

# **Mobile Robots assessment 6**

Student: Ibrahim Ahmethan

P2699379

Supervisor: Dr.Chigozirim Justice Uzor

MSc Intelligent Systems and Robotics, De Montfort University, the Gateway, Leicester

## **Abstract**

Mobile robots are autonomous or semi-autonomous machines that are capable of moving around and completing tasks within a given environment. These robots can be used for a variety of purposes, including transportation, manufacturing, inspection, and search and rescue operations. Mobile robots can be classified based on their mobility mechanisms, such as wheeled, tracked, or legged, and their mode of operation, such as teleoperated or fully autonomous. Mobile robots can also be equipped with a variety of sensors and actuators, allowing them to navigate and interact with their environment. The use of mobile robots is increasing in a variety of industries, and they are expected to play an important role in the future of automation and robotics. In this assessment, Python programming language and CoppeliaSim are utilized to get readings from the sonar and vision sensors of the Pioneer 3dx robot to detect and move toward the beacon, the goal is to allow the Pioneer 3dx mobile robot to find the beacon and return back to the center, the objectives are as the following, finding the center of the square in the beginning, getting out of the square by avoiding obstacles, mapping the environment while wandering and lastly detecting then moving toward the beacon and return to the center of the square autonomously.

*Keywords* – Mapping, Beacon, Pioneer 3dx, Path planning, Python, CoppeliaSim

## **1. Introduction**

This project focus on automating the Pioneer mobile robot to achieve a sequence of given orders without active human control and instructions. The Pioneer 3dx mobile robot will be programmed to achieve a sequence of tasks and goals, in this project our Pioneer 3dx mobile robot will have the same working principles as a smart vacuum cleaner robot, mainly mapping, wandering and returning to a previously specified point on the environment safely by avoiding obstacles.

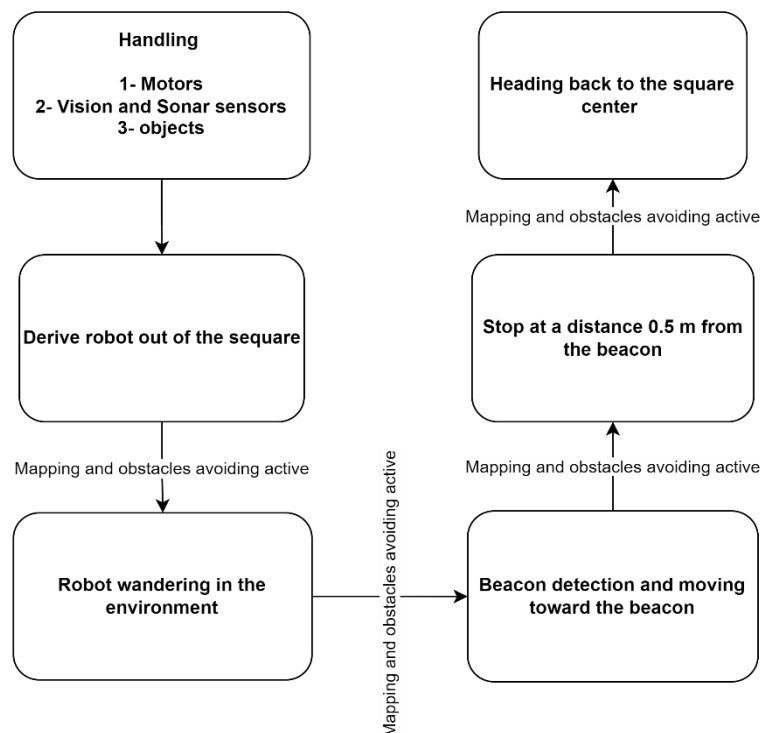
The simulation platform used in this task is CoppeliaSim which offers us a variety of supported API programming languages with their libraries, in addition, each API was designed for a specific programming language such as Python, C++, Lua and MATLAB, which helps robotic developers choose which programming language and tools to utilize. This flexibility by the CoppeliaSim APIs allows designing and implementing robotic projects and ideas in a very efficient way, however, any errors such as syntax, logic, runtime and arithmetic errors in the code during the execution will cause the robot to

miss or completely fail to achieve the given tasks, therefore, the software implementation is crucial and it must be free of any types of error, other ways the robot will fail to handle the given task.

In this task, Python programming language was used to implement the proposed design. To be able to control the Pioneer 3dx robot and achieve the given task, it is necessary to first know how to interact with the hardware and software aspects of the Pioneer robot, furthermore, the robot will depend on critical hardware elements such as motors to move the robot, proximity sensors to map the environment and avoid obstacles, lastly, vision sensors are utilized to detect and guide the robot toward the beacon. Of course, to carry out the objectives for this assessment, all the hardware and software elements must function together as a network in a proper and organized way. Thanks to CoppeliaSim, all the hardware properties of the Pioneer robot included alongside the Python API are fully supported, therefore, the interaction between the hardware and software elements was done through the CoppeliaSim platform by using Python programming language.

## 2. Design and Implementation

From the previous assessments, Lab 4 and Lab 5, the same Robot() Python class is used and extended, to cover a wider range of tasks and abilities to achieve the tasks. The following diagram displays the summary of the proposed design, the problem broken down into smaller chunks to make the task more manageable and programable, therefore, each box in the diagram is handled separately by a method or function indented under the class Robot() or in independent blocks of functions to achieve a specific task.



## 2.1. Mapping and obstacles avoidance

Pioneer robot additionally creates the room's map in addition to the other duties. Sonar sensors are utilized for this duty, the x and y coordinates of the obstacles discovered by Pioneer's sonar sensors, then the positions of obstacles are reflected back on the CoppeliaSim's graphical function, which draws the shape of the detected object on an additional tab. (Note: older versions of CoppeliaSim were used for mapping. Version 4.0.0 Edu).

One important aspect of mobile robot design is the ability to navigate and avoid obstacles in their environment. This is especially important in dynamic and complex environments where the robot must be able to adapt to changing conditions and avoid collisions with objects. In this task front and side sonar sensors are used for the purpose of mapping and obstacle avoidance, thus a method called `avoid_obstcls(self)`: is implemented to avoid obstacles during the execution of the program. The following figure 1 displays the avoidance algorithm used in this task.

as shown in figure 1, the while loop will execute the loop as long as an object is detected by the sonar sensors, furthermore, the reason behind activating the side sonar sensors is to calculate the angle of attack of the robot against an obstacle, for example, if the robot heading to an obstacle with an angle of 80 degrees, the robot will turn slightly to the right to avoid obstacle and correct its position. For the case of 90 degrees, the robot will have a 50/50 chance to turn either to the left or the right.

```
def avoid_obstcls(self):
    flag = True
    while flag:
        code, orientation = vp.simxGetObjectOrientation(clientID, se
        print(orientation[2])

        print("obstcls in exe")
        # robot.turn(2, 2)
        if robot.distance(s_values[3]) < 0.6: ## 212 front left
            robot.turn(-1, 1)
            return True
        elif robot.distance(s_values[4]) < 0.6: ## 216 front right
            robot.turn(1, -1)
            return True
        if robot.distance(s_values[2]) < 0.6: ## 212 front-left
            robot.turn(-1, 1)
            return True
        elif robot.distance(s_values[5]) < 0.6: ## 216 front-right
            robot.turn(1, -1)
            return True
        if robot.distance(s_values[1]) < 0.4: ## 212 left side
            robot.turn(-0.7, 0.7)
            return True
        elif robot.distance(s_values[6]) < 0.4: ## 216 right side
            robot.turn(0.7, -0.7)
            return True
        else:
```

Figure 1. obstacles avoidance algorithm

## 2.2. Driving the Pioneer out of the square.

Initially, the Pioneer robot will start at any random position inside the square. The task is to program the robot to be able to find its way out of the square, first of all, the robot must travel to the center of the square from any random position inside the square, secondly, the robot has to determine the location of the exit and move in a straight line toward the exit without hitting the walls. After having the robot out of the square, the robot will be in a wandering state until the beacon detection state.

