Wall Following Pioneer 3DX Mobile Robot Technical Report

Author, *Ibrahim Ahmethan P2693799*, Supervisor, *Dr. Chigozirim Justice Uzor.* De Montfort University The Gateway House LE1 9BH

I. INTRODUCTION

THIS project can be considered as one of the early stages of the evolution of intelligent mobile robots, mainly automating robots to perform a specific task is getting more common in the industry and social life, starting from smart robots in Amazon's wearhouse to basic home appliances such as vacuum robots and so many other applications of robotics are taking place to increase human productivity, in this project the 3DX Pioneer Mobile Robot will be our guest (test pad) in the Coppelia simulator environment, there are many objectives for this task, it starts with establishing a client-server application interface API connection between Python programming language and Coppelia simulator interface, making the robot wander randomly in given environment until a steady state condition satisfied, dealing with sensor measurements by processing them in order to detect the walls and determine position of the Pioneer robot, moving and controlling the robot's actuators (DC left/right motors) and path following by avoiding obstacles (walls) in the scene, lastly the final aim that will be achieved to perform the wall following task after testing and improving the performance of the robot.
More technical, algorithmic and implementation methods and details will be explained in the following section of this report.
In summary, this project aims to program the Pioneer mobile robot to achieve the given task, which is wall following, of course, to be able to make the robot wander and then follow the wall all the mentioned objectives must be functioning correctly with minimum errors and high standards.

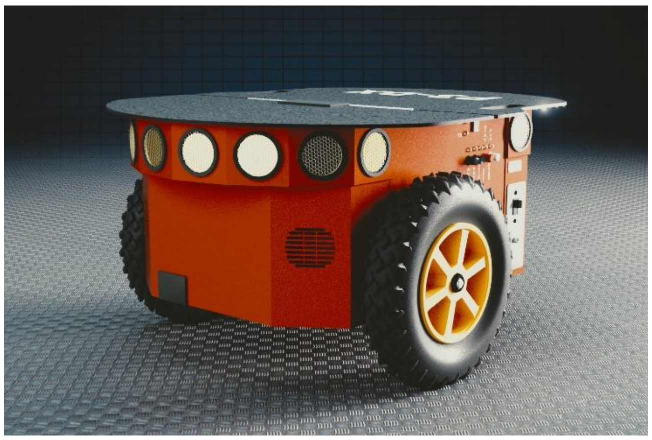


Figure 3. Pioneer 3DX Mobile Robot

II. HARDWARE

In this section, some basic information about the hardware of the Pioneer 3DX robot is represented.

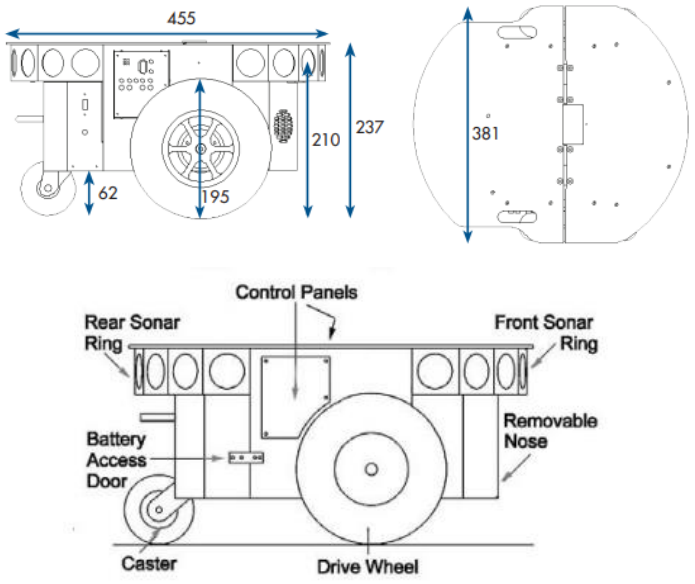


Figure 1. Dimensions in (mm) and Sensors positions of Pioneer 3DX Robot

Alongside the robot's characteristics, the main interesting part regarding this task is the ultrasonic sensors and the right/left motors.

