplicity, in this particular robot the motors have been configured to ensure that if both angular velocities are positive, the robot moves forward, but this is not the case for every robot model available. Once you are done, run your Python script.

```
# Acquire handles (in _init_motors)
rc, handle = sim.simxGetObjectHandle(client_id, motor_name,
    sim.simx_opmode_blocking)

# Set motor speeds
rc = sim.simxSetJointTargetVelocity(client_id, handle, w,
    sim.simx_opmode_oneshot)
```

Code 1. Remote API function to set motor speeds. Check the documentation for more details.

[3]You can find the complete API documentation for Python at https://www.coppeliarobotics.com/helpFiles/en/remoteApiFunctionsPython.htm. Bookmark this web page, you will be checking it frequently...

[4]A common convention in Python is to name private variables and methods with a leading underscore.

3. Taking a glance at the environment (1.5 points)

Now that your robot is able to move, complete `_init_sensors` and sense to give the robot the ability to measure its surroundings using its 16 ultrasonic sensors (Figure 7). Do not bother about encoders yet. Code 2 shows the functions used to read proximity sensors and how to compute the distance to the detected object. Take into account that the measurement will only be valid if the obstacle is within the sensor's range (1 m in this case).

```
# First call (in _init_sensors)
sim.simxReadProximitySensor(client_id, sensor_handle,
    sim.simx_opmode_streaming)

# Subsequent calls (in the sense method)
_, is_valid, detected_point, _, _ = sim.simxReadProximitySensor(client_id,
    sensor_handle, sim.simx_opmode_buffer)

# Compute the Euclidean distance to the obstacle
distance = np.linalg.norm(detected_point)
```

Code 2. Remote API function to read proximity sensors. Do not forget to call `simxGetObjectHandle` for every sensor you want to monitor and to `import numpy as np` to compute the distance with the the last function. Non-required return arguments in Python are named with an underscore as shown in the second call to `sim.simxReadProximitySensor`.

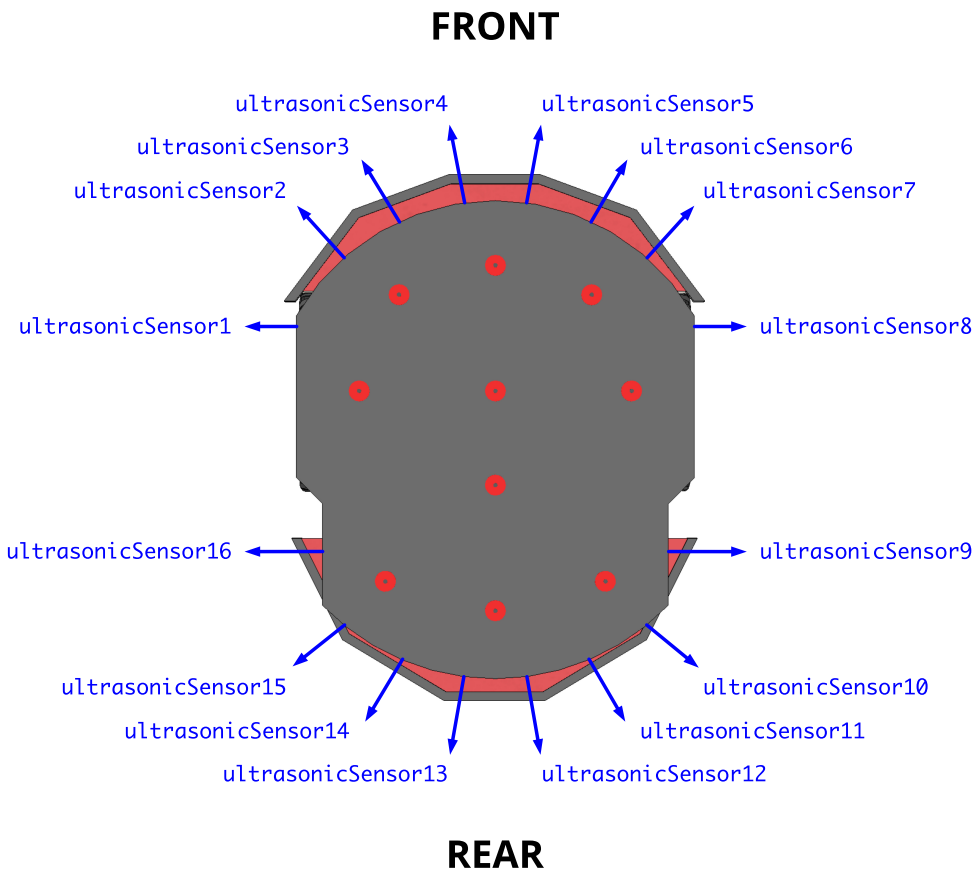


Figure 7. Ultrasonic sensor names and locations in a Pioneer P3-DX robot.

4. Retracing your steps (1 point)

Other interesting measurements you can obtain from the robot are the average of the actual linear and angular velocities it applied during the last time step. Ideally, these should match the values provided to the `move` method, but since motors cannot instantly change speed, and considering that wheels may slip, this is rarely the case. Unfortunately, the Pioneer P3-DX is not equipped with an encoder that returns the rotation of each drive wheel. However, we have programmed a Lua script in CoppeliaSim that emulates its behavior. As you will see later in the course, it is really useful to have a precise estimate of these variables and, actually, most commercial mobile robots have encoders.