To derive the robot out of the square a method called `def out_of_square(self):` under the `class Robot():` is used, the method contains three sub-functions as the following, `def check_center():`, `def ver():` and `def hor():`. All three mentioned functions are executed under the while loop as shown in figure 2.

```
flag = True
while flag:
    error, code = vp.simxGetObjectPosition(clientID, self.p3dx, -1, vp.simx_opmode_streaming)
    x = round(code[0], 2)
    if x < 1.78 and x > -2.40:
        print("yes")
        hor()
        ver()
        check_center()
    else:
        flag = False

print("Robot")
return "out of square"
```

Figure 2. `def out_of_square(self):` method execution

`hor()` function is responsible to obtain the sensor readings and adjust the wheels' speed to make the robot travel to the middle of the horizontal axis inside the square (concerning the scene), meanwhile, `ver()` function is in charge to obtain the sensor readings and adjust the wheels' speed to allow the robot to travel to the middle of the vertical axis inside the square. In conclusion, after finding the middle horizontal and vertical points inside the square, the actual x and y coordinates corresponding to the center of the square will be found.

the `check_center()` function is executed recursively under the while loop, each time the function is executed it checks the robot's live position x and y positions in the scene, the coordinates of the center of the square are already known, therefore once the robot arrives to a pre-defined x and y coordinates, the loop will stop executing, hence the robot will stop at the center.

hor() and ver() functions are executed recursively, each time when the while loop runs, one movement in the horizontal axis and then one movement in the vertical axis toward the center is carried out. As a result, at each execution of the while loop, the robot gets closer to the center point, this algorithm mimics a baby crawling step by step.

Different approaches were developed to find the center, however, some of them did not work correctly at some point, one of the approaches developed was to calculate the middle distance in x and y positions inside the square and adjust the wheels' speed accordingly, the problem with this approach was sometimes the robot got out of the square without hitting the center point, thus the novel design of baby crawling algorithm was implemented to make sure that the robot will not miss the center point.

Lastly, once the robot reaches the center point, the robot will spin around itself 360 degrees, while spinning the robot's front sonar sensors 4 and 5 will be scanning for any gap (the exit way), once a gap is detected the robot will stop spinning and applies equal speed to both wheels in order to exit the square.

## **2.3 Detecting the beacon in the environment**

The main goal of this task is to automate the robot to be able to find the beacon by detecting it and applying proper speed to the wheels to direct the robot toward the beacon, meanwhile avoiding the obstacles and mapping the environment are carried out by different functions.

A vision sensor attached to the Pioneer robot is used to get a live view of the scene, of course having a live stream of the scene is not enough to achieve the given task. Initially, two different approaches were considered to detect the beacon, the first approach is to consider using the OpenCV library to process the footage obtained from the vision sensor of the robot, and the second approach was to take advantage of the vrep library and some of its function provided by the CoppeliaSim Python API.

The second approach is used to enable the robot to detect the beacon, the beacon's color changed to red color to distinguish the beacon from the other object scene and give it a unique color spectrum. A method called `read_vioion(self):` is responsible to detect any red-colored object obtained from the vision sensor footage, the next step is to direct the Pioneer robot toward the red-colored object (the beacon). The following `read_vioion(self):` function shows how the red-color object is distinguished from the footage.

```
def read_vioion(self):

    code, detection_state, packets = vp.simxReadVisionSensor(clientID, self.camera_1, vp.simx_opmode_buffer)
    codd1, detection_state1, packets1 = vp.simxReadVisionSensor(clientID, self.camera_2, vp.simx_opmode_buffer)
    code2, detection_state2, packets2 = vp.simxReadVisionSensor(clientID, self.camera_3, vp.simx_opmode_buffer)

    return packets[0][6], packets1[0][6], packets2[0][6]
```

Figure 3. `read_vioion(self)`: method to distinguish red-color object

Thanks to the CoppeliaSim Python API functions, from the `vrep` class, a function called `vp.simxReadVisionSensor` is implemented to obtain the RGB color spectrum from the footage of the vision sensor, thus the R, G and B channels are obtained individually, the following figure 4 shows the details of the `simxReadVisionSensor` function.

#### simxReadVisionSensor

Description	Reads the state of a vision sensor. This function doesn't perform detection, it merely reads the result from a previous call to <code>sim.handleVisionSensor</code> ( <code>sim.handleVisionSensor</code> is called in the default main script). See also <code>simxGetVisionSensorImage</code> and <code>simxGetObjectGroupData</code> .
Python synopsis	<code>number returnCode, bool detectionState, array auxPackets=</code> <code>simxReadVisionSensor</code> <code>(number clientID, number sensorHandle, number operationMode)</code>
Python parameters	<b>clientID</b> : the client ID. refer to <code>simxStart</code> . <b>sensorHandle</b> : handle of the vision sensor <b>operationMode</b> : a <a href="#">remote API function operation mode</a> . Recommended operation modes for this function are <code>simx_opmode_streaming</code> (the first call) and <code>simx_opmode_buffer</code> (the following calls)
Python return values	<b>returnCode</b> : a <a href="#">remote API function return code</a> <b>detectionState</b> : the detection state (i.e. the trigger state). <b>auxPackets</b> : by default CoppeliaSim returns one packet of 15 auxiliary values: the minimum of {intensity, red, green, blue, depth value}, the maximum of {intensity, red, green, blue, depth value}, and the average of {intensity, red, green, blue, depth value}. Additional packets can be appended in the <a href="#">vision callback function</a> .
Other languages	C/C++, Java, Matlab, Octave

Figure 4. `vrep simxReadVisionSensor` function

As shown in the above figure 4, Python return values are as the following, `returnCode`, `detectionState` and `auxPackets`, the return values that we are interested in is the `auxPackets`. The `auxPackets` value itself returns 15 auxiliary values of minimum, maximum and average of intensity, RGB colors and depth values. It is worth mentioning that only the values of the R channel are used and processed.

To detect the red color in the scene, the maximum of red color values are obtained from the `auxPackets` return values, as shown in the `read_vioion()` method the return values are `packets[0][6]` which corresponds to the maximum red color in the `auxPackets` return values, this means any red color pixels appear in the footage will be detected by the `auxPackets` return values. In summary, by utilizing the `simxReadVisionSensor` any red pixels that appear in the footage of the vision sensor will be detected, hence the red color beacon will be detected since it appears as a red color in the footage.

After successful detection of red pixels appearing in the video footage, the robot must correct its position to head toward the beacon, for this purpose, three vision sensors are attached to the robot, covering approximately 180 degrees of the front view. The vision sensors are attached to the Pioneer robot as the following, front, right and left vision sensors, therefore, three different views are obtained from the robot, front, left and right views. The reason behind using three vision sensors is to be able to direct the robot toward the beacon.

A simple algorithm adapted to direct the robot, by using three different vision sensors, if the beacon

appears in the right vision sensor, it means that the beacon is on the right side of the robot, therefore, the robot will start to make a smooth right turn until the red beacon appears only in the front vision sensor.

Once the beacon appeared ONLY in the front vision sensor, the robot will start moving in a straight line and avoiding obstacles at the same time. If the red pixels appear in the left sensor, it means that the beacon is on the left side according to the robot's position, thus the robot will start correcting its heading and start turning to the left until the red pixels appear only in the front vision sensor.

In summary, the beacon will start appearing on different vision sensors during the journey, and depending on which vision sensor red pixels are detected, the proper adjustments of the right and left wheels will be applied.

The following diagram shows the beacon detection and direction algorithms implementation. Furthermore, a top view of the robot and footage views of the front, left and right sensors are displayed.