The robot was designed with a total of 16 ultrasonic sensors, 8 sensors are positioned on the front side (front ring) and the other 8 sensors are positioned at the rear side of the robot (rear ring), the purpose of these ultrasonic sensors is to scan the environment with 360° degrees, in another word the sensors completing a whole circle with 180° front and 180° back sensor readings.

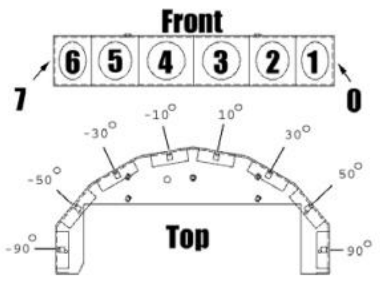


Figure 2. Pioneer 3DX Mobile Robot Front Sensors and their angle difference.

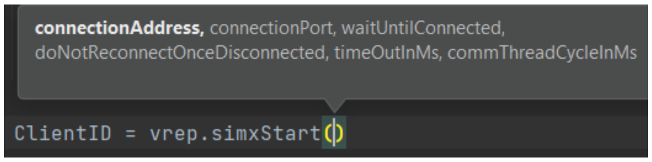
III. SOFTWARE

API was utilized in this task to establish a connection between the PC and the robot as the following:

1- PyCharm which is a Python-integrated development environment **(IDE) considered a client.**
2- CoppeliaSim is a simulator platform that offers an environment to test and run the pioneer robot where CoppeliaSim is **Considered a server**

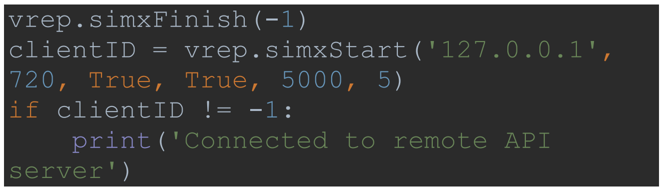
The CoppeliaSim simulation environment operates as a server, while the PC with its software (PyCharm) takes the role of client software that operates on a PC and is connected to the robot in the simulator via creating a communication thread with the CoppeliaSim (the server)
To establish a connection between CoppeliaSim and PyCharm a remote API connection is implemented by using the SimxStart function provided in the sim library.
The SimxStart Python synopsis for remote API connection is implemented as the following:

Algorithm: Python synopsis and Starting API connection



```
connectionAddress, connectionPort, waitUntilConnected,
doNotReconnectOnceDisconnected, timeOutInMs, commThreadCycleInMs

ClientID = vrep.simxStart()
```



```
vrep.simxFinish(-1)
clientID = vrep.simxStart('127.0.0.1',
720, True, True, 5000, 5)
if clientID != -1:
    print('Connected to remote API
server')
```