Your task is to complete the `_sense_encoders` method —notice that `_init_encoders` has already been filled in for you—, which should solve forward kinematics from wheel rotations. Since this is an emulated sensor, the function you need to use to read the displacement in radians is different from the previous cases (see Code 3).

```
# Read encoder rotation
_, value = sim.simxGetFloatSignal(client_id, sensor_name,
    sim.simx_opmode_buffer)
```

Code 3. Remote API function to read a floating point value. For the encoders, the `sensor_name` should be either `leftEncoder` or `rightEncoder`, as shown in `_init_encoders`.

5. Navigating corridors (4 points)

After this never-ending tutorial, let's put your minds to better use. The main exercise of this lab session is to make your robot follow a wall. This is a simple yet useful exploration method that will allow the robot to move around in an unknown environment without colliding with surrounding objects. In the following lab session, you will learn how to localize the robot as it moves within a given map of the environment.

The `navigation.py` module contains an `explore` method where you can write your code. Feel free to add as many additional attributes and methods to the class as you need. In order to make things a bit simpler, you will be testing your algorithm in incrementally more difficult environments as explained in the subsections below. Your goal is to develop a single wall following algorithm that works for all three cases. Since there are many approaches to solve this problem, you might want to consider some of the following ideas. Notwithstanding, there exist many other perfectly valid alternatives.

- Program a state machine (a purely reactive control) that makes the robot go forward or turn —probably on the spot— depending on the front and side sensor measurements. In this case, it is essential to correctly adjust the transition thresholds or you will end up bumping into the walls.
- Combine the state machine with one or several controls.
  - You can try to implement a center line follower that keeps a constant distance with both walls. A PD control to issue steering commands (modify ω) might be enough, but if you decide to go for a PID, be careful with the integral term, since it can make your control unstable really quickly.
  - To follow a single wall, you might want to consider an angle and distance control. A double proportional control should work reasonably well.

5.1. Following a simple corridor (1.5 points)

In CoppeliaSim, go to File > Open scene... and load `lab01.ttt`, which is a maze environment with two separate circuits as shown in Figure 8. Starting at coordinates (4.0, –4.0) facing north (upwards) you first need to make the robot follow the outer corridor until it reaches the opposite end (Figure 8(a)).

5.2. Learning to perform U-turns (1 point)

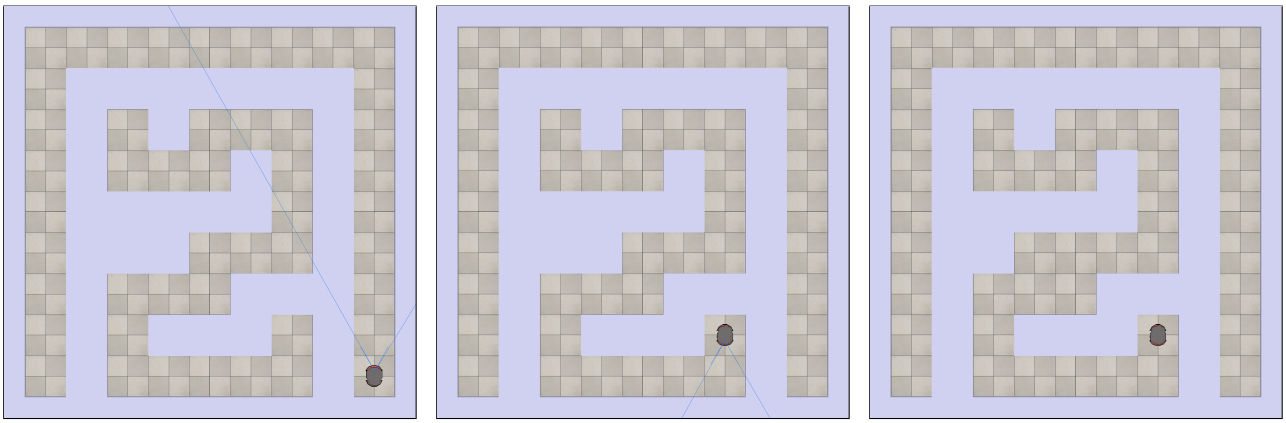
Instead of getting stuck at dead ends, modify your algorithm to make the robot turn 180º and start following the corridor in reverse direction.

5.3. Dealing with a more involved maze (1.5 points)

Finally, place the robot at coordinates (2.0, –3.0) heading south (downwards), as shown in Figure 8(b), and adjust your algorithm to explore this more complex environment back and forth.

6. Exploring the wild (1 point)

As you already know, the ultimate goal of these lab sessions is to help you get started with your final project. Load `lab01_challenge.ttt` and adapt your algorithm to handle open areas. As depicted in Figure 8(c), the cuboid at (–1.0, –1.0) has been removed. Consequently, in that region some of the robot's side sensor measurements will be out of range. Your task is to make sure that, in this situation, the robot continues following the only side wall it can see.



(a) **(b)** **(c)**

Figure 8. Starting robot pose for the simple (a) and complex (b) corridor environments. (c) is slightly more challenging since it has some open space.