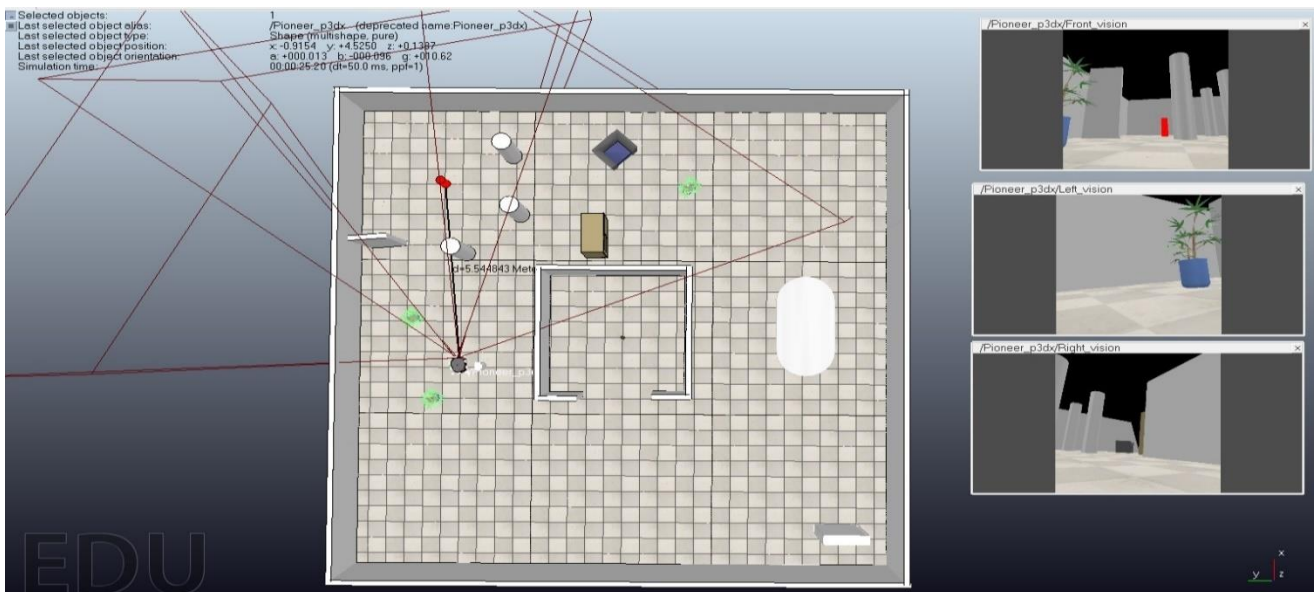
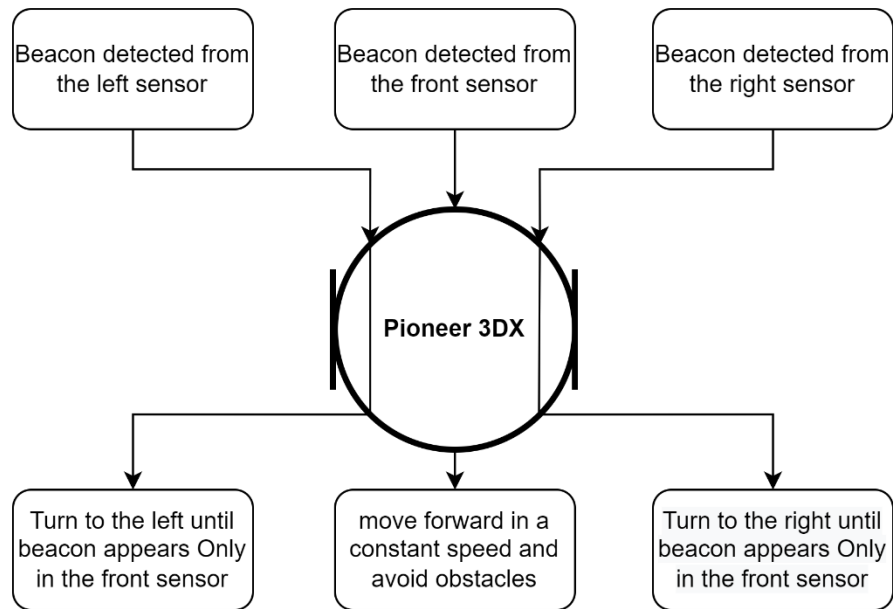
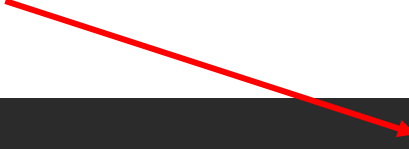


Figure 5. Top view of Pioneer robot

As shown in figure 5, three vision sensors are covering approximately 180 degrees of the front view, in figure 5, the beacon is detected from the front vision sensor, thus the robot moves a straight line toward the red beacon.

In the following figure, the software implementation of beacon detection and following is represented, the last stage of the beacon detection and following is to stop at a distance 0.50 m away from the beacon, to achieve this, a function in CopeilaSim API called `simxCheckDistance()` implemented to measure the distance between two entities (the Pioneer robot and the beacon).



```
while flagge:

    code, beacon_robot_dist = vp.simxCheckDistance(clientID, robot.p3dx, robot.beacon, vp.simx_opmode_buffer)
    # code, position_of_beacon = vp.simxGetObjectPosition(clientID, robot.beacon, -1, vp.simx_opmode_buffer)
    print(f" distance to robot {beacon_robot_dist}")
    robot.turn(2, 2)

    if beacon_robot_dist < 0.6:
        robot.turn(0, 0)
        robot.turn(0, 0)
        robot.turn(0, 0)
        robot.turn(0, 0)
        robot.turn(0, 0)
        robot.turn(0, 0)
        robot.turn(0, 0)
        robot.turn(0, 0)
        robot.turn(0, 0)
        robot.turn(0, 0)
```

Figure 6. `simxCheckDistance` function

```
if robot.read_vioion()[0] == 1 and robot.read_vioion()[1] == 1: ## beacon appeared in the left and front vision sensor
    robot.turn(2, -1) ## turn left
    robot.turn(2, 1) ## make a smooth turn to the left
    # robot.turn(2, -2)
    print("turn left")
if robot.read_vioion()[0] == 1 and robot.read_vioion()[2] == 1: ## beacon appeared in the right and front vision sensor
    robot.turn(-1, 2) ## turn right
    robot.turn(1, 2) ## make a smooth turn to the right
    # robot.turn(-2, 2)
    print("turn right")
if robot.read_vioion()[0] == 1 and robot.read_vioion()[1] != 1 and robot.read_vioion()[2] != 1: ## beacon ONLY appears
    robot.turn(2, 2)
    print("go ahead")

if robot.read_vioion()[1] == 1 and robot.read_vioion()[0] != 1: ## beacon appeared ONLY in the left vision sensor
    robot.turn(2, -1) ## turn left
    robot.turn(2, 1) ## make a smooth turn to the left
    # robot.turn(2, -2)
    # robot.turn(2, -2)
    print("turn left")
if robot.read_vioion()[2] == 1 and robot.read_vioion()[0] != 1: ## beacon appeared ONLY in the right vision sensor
    robot.turn(-1, 2) ## turn right
    robot.turn(1, 2) ## make a smooth turn to the left
    # robot.turn(-2, 2)
    # robot.turn(-2, 2)
    print("turn right")
```

Figure 7. Detection algorithms of Pioneer robot



## 2.3. Returning to the center

After successful detection of the beacon and stopping at a distance 0.50 m away from the beacon, it is time for the Pioneer robot to return home, this task is very similar to the smart vacuum cleaners. To derive the robot back autonomously to the center, three imaginary dashed lines were established as shown in figure 8, two lines along the x-axis and one line along the y-axis. To allow the robot to travel back to the center, a threshold value is set for each imaginary line according to their coordinates, for example, the longest imaginary line on the x-axis as shown, while the robot traveling back to the center, the x-axis value of the dashed imaginary line is  $x = -5$ , meanwhile the robot x and y position are calculated and updated continuously during the travel time, hence, when the robot's x position is bigger or smaller than imaginary  $x = -5$  it means that the robot is out of the track, thus, the robot will adjust its wheel speed to maintain its x position at a position equal to the imaginary x value line. The imaginary lines have a fixed horizontal and vertical x and y-axis.

