FSM – Wall Following Mobile Robot Pioneer-3DX

Technical Report

Robert Barbulescu, MSc Intelligent Systems & Robotics

Number of words: 3871, Number of pages: 16.

Contents

1 Introduction

According to Ghosh et al. (2017) and Orozco-Rosas et al. (2019) path planning is the foundation of control and navigation for mobile robots. Depending on the amount of information known about the environment, path planning can be divided into *global path planning* and *local path planning* (Yu et al., 2020, p.14). The global path planning is "determining a path in configuration space between the initial configuration of the robot and a final configuration such that the robot does not collide with obstacles and the planned motion is consistent with the kinematic constraints of the vehicle", in simple terms, is the process of deciding the best way to move the robot from a start location to an end location (Terzimehic et al., 2011, p.1). Furthermore, Terzimehic et al. (2011) describes it as a way of planning that "encompasses all of the robot's acquired knowledge to reach a goal" and is usually run as a planning phase before the robot begins its journey. On the other hand, local (reactive) navigation does not require prior knowledge of the environment as the robot reacts to the detected obstacle and changes its heading direction in real time, using sensors, to avoid the obstacle (Lim Chee Wang et al., 2002, p.821). While the global path planning is most suitable approach for the purpose of our work, as we will be using a structured environment, it is often implemented through algorithms in order to determine the most efficient path and avoid obstacles. Since the task the robot will need to complete do not require such advanced methods, we will not be using algorithms but rather focus on the robot's sensors to avoid obstacles and follow our determined path.

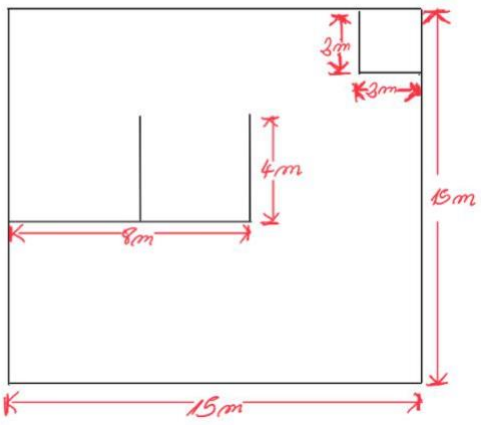
1.1 Purpose

The work presented in this report considers creating a program that will allow the Pioneer 3DX mobile robot to wander randomly within a given environment until a wall is detected and follow it in a determined direction. The program requires the creation of a randomized movement, use of the robot's sensors, and implementation of various parameters to allow for the path following to be accurate. For this purpose, a python script has been designed and tested using an environment created in CoppeliaSim Edu (V-REP) to control the robot's movement through a python remote API, the program allows for various parameters to be inputted such as: speed, distance, angle turning, etc. The main objective of this project is to design a successful program meeting the requirements presented above and to obtain information from the results concerning the routing and path the robot will take to complete this task, as well as devise ways to improve the program in order to: make the path smoother, avoid near misses, and spend less time overshooting corrections. Finally, we will be looking at existing alternatives ways to complete this task.

1.2 Environment

As previously mentioned, the following specifications for the environment along with the scene were provided (Figure 1):

Figure 1 - Map design & measurements.



