CYBER-PHYSICAL SYSTEMS AND ROBOTICS

Lab 1. Perception and locomotion

# Preparation

1. The pass keyword is used when there is a need to write a statement but as of now, nothing has been implemented yet. Nothing will happen when executing this line of code. The idea is to replace it afterward.
2. The Idle class allows to execute the loop in main.py with a fixed time step. This way one can control the update rate of the measurements as well as the commands given to the robot. This limits the data transfer to an acceptable value which is a trade-off between resource efficiency and precision of the robot control.
3. It is impossible to instantiate the Robot class. Indeed, it is an abstract class, implemented thanks to the ABC class which is part of the abc package. An abstract class must be complemented to be instantiated. The Robot class has abstract methods, like move and sense, which must be implemented by the class RobotP3DX, inheriting from Robot. Then, it is possible to create an instance of RobotP3DX.
4. In the \_\_init\_\_ method of class RobotP3DX, there is a call to the \_\_init\_\_ method of the mother class Robot, which initialize attributes defined in Robot. RobotP3DX can access them as it inherits from this class.

# Code

## Wandering around

First, we modify the \_init\_motor method, with calls to the V-REP API to recover the handles (reference code to an object) of the two motors given their names. Finally, we return them in the form of a dictionary with keys “right” and “left”. The names could be retrieved in the simulator V-REP.

rc, handle\_right = vrep.simxGetObjectHandle(self.\_client\_id, "Pioneer\_p3dx\_rightMotor",  
 vrep . simx\_opmode\_blocking)  
rc, handle\_left = vrep.simxGetObjectHandle(self.\_client\_id, "Pioneer\_p3dx\_leftMotor", vrep.simx\_opmode\_blocking)  
  
motors = {"left": handle\_left, "right": handle\_right}

Code 1: Recovering the motors handles

Once we have the handle for each motor, the speed and direction of the robot can be controlled applying the desired velocity to each wheel. This happens in the move method as shown below. It is worth noting that the simxPauseCommunication API function is used so that both wheels receive the velocities at the same time.

rc = vrep.simxPauseCommunication(self.\_client\_id, True)  
rc = vrep.simxSetJointTargetVelocity(self.\_client\_id, self.\_motors['right'], (v + self.TRACK/2 \* w) / self.WHEEL\_RADIUS, vrep.simx\_opmode\_oneshot)  
rc = vrep.simxSetJointTargetVelocity(self.\_client\_id, self.\_motors['left'], (v - self.TRACK/2 \* w) / self.WHEEL\_RADIUS, vrep.simx\_opmode\_oneshot)  
rc = vrep.simxPauseCommunication(self.\_client\_id, False)

Code 2: Setting the speed

As seen in the code, differential inverse kinematics must be applied since we know the desired velocity for the object and we need to determine what the speeds for the actuators (motors) should be to reach it.

Where is the angular velocity of the wheel, the objective velocity of the robot, the distance between the wheels and the angular velocity of the robot.

## Taking a glance at the environment

Then, we had to do the same with the sensors. We could use the same API function but as there are 16 sensors, we used a loop to scan them all. Names were generated using a simple concatenation and handles were appended to a list previously created.

Sensors are also read a first time in the \_init\_sensors method in order to ensure there work correctly. In a first approach, the return code rc is read and a very simple verification is made, to ensure there is no error.

handle\_sensor = []  
for i in range (1, 17):  
 rc, handle = vrep.simxGetObjectHandle(self.\_client\_id, "Pioneer\_p3dx\_ultrasonicSensor" + str(i),  
 vrep.simx\_opmode\_blocking)  
 handle\_sensor.append(handle)  
 if rc != vrep.simx\_return\_ok:  
 print("error")  
 vrep.simxReadProximitySensor(self.\_client\_id, handle, vrep.simx\_opmode\_streaming)  
  
return handle\_sensor

Code 3: Recovering the sensors handles and storing them in a list

To read the sensors, the method sense was implemented, using the API function vrep.simxReadProximitySensor with a loop. The distance are computed thanks to the norm function of the numpy package.

distance = []  
for i in range(0, 16):  
 rc, is\_valid, detected\_point, \_, \_ = vrep.simxReadProximitySensor(self.\_client\_id, self.\_sensors[i],  
 vrep.simx\_opmode\_buffer)  
  
 if is\_valid:  
 distance.append(np.linalg.norm(detected\_point))  
  
 else:  
 distance.append(1)  
  
# print(distance)  
return distance

Code 4: Reading the sensors

When the sensors read something, the is\_valid variable is se to True and the distance to the wall is calculated. When there is no reading, it means that the wall is too far for the sensors to detect it, and we must set a value for the distance. A value is needed since it will later be used in the control loop in order to calculate the error, and in our case we set the distance to 1 as it gave us the best results.