The same logic is applied to the y-axis, once the robot reaches the end of the first x-axis imaginary line, it is designed to make a 90 degrees left turn, then continue along the imaginary y-axis by maintaining the threshold value of the y imaginary line.

At the end of the journey, once the robot reaches the center point inside the square, the x and y coordinates of the center point are known already, therefore, when the robot reaches the final destination, the robot's x and y coordinates and the predefined x and y coordinates of the center will be satisfied, thus the robot stops at the center. As a result, the goal of this assessment will be completed by satisfying all the objectives and goals.

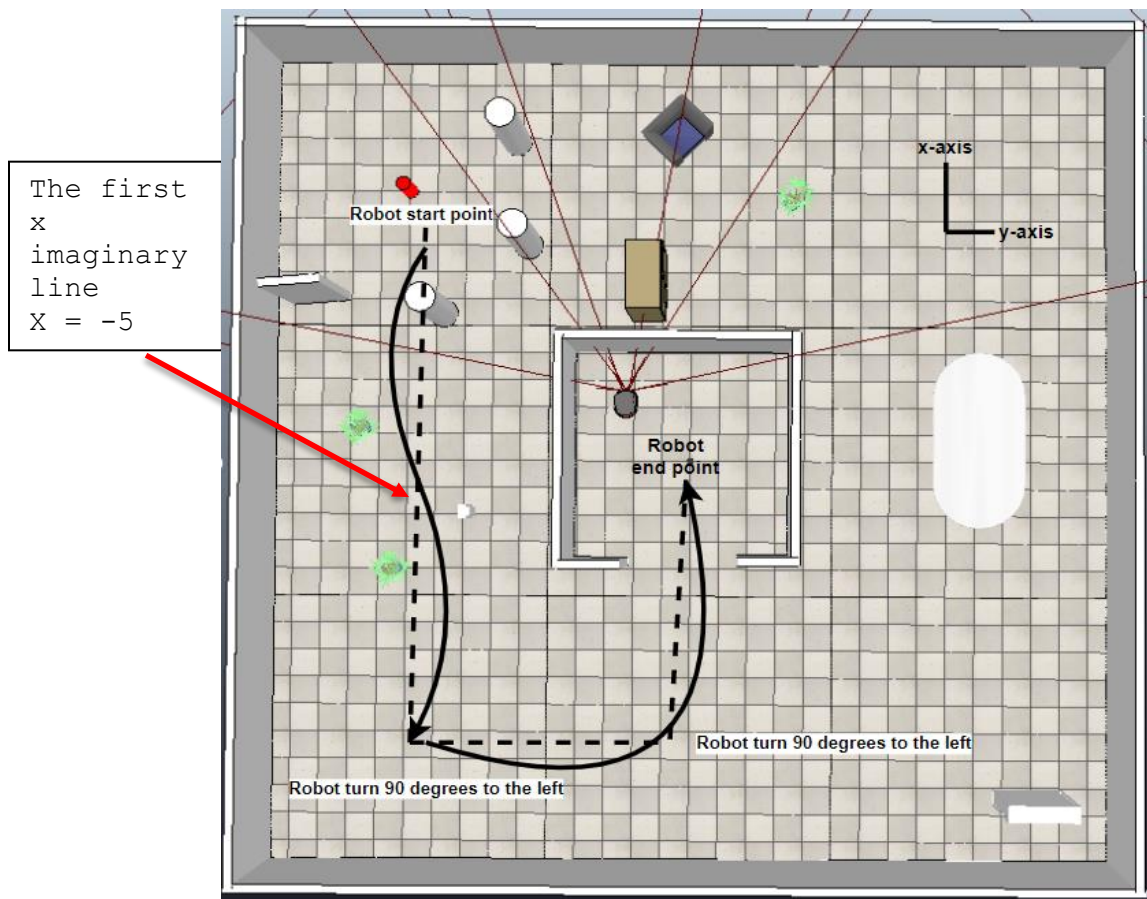


Figure 8. Imaginary lines at the x and y coordinates

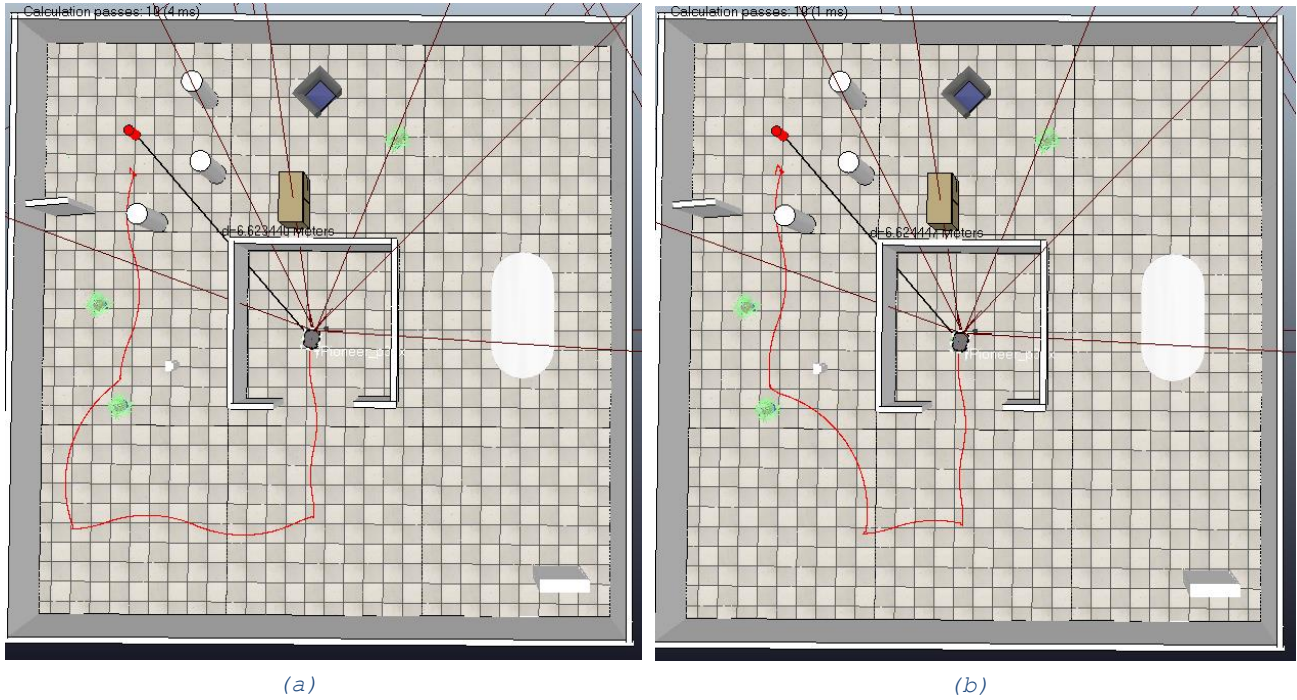


Figure 9. Pioneer robot path from the beacon to the center

The robot was tested in two different scenarios on the way back to the center, in the first scenario shown in fig 9 (a), the robot made a slight right turn to avoid the obstacle and then started to merge into the imaginary x line again until it reached the stop point. The same case is in fig 9 (b) but in opposite direction.

```

ib = True
while ib: ## while loop for the first imaginary x line = -5
    code, position = vp.simxGetObjectPosition(clientID, self.p3dx, -1, vp.simx_opmode_streaming)## P3dx live position
    if position[0] > -5: ## if robot x position is bigger than -5 turn slightly to the left
        robot.turn(1.8, 2)
    if position[0] < -5: ## if robot x position is smaller than -5 turn slightly to the right
        robot.turn(2, 1.8)
    if position[1] <= 0.40: ## if robot y position at 0.40 stop the robot
        robot.turn(0, 0)
        time.sleep(3)
        ib = False

ab = True
while ab:## this while loop turns the robot 90 degrees to the left
    code, orientation = vp.simxGetObjectOrientation(clientID, self.p3dx, -1, vp.simx_opmode_buffer)
    robot.turn(1, -1)
    # if orientation[2] == 0:
    #     robot.turn(0,0)
    #     ab = False
    if orientation[2] > -0.3 and orientation[2] < -0.1:## if robot turned 90 degrees stop
        robot.turn(0, 0)
        time.sleep(1)
        ab = False