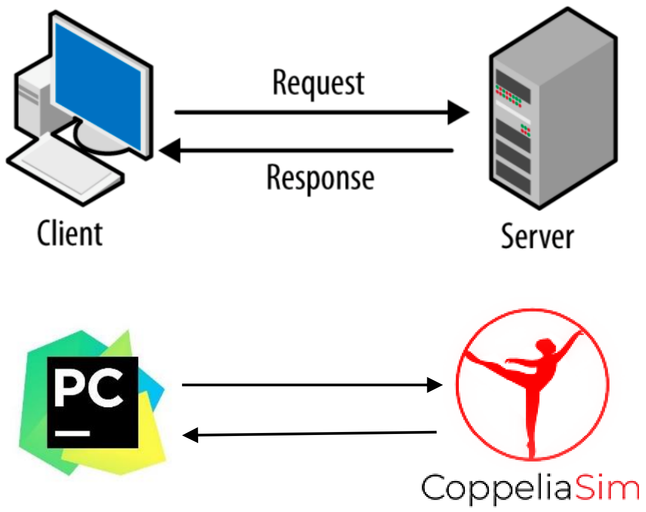
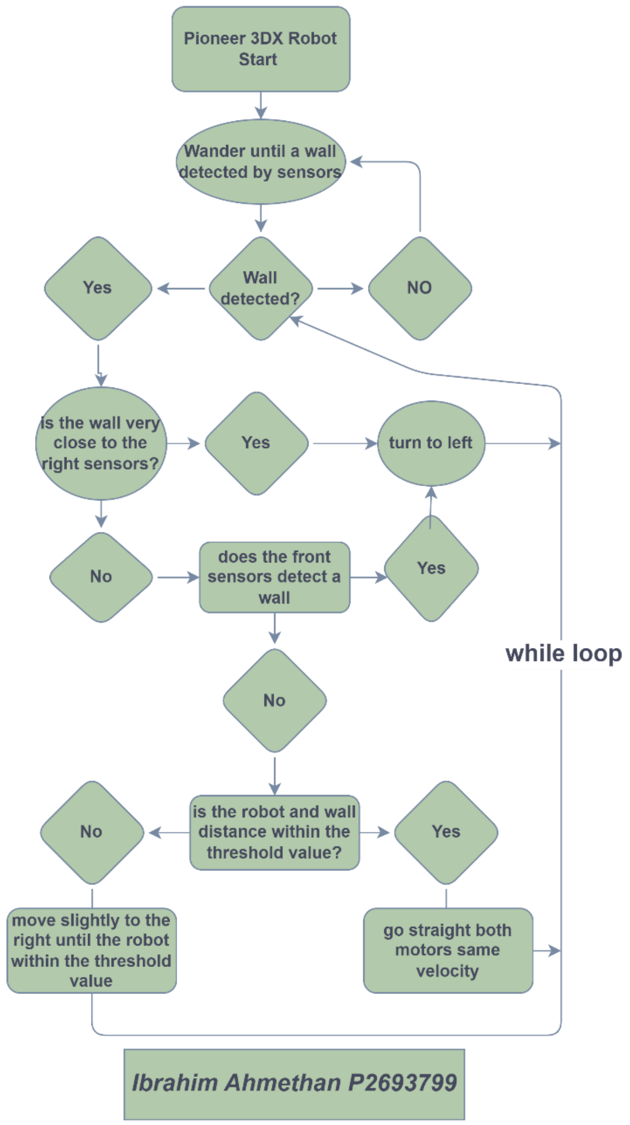


Figure 4. Client-Server Diagram of PyCharm and CoppeliaSim

IV. ALGORITHMS AND IMPLEMENTATION

A. Algorithms

When it comes to algorithms, the robot's main aim is to follow the wall, meanwhile, many algorithms will be functioning to achieve a different task, and threshold values will be set with default values to determine and maintain a reasonable distance between the robot and wall, as shown in figure 5 with three various regions, too far, sweet spot and too close, thus the aim is to maintain the path of the robot within the sweet spot by avoiding the too far and too close regions, accordingly, the wall following algorithms will be working as a correction function with a constant feedback mechanism.
The following flowchart explains the working principles for this task, starting with the wandering statement until the steady state condition is satisfied (until the wall is detected).
Please note that this algorithm is designed to follow the wall in only one direction (counterclockwise).


Pioneer 3DX Robot Start
Wander until a wall detected by sensors
Yes
Wall detected?
NO
is the wall very close to the right sensors?
Yes
turn to left
No
does the front sensors detect a wall
Yes
while loop
No
No
is the robot and wall distance within the threshold value?
Yes
move slightly to the right until the robot within the threshold value
go straight both motors same velocity
Ibrahim Ahmethan P2693799