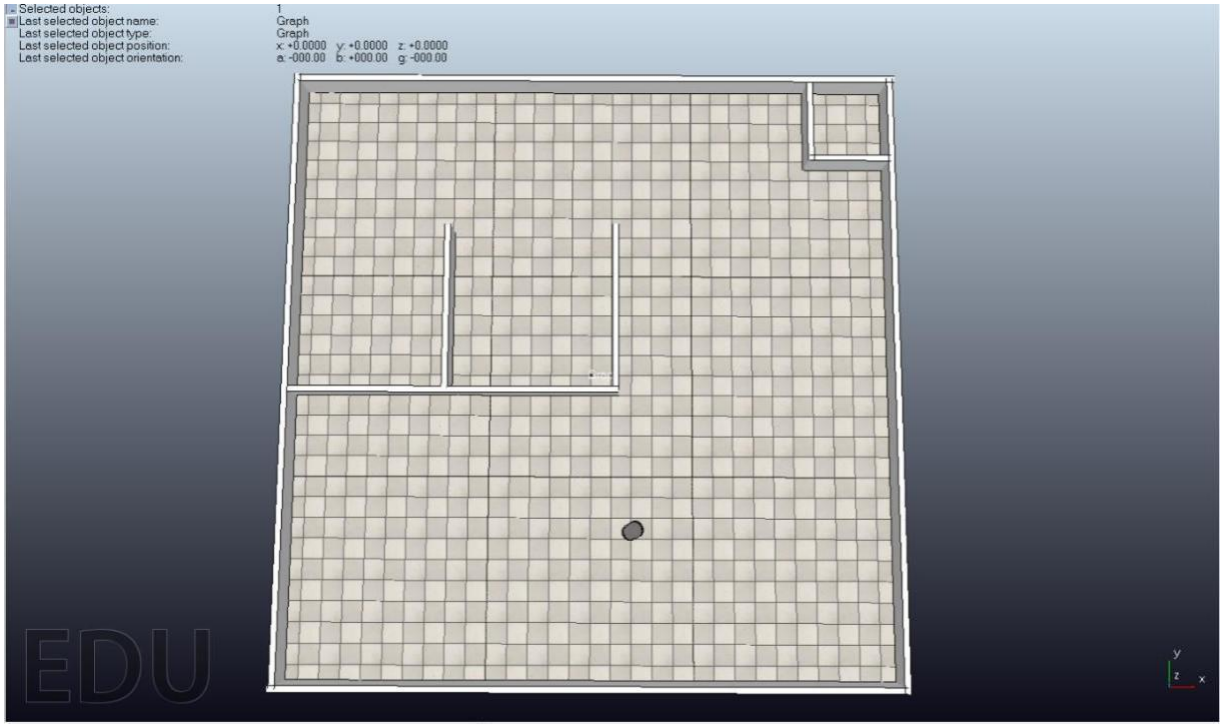
- The map compromises a wall which is the skeleton of the path to follow,

- It contains lines that indicate the ideal path that the robot should follow, and

- A bounding box around the area so the robot cannot escape.

Our map, Figure 1, compromises a square plane surrounded by a 15 meters bounding box, with addition of a 2 by 2 meter smaller box placed inside the surrounding walls in the far right corner, and a further 8 by 4 meter walls.

Figure 2 - Map created in CoppeliaSim.



Using the specifications and measurements provided in Figure 1, the above map was created and provided to us for the purpose of our work.

1.3 Summary of Results

The task was completed successfully, a program was developed that starts at a random position on the scene presented above, takes a random turn, and follows the walls on a determined path. Parameters were changed during the testing phase in order to improve the program, a more efficient configuration was discovered and implemented later on in our report.

2 System Description

2.1 Rationale

Considering the tasks presented in paragraph 1.1 and the specifications provided in paragraph 1.2, we will be using CoppeliaSim as our virtual environment and simulation platform as it includes many types of robots available for use, provides an easy interface, and contains various tools for recording the performance of our program. Furthermore, CoppeliaSim offers a Remote API allowing the control of the mobile robot from an external application such as Python, which was used to create the program that controls the robot's movement. For the purpose of our simulation it was necessary to disable the child script associated with the robot in order for all avoidance obstacles scripts to be developed in Python and the mobile robot to be linked using the Remote API feature of CoppeliaSim.

In regards to the design of our system we choose Python as our developing language for several reasons, such being: the ability to use the **random** function, to take a random value and input it as the velocity and turning angle of the robot's motors, familiarity with language, and easy scalability with the option to easily build on top of our program for improvement.

2.2 Python 3.8

Python is a general-purpose and popular programming language, it is concise and easy to read, and it can be used for everything from web development to software development and scientific applications. Python is also an interpreted, object-oriented, high-level programming language with dynamic semantics and the high-level built in data structures, combined with dynamic typing and dynamic binding, which makes it suitable for our requirements. Furthermore, bugs or bad inputs will not cause a segmentation fault, instead, when the interpreter discovers an error, it raises an exception and when the program doesn't catch the exception, the interpreter prints a stack trace (Python.org, online). This makes it an efficient way for us to debug and deal with any errors during the development phase.

2.3 CoppeliaSim Edu (V-REP)

CoppeliaSim Edu (previously known as V-REP) is a robot simulator with integrated development environment and is based on a distributed control architecture: each object/model can be individually controlled via an embedded script, a plugin, a ROS or BlueZero node, a remote API client, or a custom solution, which makes CoppeliaSim very versatile and ideal for multi-robot applications. Controllers can be written in C/C++, Python, Java, Lua, Matlab or Octave (coppeliarobotics.com, online).We are interested in the python controller especially for our work.