```

Figure 10. the first imaginary x line

### 3. Conclusion and result

The purpose of this study is to enable the robot to autonomously travel out of the square and detect the beacon and then return to the center. All the objectives of this assignment are achieved successfully, the Pioneer robot started from the square, traveled in the scene, detected the beacon and successfully headed toward the beacon. Meanwhile, mapping and object avoidance were active all the time during the execution of the objectives. It is worth noting that this assignment has been an effective practice to understand mobile robot operations and automation. Furthermore, a considerable amount of knowledge and experience has been gained during this assignment's research and implementation phase, especially how to utilize CoppeliaSim simulation platform with programming languages to develop, test and improve mobile robotic projects as a professional.

Future studies and innovation can be considered to build fully or partly AI-powered mobile robotics to achieve the same given task in this assignment with a more innovative approach and method. For example, beacon detection could be done by using AI-powered object recognition and detection instead of using conventional methods. This approach could be done by creating enough amount of datasets for the beacon and then training a convolutional neural network (CNN) to recognize and follow the object (the beacon). Another innovative approach is to utilize some algorithms such as A\* to enable the robot to find the shortest way or path to a specific point.

The following fig 11 displays the path of the robot during the execution, the robot traveled in the scene to find the beacon. During the mission, obstacles were avoided, and a clear representation of the scene was mapped.

The beacon is surrounded by many obstacles to test the ability and maneuverability performance of the purposed design.

Some methods such as mapping and drawing the robot path implemented in the older version of CoppeliaSim.