Flowchart of the wall following robot

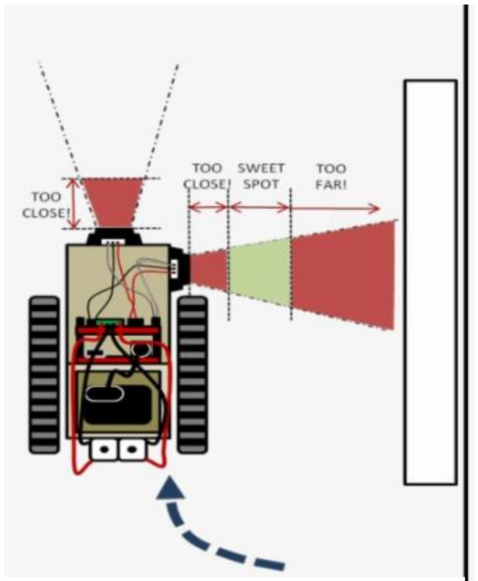


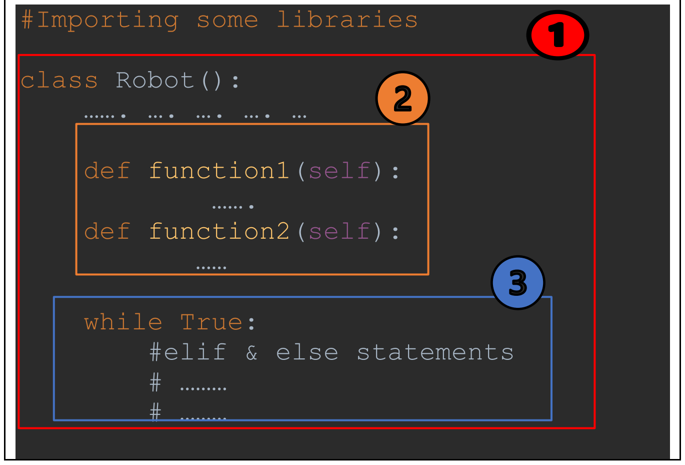
Figure 5. three different regions regarding the position of the robot

B. *Main Structure of Implementation*

Python programming language and Sim library are utilized to achieve the objectives and the aim of wall following robot, the structure of the coding section in Python has three main chunks, the purpose of chunking is to break down the problem into smaller parts, considered as a top-down approach, it helps us to optimize the code and algorithm effectively, the chunks are as the following:

1- Robot class
2- Many functions (indented under the Robot class)
3- While loop (indented under the Robot class)

Algorithm: Summary of Data Structure of The Task



```
#Importing some libraries

class Robot():
    ……. …. …. …. …

    def function1(self):
        …….
    def function2(self):
        ……

    while True:
        #elif & else statements
        # ………
        # ………
```

C. *Object Handling and reading*

Before start moving the robot, we need to handle (activate) The sensors and the right/left motors in the Pioneer 3DX. The following code is utilized to handle objects.

Algorithm: Handling and Reading Sensors

```
s_values = []

for i in range(1, 17):
    ## first chunk

    error, self.i =
    vrep.simxGetObjectHandle
   (clientID,
   'Pioneer_p3dx_ultrasonicSensor' +
    str(i), vrep.simx_opmode_blocking)

    ## second chunk
    error =
  vrep.simxReadProximitySensor(clientID,
   self.i, vrep.simx_opmode_streaming)
        s_values.append(self.i)
```

Since there are a total of 16 sensors on the robot, all sensors are handled by the **for loop,** by iterating the sensors starting from sensor 1 up to sensor 16, meanwhile, the second chunk under the for loop is deployed to register the sensor's readings and append them to the **s_values** empty list to retrieve the sensors readings later, thus in summary simxGetObjectHandle and the simxReadProximitySensor function are utilized to handle and read the sensor values.

Algorithm: Methods Used in Robot Class

```
def move(self):
     ## function used
simxSetJointTargetVelocity(self.R_motor)
    ……. … …..
def turn (self, turn_right, turn_left):
     ## function used
    simxSetJointTargetVelocity (…)
    …. …. …. ….
def wandering(self):
    ….. ….. ….
    simxSetJointTargetVelocity (…)

def distance (self, sensor):
    …. …. …… … …
```

Starting with the def move () method, which is the first and basic task, to move the robot an attribute created before under the robot class then called in the def move (method), more precisely the **self.R_motor** and **self..L_motor** attributes are called within the simxSetJointTargetVelocity and the velocity of both motors are set to be 1.5 initially.
The next def **turn ()** method was responsible to apply torque to the right and left wheels to change the heading of the robot, for example when the right wheel has a velocity greater than the velocity of the left wheel, the robot will start to move in a

trajectory biased to the left, in another word, the robot will start turning to the left and vice versa.
The two methods of **def wandering () and def distance ()** are responsible to allow the robot to wander and retrieve the detected distance of the wall respectively.
Wandering is done by importing the random library, then assigning random x, and y for the right and left wheels respectively with the range between 1-6 so the robot for each iteration will have random velocity for the left and right motors.