## Exploring the wild

The first strategy used was to implement a simple 2-state machine: either the robot is moving on a straight line of the maize or it is turning, making a curve. Tests were performed to distinguish these cases depending on the measurements of the sensors.

if measurements[4-1] > 0.4 and measurements[5-1] > 0.4 and measurements[3-1] > 0.3 and measurements[6-1] > 0.3:

Code 5: Test to determine the state of the robot

In Figure 1 a representation of the robot and the location of the sensors is displayed. The first steps focused mainly on detecting the sensibility of said sensors in other to precisely decide when to turn and when to keep going forward.

1

8

16

9

Figure 1: Sensor distribution

The code and description of the first challenges for this lab will not be explained since they are included in the analysis of the final model, exposed in the following section.

## Road to the final project

The final code will now be explained, since it works in all cases proposed in the lab. The main is shown in the following code snippet.

error\_acumulation = CircularBuffer(4)

for error in error\_acumulation:  
 error\_acumulation.record(0)  
  
try:  
 while True:  
 # Write your control algorithm here  
 z = robot.sense()  
 v, w = navigation.explore(z, error\_acumulation)  
 robot.move(v, w)  
 loop\_time, overflow = idle.task()  
  
 if overflow:  
 print('Loop time: {0:.3f} s'.format(loop\_time))  
  
except KeyboardInterrupt: # Break the infinite loop to gracefully close the connection  
 pass

Code 6: Main loop

It starts by initializing the CircularBuffer class to 4, which is used to store the values of past errors (in this case, those in the last 4those in the last 4 sampling periods). Once this is done, we proceed with the main loop. The sensor readings are obtained with the robot sense method explained before, and said readings together with the error accumulation are the arguments that the “explore” method in the navigation class will use. This method is where the actual decision of where de robot should go is made.

if ((measurements[1-1] > 0.9 and measurements[2-1] > 0.9) and (measurements[7-1] < 0.8 and measurements[8-1] < 0.8) and measurements[4-1] > 0.9 and measurements[5-1] > 0.9):  
 error = 2 \* ((measurements[6-1]) + (measurements[7-1]) + (measurements[8-1]) - 0.45 - 0.45 - 0.5)  
 print("Lado izquierdo vacío")  
elif ((measurements[7 - 1] > 0.9 and measurements[8 - 1] > 0.9) and (measurements[1-1] < 0.8 and measurements[2-1] < 0.8) and measurements[4-1] > 0.9 and measurements[5-1] > 0.9):  
 error = -2 \* ((measurements[1 - 1]) + (measurements[2 - 1]) + (measurements[3 - 1]) - 0.45 - 0.45 - 0.5)  
 print("Lado derecho vacío")  
else:  
 error = - 1 \* (measurements[1-1]) - (measurements[2-1]) - (measurements[3-1]) + (measurements[6-1]) + (measurements[7-1]) + (measurements[8-1]) #La mayor parte del tiempo  
  
error\_acumulation.record(error) #Error acumulation es un buffer circular

Code 7: Wall detection

The snippet labeled as code 7 is where the robot detects if there is a wall and which one it is. The error reading will therefore depend on what wall it is and its proximity and is used next in order to set the robots velocity.

At first, a derivative control was tried, however it was discarded due to some problems when corners were reached. This is why we settled on a control depending on the integral as shown in code snippet 8.

integ = sum(error\_acumulation.get\_all()) \* 0.05  
  
k = 0.3  
  
if np.abs(integ) < 0.2:  
 v = k  
 w = - 0.7 \* integ  
else:  
 v = 0.1 \* integ  
 w = - 1.6 \* integ  
  
if ((measurements[4-1] - 1) + (measurements[5-1] - 1)) < -1.4:  
 v = 0  
 w = -1

Code 8: Velocity control

As we can see there are basically three cases:

1. If the accumulated error is very small we basically keep going forward
2. If it is big then we turn on the direction where there is no wall since it mean we have reached a corner.
3. The third case is when a dead end is reached and we must perform a 180º turn so we can keep going.

We end the loop calling the idle method so that each iteration lasts a previously fixed amount of time (the sampling period, in this case 0.05s).

It is worth noting that this script works for all use cases proposed in the lab, including the challenge with a missing square in the circuit. The possible improvements would probably be in the direction of further tuning of parameters since, although the reliability of our robot is close to 100%, its speed in completing the task still has some room for improvement.