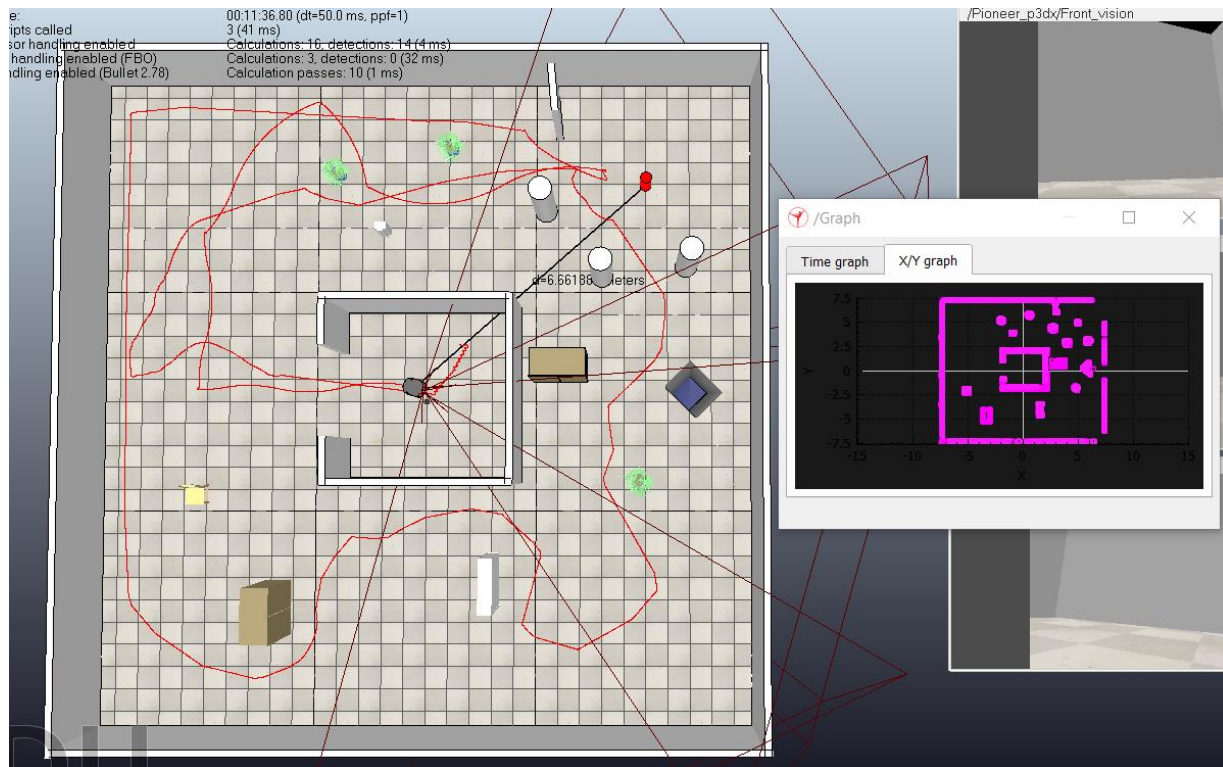


Figure 11. Pioneer robot path and scene mapping

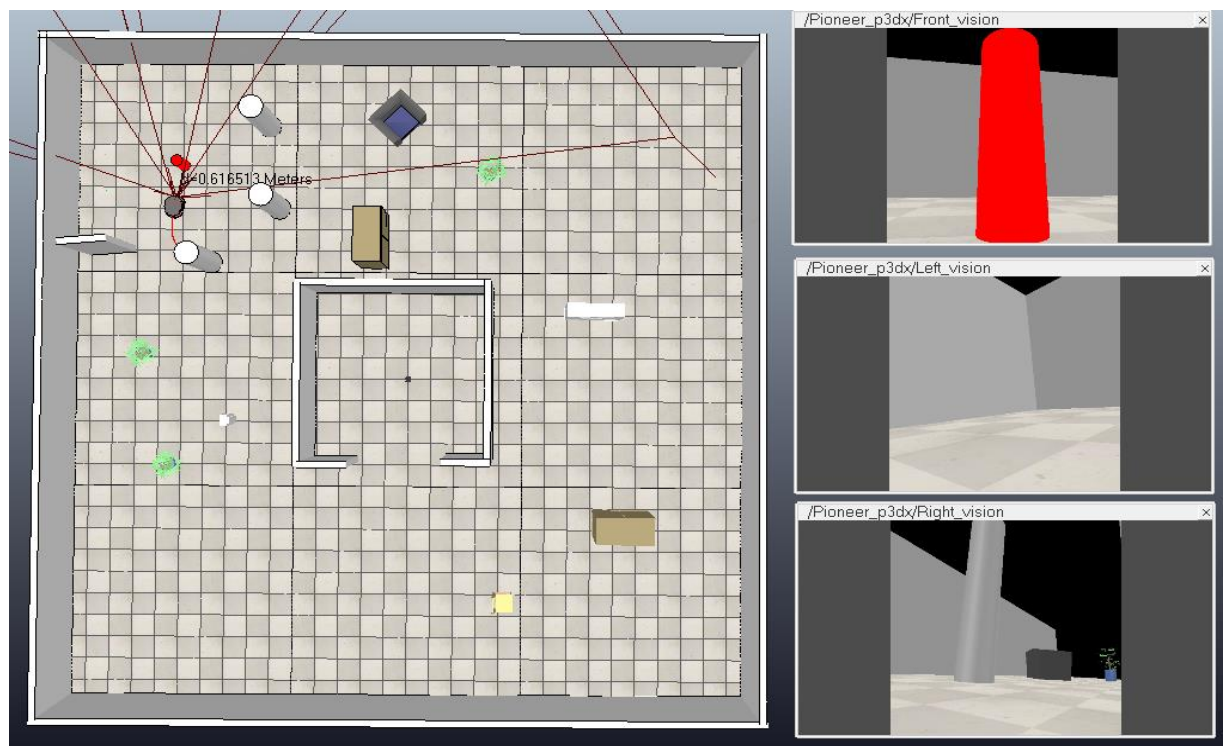


Figure 12. Beacon detected from the front vision sensor



## 4. References

- [1] <https://www.youtube.com/watch?v=PwGY8PxOOXY>
- [2] <https://www.youtube.com/watch?v=SQont-mTnfM&t=878s>
- [3] <https://www.coppeliarobotics.com/helpFiles/en/remoteApiFunctionsPython.htm>
- [4] <https://github.com/topics/coppeliassim?l=python>
- [5] <https://medium.com/@afakharany93/path-tracking-tutorial-for-pioneer-robot-in-vrep-acffc76875b>
- [6] D. S. O. Correa, D. F. Sciotti, M. G. Prado, D. O. Sales, D. F. Wolf and F. S. Osorio, "Mobile Robots Navigation in Indoor Environments Using Kinect Sensor," *2012 Second Brazilian Conference on Critical Embedded Systems*, 2012, pp. 36-41, doi: 10.1109/CBSEC.2012.18.
- [7] M. M. Almasri, A. M. Alajlan and K. M. Elleithy, "Trajectory Planning and Collision Avoidance Algorithm for Mobile Robotics System," in *IEEE Sensors Journal*, vol. 16, no. 12, pp. 5021-5028, June15, 2016, doi: 10.1109/JSEN.2016.2553126.
- [8] E. Xu, Z. Sun, Z. Wang, X. Yu, C. Hu and M. Faied, "Pioneer P3-DX Robot to Achieve Self Driving Car," *2018 IEEE International Conference on Electro/Information Technology (EIT)*, 2018, pp. 0234-0239, doi: 10.1109/EIT.2018.8500258.
- [9] [https://www.inf.ufrgs.br/~prestes/Courses/Robotics/manual\\_pioneer.pdf](https://www.inf.ufrgs.br/~prestes/Courses/Robotics/manual_pioneer.pdf)
- [10] <https://cyberbotics.com/doc/guide/pioneer-3dx>
- [11] S. Syntakas, K. Vlachos and A. Likas, "Object Detection and Navigation of a Mobile Robot by Fusing Laser and Camera Information," *2022 30th Mediterranean Conference on Control and Automation (MED)*, 2022, pp. 557-563, doi: 10.1109/MED54222.2022.9837249.
- [12] <http://www.controlsystems-lab.gr/main/mobile-robots/>
- [13] Doran, M., Sterritt, R., & Wilkie, FG. (2020). Autonomic architecture for fault handling in mobile robots. *Innovations in Systems and Software Engineering*. A NASA Journal, 16(3-4), 263-288. [ ISSE-D-19-00010R1]. <https://doi.org/10.1007/s11334-020-00361-8>
- [14] <https://www.youtube.com/watch?v=SQont-mTnfM&t=879s>
- [15] <https://www.youtube.com/watch?v=OfpB87pRoUk>
- [16] [https://www.youtube.com/watch?v=kOjQRYmeX\\_o&t=1472s](https://www.youtube.com/watch?v=kOjQRYmeX_o&t=1472s)
- [17] <https://www.youtube.com/watch?v=xI-ZEewIzzI&t=921s>
- [18] <https://www.youtube.com/watch?v=YsZG4pL0Ges>



## Appendix

```
import math
import winsound
import numpy as np
import random
import sim as vp
import matplotlib.pyplot as plt
import time

print("Hello Dr.Uzor!")
s_values = [] ## sensor 7 readings will be appended to this list
class Robot(): # the main Robot class with several methods
    def __init__(self):
        for i in range(1, 17): ## this loop used to activate all the sensors in the robot by
            looping through the sensors
                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.pos = vp.simxGetObjectHandle(clientID, "Dummy", vp.simx_opmode_blocking)

        # activating left and right 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)

        code, self.p3dx = vp.simxGetObjectHandle(clientID, "Pioneer_p3dx", vp.simx_opmode_blocking)
        code, self.camera_1 = vp.simxGetObjectHandle(clientID, "Front_vision",
            vp.simx_opmode_oneshot_wait)
        code, self.camera_2 = vp.simxGetObjectHandle(clientID, "Left_vision",
            vp.simx_opmode_oneshot_wait)
        code, self.camera_3 = vp.simxGetObjectHandle(clientID, "Right_vision",
            vp.simx_opmode_oneshot_wait)

        code, self.beacon = vp.simxGetObjectHandle(clientID, "beacon", vp.simx_opmode_blocking)
        code, self.get_dist = vp.simxGetDistanceHandle(clientID, "beacon", vp.simx_opmode_blocking)
        code, self.read_dist = vp.simxReadDistance(clientID, self.get_dist,
            vp.simx_opmode_streaming)
        code, beacon_distance_robot = vp.simxCheckDistance(clientID, self.p3dx, self.beacon,
            vp.simx_opmode_streaming)
        code, self.graph = vp.simxGetObjectHandle(clientID, 'Graph', vp.simx_opmode_blocking)

        code, self.min_dist = vp.simxCheckDistance(clientID, self.p3dx, self.beacon,
            vp.simx_opmode_streaming)
        code, detection_state, packets = vp.simxReadVisionSensor(clientID, self.camera_1,
            vp.simx_opmode_streaming)
        code, detection_state1, packets1 = vp.simxReadVisionSensor(clientID, self.camera_2,
            vp.simx_opmode_streaming)
        code, detection_state2, packets2 = vp.simxReadVisionSensor(clientID, self.camera_3,
            vp.simx_opmode_streaming)
```

```

def move(self): ## method created to move the robot with default speed 1.5 for both motors
    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):## method to steer the robot with two attributes left and
right steering
    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.pos, -1, vp.simx_opmode_blocking)
    y = np.array(position)
    # return np.linalg.norm(y)
    return position

def orientation(self):
    code, ori = vp.simxGetObjectOrientation(clientID, self.pos, -1, vp.simx_opmode_streaming)
    return ori

def set_position(self):
    code = vp.simxSetObjectPosition(clientID, self.pos, -1, [-0.30, 0.30],
vp.simx_opmode_oneshot)
    return code

def set_joint(self):
    code1 = vp.simxSetJointTargetVelocity(clientID, self.R_motor, 0.5, vp.simx_opmode_oneshot)
    code2 = vp.simxSetJointTargetVelocity(clientID, self.L_motor, 0.5, vp.simx_opmode_oneshot)

    code3 = vp.simxSetJointTargetPosition(clientID, self.R_motor, 0.30,
vp.simx_opmode_streaming)
    code4 = vp.simxSetJointTargetPosition(clientID, self.L_motor, 0.30,
vp.simx_opmode_streaming)
    return code3, code4

def wandering(self): ## method to allow the robot wander if no wall detected
    x = random.randint(0, 3)
    y = random.randint(0, 3)
    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): ## method to measure the distance with one attribute sensor
    returnCode, detection, detection_point, detection_object_handle, surface_normal =
vp.simxReadProximitySensor(
    clientID, sensor, vp.simx_opmode_buffer)
    x = np.array(detection_point) ## converting the x, y and z values to array. in order to
linalg.norm function
    if detection == True:
        return np.linalg.norm(x) ## normalizing the x, y and z coordinates to have a single
return value
    else:

```



```
    return 1.7
```

```
def avoid_obstacle(self): ## turn robot both sides left or right. it dependes on the sensor reading
```

```
    if robot.distance(s_values[2]) < 0.9:
        robot.turn(0,2)
    elif robot.distance(s_values[7]) < 0.9:
        robot.turn(2, 0)
    else:
        robot.turn(0, 1)
```

```
def out_of_square(self):
```

```
    # vp.simxSetObjectPosition(clientID, self.p3dx, -1, [0.0250, 0.1750, 0.0000],
vp.simx_opmode_oneshot_wait)
    # error, code = vp.simxGetObjectPosition(clientID, self.p3dx, -1, vp.simx_opmode_streaming)
    # x_coordinate = round(code[0], 2)
    # y_coordinate = round(code[1], 2)
```

```
def check_center():
```

```
    error, code = vp.simxGetObjectPosition(clientID, self.p3dx, -1,
vp.simx_opmode_streaming)
    x_coordinate = round(code[0], 2)
    y_coordinate = round(code[1], 2)
    flagg = True
    if x_coordinate > -0.35 and x_coordinate < 0.20 and y_coordinate > -0.35 and
y_coordinate < 0.35 and x_coordinate != 0 and y_coordinate != 0:
        robot.turn(0, 0)
        print("this is the centre!")
```

```
    while flagg:
```

```
        if robot.distance(s_values[3]) == 1.7 and robot.distance(s_values[4]) == 1.7:
            robot.turn(0, 0)
            for i in range(170):
                robot.turn(2, 2)
```

```
            robot.turn(0, 0)
            print("1")
            flagg = False
```

```
        else:
```

```
            robot.turn(-1, 1.5)
            print("2")
```

```
    print("3")
```

```
    print("4")
    print("no")
    ibo = False
    return ibo
```

```
def hor():
```

```
    if robot.distance(s_values[3]) < 1.6 and robot.distance(s_values[4]) < 1.6:
        robot.turn(-1, -1)
        print(f" front {robot.distance(s_values[3])} and
{robot.distance(s_values[4])}")
        print("con 1")
    elif robot.distance(s_values[3]) > 1.9 and robot.distance(s_values[4]) > 1.9:
        robot.turn(1, 1)
        print(f" front {robot.distance(s_values[3])} and
```

```

{robot.distance(s_values[4])}")
        print("con 2")

        # else:
        #     robot.turn(0, 0)

    def ver():

        if round(robot.distance(s_values[8]) + robot.distance(s_values[9]), 1) >
round(robot.distance(s_values[0]) + robot.distance(s_values[15]), 1):
            print("turn right")
            robot.turn(2, 2)
            robot.turn(-1, 1)
            robot.turn(-1, 1)
            robot.turn(-1, 1)
            robot.turn(-1, 1)
            robot.turn(-1, 1)
            robot.turn(-1, 1)
            robot.turn(-1, 1)
            robot.turn(-1, 1)

            elif round(robot.distance(s_values[8]) + robot.distance(s_values[9]), 1) <
round(robot.distance(s_values[0]) + robot.distance(s_values[15]), 1):
                print("turn left")
                robot.turn(2, 2)
                robot.turn(1, -1)
                robot.turn(1, -1)
                robot.turn(1, -1)
                robot.turn(1, -1)
                robot.turn(1, -1)
                robot.turn(1, -1)
                robot.turn(1, -1)
                robot.turn(1, -1)

        flag = True
        while flag:
            error, code = vp.simxGetObjectPosition(clientID, self.p3dx, -1,
vp.simx_opmode_streaming)
            x = round(code[0], 2)
            if x < 1.78 and x > -2.40:
                print("yes")
                hor()
                ver()
                check_center()
            else:
                flag = False

        print("Robot")
        return "out of square"

    def robot_position(self):
        code, position = vp.simxGetObjectPosition(clientID, self.p3dx, -1, vp.simx_opmode_blocking)

```

```

        # code, angle = vp.simxGetObjectOrientation(clientID, self.p3dx, -1,
vp.simx_opmode_streaming)
        # print(position)
        # time.sleep(3)
        # print(round(position[0], 2))
        # print(round(position[1], 2))
        # time.sleep(3)
        # print(f" this is object orientation {round(angle[1], 2)}")
        return round(position[0], 2), round(position[1], 2)

    def vision_sensor(self):
        code, resolution, image = vp.simxGetVisionSensorImage(clientID, self.camera_1, 0,
vp.simx_opmode_buffer)
        im = np.array(image, dtype= np.uint8)
        im.resize([resolution[0], resolution[1], 3])
        plt.imshow(im)

        return plt.imshow(im), image

    def detect_red_color(self):
        total_red_pixels = []

        for j in range(0, len(self.vision_sensor()[1]), 3):

            print(f" this is j {self.vision_sensor()[1][j]}")
            total_red_pixels.append(j)
            y = total_red_pixels.count(-1)
            print(f" total if red pixels are {y}")
            print(f" len is {len(total_red_pixels)}")
            print(total_red_pixels)

    def read_vioion(self):

        code, detection_state, packets = vp.simxReadVisionSensor(clientID, self.camera_1,
vp.simx_opmode_buffer)
        codd1, detection_state1, packets1 = vp.simxReadVisionSensor(clientID, self.camera_2,
vp.simx_opmode_buffer)
        code2, detection_state2, packets2 = vp.simxReadVisionSensor(clientID, self.camera_3,
vp.simx_opmode_buffer)

        return packets[0][6], packets1[0][6], packets2[0][6]

    def beacon_position(self):
        code, position = vp.simxGetObjectPosition(clientID, self.beacon, -1,
vp.simx_opmode_streaming)

        return position

    def guided_robot(self):
        code = vp.simxSetObjectPosition(clientID, self.p3dx, -1, [4.15, 4.20, 0.40],
vp.simx_opmode_oneshot)
        return code

    def wheel_adjustment(self):
        if self.read_vioion()[0] == True:

```

```

adjust_value = self.read_vioion()[1]
threshold_value = 0.5
correction = threshold_value - adjust_value
turninig_rate = 1.1

```

```

def avoid_obstcls(self):
    flag = True
    while flag:
        code, orientation = vp.simxGetObjectOrientation(clientID, self.p3dx, -1,
vp.simx_opmode_buffer)
        print(orientation[2])

        print("obstcls in exe")
        # robot.turn(2, 2)
        if robot.distance(s_values[3]) < 0.6:## 212 front left
            robot.turn(-1, 1)
            return True
        elif robot.distance(s_values[4]) < 0.6: ## 216 front right
            robot.turn(1, -1)
            return True
        if robot.distance(s_values[2]) < 0.6: ## 212 front-left
            robot.turn(-1, 1)
            return True
        elif robot.distance(s_values[5]) < 0.6: ## 216 front-right
            robot.turn(1, -1)
            return True
        if robot.distance(s_values[1]) < 0.4: ## 212 left side
            robot.turn(-0.7, 0.7)
            return True
        elif robot.distance(s_values[6]) < 0.4:## 216 right side
            robot.turn(0.7, -0.7)
            return True
        else:
            print("end")
            flag = False
            return False
    def turn_axis(self):
        code, orientation = vp.simxGetObjectOrientation(clientID, self.p3dx, -1,
vp.simx_opmode_buffer)
        print(orientation[2])
        while orientation[2] < 2.8 and orientation[2] > -2.8:
            code, orientation = vp.simxGetObjectOrientation(clientID, self.p3dx, -1,
vp.simx_opmode_buffer)
            robot.turn(-1, 1)
    def turn_axis_2(self):
        code, orientation = vp.simxGetObjectOrientation(clientID, self.p3dx, -1,
vp.simx_opmode_buffer)
        print(orientation[2])
        while orientation[2] < 1.30 and orientation[2] > 1.40:
            code, orientation = vp.simxGetObjectOrientation(clientID, self.p3dx, -1,
vp.simx_opmode_buffer)
            robot.turn(-1, 1)

    # return detection_state, packets[0][1], packets[0][6], packets[0][11]
def return_home(self):
    print("hellooooooooo")