Algorithm: Function Used in Robot Class for Wandering

```
def wandering(self):

    x = random.randint(0, 12)
    y = random.randint(0, 12)

    vrep.simxSetJointTargetVelocity
   (clientID, self.R_motor, x,
  vrep.simx_opmode_blocking)

    vrep.simxSetJointTargetVelocity
    (clientID, self.L_motor, y,
    vrep.simx_opmode_blocking)
```

Lastly, the **def distance(sensor)** function is utilized to acquire the sensors readings from the **s_values** list to measure the distance between the robot and the walls.

Algorithm: Method Used in Robot Class to measure the distance

```
def distance (self, sensor):
    returnCode, detectionState,
  detectedPoint, detectedObjectHandle,
  detectedSurfaceNormalVector =
  vrep.simxReadProximitySensor(
      clientID, sensor,
  vrep.simx_opmode_buffer)
    x = np.array(detectedPoint)
    if detectionState == True:
        return np.linalg.norm(x)
```

Many functions/methods are deployed under the Robot class, each function has its duty, some of them are called during the execution of the while loop as in the case of **def distance ()** function is under the while loop, more precisely in the else/elif statements to measure all the sensors reading and checking the obtained distance values with the default threshold values.
The while loop will be running all the time as long as the robot detects a wall if in any case, the robot stops detecting a wall, the execution of the while loop will stop immediately and return to the wandering state, which means the algorithms will start running from the initial point.

Algorithm: def turn () and def distance () implementation

```
while True:

    if robot.distance(s_values[0]) < 1:
        robot.turn(2.5, -2.5)

    elif robot.distance(s_values[1]) <1:
        robot.turn(2.5, -2.5)

    elif robot.distance(s_values[2]) <1:
        robot.turn(2.5, -2.5)
```

The most challenging part of this task was to maintain the robot's path while maneuvering for curves and U-turns, after many times of trial and error finally the desired sensors and appropriate threshold sensors value was determined and then assigned specifically for each condition, thus the robot was able to make a smooth U-turn then continue following the wall.

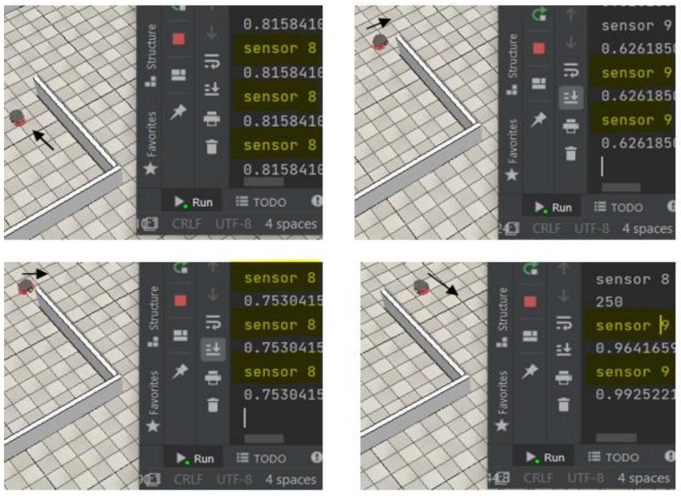

```
0.8158416
sensor 8
0.8158416
sensor 8
0.8158416
sensor 8
0.8158416
```
```
sensor 9
0.626185
sensor 9
0.626185
sensor 9
0.626185
```
```
sensor 8
0.7530415
sensor 8
0.7530415
sensor 8
0.7530415
```
```
sensor 8
250
sensor 9
0.9641659
sensor 9
0.9925221
```

Figure 6. U turn maneuver of the robot

Algorithm: Sensors 8 and 9 with threshold values during the U maneuver of the robot

```
elif robot.distance(s_values[7]) < 1:
    robot.turn(0.2, 0.6)
    print("sensor 8")
    print(robot.distance(s_values[7]))
elif robot.distance(s_values[8]) < 2:
    robot.turn(0.2, 1.5)
    print("sensor 9")
    print(robot.distance(s_values[8]))
```

As shown in the algorithm above, while the sensors reading 8 and 9 below the threshold value, the left motors will have 0.6, and 1.5 velocities respectively, as a result of this angular velocity the robot will start to make a U-turn shape in a clockwise path.

V CONCLUSION

In conclusion, the purpose of this task was to find the wall in the given sense and then follow the wall including curves smoothly by maintaining a standard distance between the wall and the robot (threshold value), the challenging part of wall following was to follow the wall by figuring out appropriate threshold and check it against the obtained sensors values to apply appropriate velocities for both the right and left wheels and smoothly follow the curves without any deviation, the other challenging part was to develop specific conditions for the robot to perform the U-turn maneuver and continue to follow the wall as long as there are no obstacles in front of the robot, in this part, the sequence conditions of elif/else statements in while loop, dealing with the sensors, allocating balanced velocities for the right and left wheels to be applied under the condition of U-turn and assigning threshold value was taking into consideration to overcome the U-turn malfunction of the Mr. 3DX Pioneer.
Possible improvement and enhancing the robot's performance could be done by implementing fewer sensors, hence the computational cost will be less as a result of decreasing the number of sensors, in my case, I attempted to use fewer sensors, however, for some vague reasons that caused another issue, especially during the U-turn maneuvering, so I decided to prefer the usage of all sensors over the computational cost, but I believe that this could be done with fewer sensors, another future improvement is to introduce a function that allows the robot follow the wall in both clockwise and counterclockwise by taking into consideration the position or the heading(angle) of the robot.
the objectives of this task were successfully performed and achieved the main aim which is wall following robot. The design of this task was implemented by breaking down the task into smaller tasks and allocating each small task to perform its purpose such as wandering and measuring the distance, furthermore, the second stage of the task was to test the algorithms and implementation by correcting errors and improving the robot's performance in the simulator environment, finally after all tests are done successfully, the robot succeeded to follow the wall in the given scene.

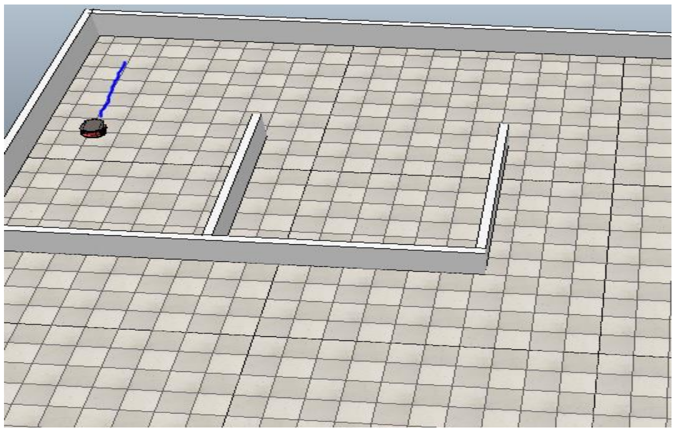


Figure 7. Pioneer robot following the wall

REFERENCES

[1] Legacy remote API functions (Python) https://www.coppeliarobotics.com/helpFiles/en/remoteApiFunctionsPython.htm
[2] https://www.cyberbotics.com/doc/guide/pioneer-3dx?version=cyberbotics:R2019a#pioneer-3-dx-model
[3] https://www.youtube.com/watch?v=iD2Dc6r7PeQ
[4] https://www.coppeliarobotics.com/videos
[5] https://www.coppeliarobotics.com/helpFiles/
[6] https://www.mdpi.com/2227-7390/8/8/1254/htm
[7] Learning materials Week 3, Week 4 and Week 5
[8] https://www.generationrobots.com/media/Pioneer3DX-P3DX-RevA
[9] https://www.youtube.com/watch?v=SQont-mTnfM&t=1381s
[10] https://github.com/nikolai-kummer?tab=overview&from=2017-12-01&to=2017-12-31
[11] https://www.youtube.com/watch?v=xI-ZEewIzzI&t=8s
[12] https://www.youtube.com/watch?v=DkOXpZtEZwg
[13] https://www.youtube.com/watch?v=heNYjDOVcKY
[14] https://www.youtube.com/watch?v=iD2Dc6r7PeQ&t=7s