```

```

        code, position = vp.simxGetObjectPosition(clientID, self.p3dx, -1,
vp.simx_opmode_streaming)
        code, orientation = vp.simxGetObjectOrientation(clientID, self.p3dx, -1,
vp.simx_opmode_streaming)

        code, position = vp.simxGetObjectPosition(clientID, self.p3dx, -1,
vp.simx_opmode_streaming)
        robot.wandering()
        print(position[0])
        if position[0] > -0.50:
            robot.turn_axis()
        flag = True
        while flag:
            code, position = vp.simxGetObjectPosition(clientID, self.p3dx, -1,
vp.simx_opmode_streaming)
            robot.avoid_obstcls()
            robot.turn(2, 2)
            if position[0] < -4.00:
                flag = False

        robot.turn_axis_2()
        for i in range(120):
            robot.turn(2, 2)
        return print(position), print(orientation)

    def correct_heading(self):
        code, orientation = vp.simxGetObjectOrientation(clientID, self.p3dx, -1,
vp.simx_opmode_buffer)
        code, position = vp.simxGetObjectPosition(clientID, self.p3dx, -1,
vp.simx_opmode_streaming)
        print(orientation[2])
        print(position[0])
        ibo = True
        ah = True
        while ibo:
            code, position = vp.simxGetObjectPosition(clientID, self.p3dx, -1,
vp.simx_opmode_streaming)
            code, orientation = vp.simxGetObjectOrientation(clientID, self.p3dx, -1,
vp.simx_opmode_buffer)

            while ah:
                code, orientation = vp.simxGetObjectOrientation(clientID, self.p3dx, -1,
vp.simx_opmode_buffer)
                if orientation[2] < 2.8 and orientation[2] > -2.8:
                    robot.turn(-1, 1)
                if orientation[2] > 2.8 and orientation[2] < 3:
                    robot.turn(0, 0)
                    time.sleep(3)
                    ah = False

            # robot.avoid_obstcls()
            if robot.distance(s_values[3]) < 0.5:
                robot.turn(-1, 1)
            if robot.distance(s_values[2]) < 0.5:
                robot.turn(-1, 1)
            if robot.distance(s_values[4]) < 0.4:
                robot.turn(1, -1)
            if robot.distance(s_values[5]) < 0.4:

```

[illegible]

```

        # robot.turn(2, 2)
        ab = True
        while ab:
            code, orientation = vp.simxGetObjectOrientation(clientID, self.p3dx, -1,
vp.simx_opmode_buffer)
            robot.turn(1, -1)
            # if orientation[2] == 0:
            #     robot.turn(0,0)
            #     ab = False
            if orientation[2] > -0.3 and orientation[2] < -0.1:
                robot.turn(0, 0)
                time.sleep(1)
                ab = False

        ia = True
        while ia:
            code, position = vp.simxGetObjectPosition(clientID, self.p3dx, -1,
vp.simx_opmode_streaming)
            if position[1] > 0.30: ## y axis
                robot.turn(0.9, 1)
            if position[1] < 0.30:
                robot.turn(1, 0.9)
            if position[0] > -0.30:
                robot.turn(0 ,0)
                time.sleep(3)
                print("THE IS THE END!!!")
                ia = False

vp.simxFinish(-1)
clientID = vp.simxStart('127.0.0.1', 706, True, True, 19994, 5)
if clientID != -1:
    print('Connected to API Coppelia server')
    robot = Robot()
    robot.out_of_square()
    print("hello world")
    emp_list = [0]
    # robot.set_object()

    # robot.set_position()
    # robot.set_joint()
    code, resolution, image = vp.simxGetVisionSensorImage(clientID, robot.camera_1, 0,
vp.simx_opmode_streaming)
    code, resolution1, image1 = vp.simxGetVisionSensorImage(clientID, robot.camera_2, 0,
vp.simx_opmode_streaming)
    code, resolution2, image2 = vp.simxGetVisionSensorImage(clientID, robot.camera_3, 0,
vp.simx_opmode_streaming)
    code, detection, packet = vp.simxReadVisionSensor(clientID, robot.camera_1,
vp.simx_opmode_buffer)
    ibo = True
    while ibo:
        # robot.turn(0, 0)
        # robot.wandering
        code, orientation = vp.simxGetObjectOrientation(clientID, robot.p3dx, -1,
vp.simx_opmode_streaming)
        print(orientation[2])

```

```

    if robot.avoid_obstcls() == True:
        robot.avoid_obstcls()

    # robot.avoid_obstcls()
    if robot.read_vioion()[0] == 1 or robot.read_vioion()[1] == 1 or robot.read_vioion()[2] ==
1:
        print("object inside frame")
        code, position_of_beacon = vp.simxGetObjectPosition(clientID, robot.beacon, -1,
vp.simx_opmode_streaming)
        flage = True
        while flage:
            code, beacon_robot_dist = vp.simxCheckDistance(clientID, robot.p3dx, robot.beacon
, vp.simx_opmode_buffer)
            # code, position_of_beacon = vp.simxGetObjectPosition(clientID, robot.beacon, -1,
vp.simx_opmode_buffer)
            print(f" distance to robot {beacon_robot_dist}")
            robot.turn(2, 2)

            if beacon_robot_dist < 0.6:
                robot.turn(0, 0)
                robot.turn(0, 0)
                robot.turn(0, 0)
                robot.turn(0, 0)
                robot.turn(0, 0)
                robot.turn(0, 0)
                robot.turn(0, 0)
                robot.turn(0, 0)
                robot.turn(0, 0)
                robot.turn(0, 0)
                robot.turn(0, 0)

            flage = False
            ibo = False

    if robot.avoid_obstcls() == True and robot.read_vioion()[0] == 1:
        robot.avoid_obstcls()
    if robot.avoid_obstcls() == True and robot.read_vioion()[1] == 1:
        robot.avoid_obstcls()
    if robot.avoid_obstcls() == True and robot.read_vioion()[2] == 1:
        robot.avoid_obstcls()
    if robot.avoid_obstcls() == False:
        # print(position_of_beacon)

        if robot.read_vioion()[0] == 1 and robot.read_vioion()[1] == 1: ## beacon
appeared in the left and front vision sensor
            robot.turn(2, -1) ## turn left
            robot.turn(2, 1) ## make a smooth turn to the left
            # robot.turn(2, -2)
            print("turn left")
        if robot.read_vioion()[0] == 1 and robot.read_vioion()[2] == 1: ## beacon
appeared in the right and front vision sensor
            robot.turn(-1, 2) ## turn right
            robot.turn(1, 2) ## make a smooth turn to the right
            # robot.turn(-2, 2)
            print("turn right")

```



```

        if robot.read_vioion()[0] == 1 and robot.read_vioion()[1] != 1 and
robot.read_vioion()[2] != 1: ## beacon ONLY appeared in the front vision sensor
            robot.turn(2, 2)
            print("go ahead")

        if robot.read_vioion()[1] == 1 and robot.read_vioion()[0] != 1: ## beacon
appeared ONLY in the left vision sensor
            robot.turn(2, -1) ## turn left
            robot.turn(2, 1) ## make a smooth turn to the left
            # robot.turn(2, -2)
            # robot.turn(2, -2)
            print("turn left")
        if robot.read_vioion()[2] == 1 and robot.read_vioion()[0] != 1: ## beacon
appeared ONLY in the right vision sensor
            robot.turn(-1, 2) ## turn right
            robot.turn(1, 2) ## make a smooth turn to the left
            # robot.turn(-2, 2)
            # robot.turn(-2, 2)
            print("turn right")

    else:
        robot.wandering()
        print("Wandering")
        robot.turn(-2, -2)
        robot.turn(-2, -2)
        robot.turn(-2, -2)
        robot.turn(-2, -2)
        robot.turn(-2, -2)
        robot.turn(-2, -2)
        robot.turn(-2, -2)

        robot.correct_heading()
        print("THIS IS THE END")

        # robot.find_centre()
        # robot.return_home()
else:
    print('Not able to connect to API!')

```