APPENDIX

```
import numpy as np
import random
import sim as vp

print("Hello Dr.Uzor!")
s_values = [] ## list values for sensor readgins
class Robot():
    def __init__(self):
        for i in range(1, 17): ## looping to handle and append sensors values to
s_list
            error, self.i = vp.simxGetObjectHandle(clientID,
'Pioneer_p3dx_ultrasonicSensor' + str(i),
                                                   vp.simx_opmode_blocking)
            error = vp.simxReadProximitySensor(clientID, self.i,
vp.simx_opmode_streaming)
            s_values.append(self.i)

        code, self.dummy = vp.simxGetObjectHandle(clientID, "Pos",
vp.simx_opmode_blocking)

        # Create the objects to call the motors
        code, self.R_motor = vp.simxGetObjectHandle(clientID,
"Pioneer_p3dx_rightMotor",
                                                    vp.simx_opmode_blocking)
        code, self.L_motor = vp.simxGetObjectHandle(clientID,
"Pioneer_p3dx_leftMotor",
                                                    vp.simx_opmode_blocking)
    def move(self):
        code = vp.simxSetJointTargetVelocity(clientID, self.R_motor, 1.5,
vp.simx_opmode_blocking)
        code = vp.simxSetJointTargetVelocity(clientID, self.L_motor, 1.5,
vp.simx_opmode_blocking)

    def turn(self, turn_right, turn_left):
        code = vp.simxSetJointTargetVelocity(clientID, self.R_motor, turn_right,
vp.simx_opmode_blocking)
        code = vp.simxSetJointTargetVelocity(clientID, self.L_motor, turn_left,
vp.simx_opmode_blocking)

    def position(self):  #  used to identify specific points or reference frames in
the scene
        code, position = vp.simxGetObjectPosition(clientID, self.dummy, -1,
vp.simx_opmode_blocking)

    def wandering(self):
        x = random.randint(0, 6)
        y = random.randint(0, 6)
        code = vp.simxSetJointTargetVelocity(clientID, self.R_motor, x,
vp.simx_opmode_blocking)
        code = vp.simxSetJointTargetVelocity(clientID, self.L_motor, y,
vp.simx_opmode_blocking)

    def distance(self, sensor):
        returnCode, detection, detection_point, detection_object_handle,
surface_normal  = vp.simxReadProximitySensor(
            clientID, sensor, vp.simx_opmode_buffer)
        x = np.array(detection_point) ## vector (x,y,z) normalization
        if detection == True:
            return np.linalg.norm(x)
        else:
            return 500

vp.simxFinish(-1)
clientID = vp.simxStart('127.0.0.1', 940, True, True, 5000, 5)
if clientID != -1:
    print('Connected to API Coppelia server')
    robot = Robot()
    while True:

        robot.position()
        robot.move()
        if robot.distance(s_values[0]) < 0.9:
            robot.turn(2.5, -2.5)
        elif robot.distance(s_values[1]) < 0.9:
            robot.turn(2.5, -2.5)
        elif robot.distance(s_values[2]) < 0.9:
            robot.turn(2.5, -2.5)
        elif robot.distance(s_values[3]) < 0.9:
            robot.turn(2.5, -2.5)
        elif robot.distance(s_values[4]) < 0.9:
            robot.turn(2.5, -2.5)
        elif robot.distance(s_values[5]) < 0.9:
            robot.turn(0.2, 0.4)
        elif robot.distance(s_values[6]) < 0.9:
            robot.turn(0.4, 0.4)
        elif robot.distance(s_values[7]) < 0.5:
            robot.turn(0.6, 0.2)
        elif robot.distance(s_values[7]) < 0.9:
            robot.turn(0.2, 0.6)
        elif robot.distance(s_values[8]) < 1.9:
            robot.turn(0.2, 1.5)
        elif robot.distance(s_values[9]) < 1.9:
            robot.turn(0.4, 1.5)
        elif robot.distance(s_values[11]) < 1.9:
            robot.turn(0.6, 2)
        elif robot.distance(s_values[12]) < 1.9:
            robot.turn(0.6, 2)
        elif robot.distance(s_values[13]) < 1.9:
            robot.turn(2, 0.4)
        elif robot.distance(s_values[14]) < 0.5:
            robot.turn(0.2, 0.6)
        elif robot.distance(s_values[14]) < 1.9:
            robot.turn(0.6, 0.2)
        elif robot.distance(s_values[15]) < 0.9:
            robot.turn(1, 0.4)
        else:
            robot.wandering()
else:
    print('Not able to connect to API!')
```