2.4 The Pioneer 3DX

The Pioneer 3DX is a fully programmable research mobile robot, highly popular in laboratory use and university research for its versatility, reliability and durability, the all-purpose base robot is used for research and applications involving: mapping, teleoperation, localization, monitoring, vision, manipulation, multirobot cooperation, autonomous navigation, reconnaissance, etc (Terzimehic et al., 2011, p.2).

The Pioneer 3DX platform compromises the following: motors with 500-tick encoders, 19cm wheels, an aluminium body, and it is equipped with two sets of sonar sensors, 8 forward-facing ultrasonic (sonar) sensors, 8 optional real-facing sonar sensors (Terzimehic et al., 2011, p.2).

Figure 4 - Pioneer 3DX - Servos (Cyberbotics.com, online).

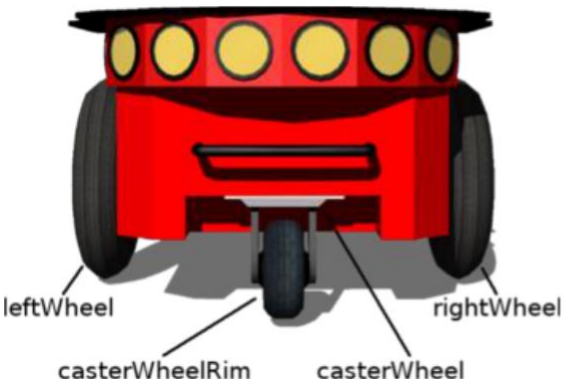
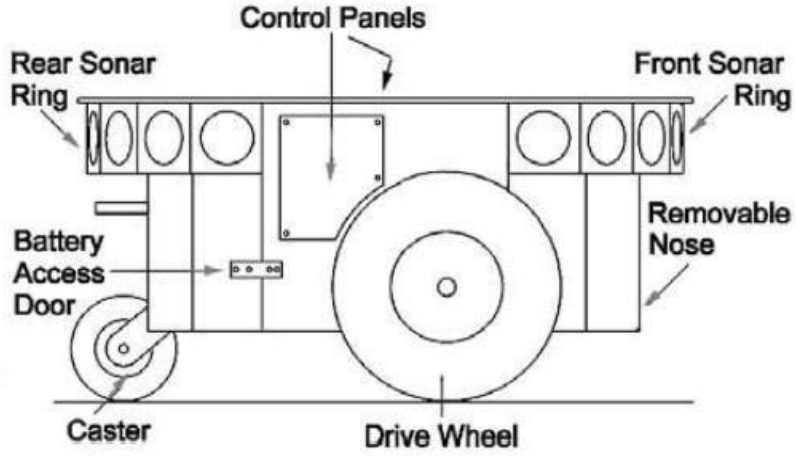


Figure 3 -Pioneer 3DX - LEDs (Cyberbotics.com, online).

The platform also consists of different I/O ports used to connect up to 16 peripheral devices and power sources, there is an onboard computer with four RS-232 serial ports, eight digital I/O ports, five AD ports, Ethernet, PC104 bus with additional ports and a PSU controller, all accessible via a common application connected to the operative system of the mobile robot (Terzimehic et al., 2011, p.2). Unfortunately, the official Adept Mobile Robots website was discontinued and no longer exists, all the information presented above were obtain from journal articles and the full operation manual that was made available by *Cyberbotics.com (online).*

Figure 5 - Pioneer 3DX - Features (Terzimehic et al., 2011, fig.2).



3 System Design

The script file containing the program presented in this report is available at the following GitHub repository: https://github.com/robertandreibarbulescu/IMAT5121-Mobile_Robotics.git.

3.1 Obstacle Avoidance

Several methods currently exist for mobile robots to avoid collision with obstacles, some of which are being implemented using algorithms such as *The Bug Algorithm*, proposed by Lumelsky and Stepanov (1987), *The Model Predictive Control (MPC)*, proposed by Anderson et al. (2010), *The Nearness Diagram (ND) Method*, proposed by Minguez and Montanoother (2000), and other methods implement *Fuzzy Logic Controllers* such as the system proposed by Batti et al. (2019). While the systems presented above use different parameters and sensors to detect and avoid collision with objects in their path, we found that for the purpose of our work such complex methods are not necessary. Instead, our main focus will be the use of the robot's ultrasonic sensors and motors. However, only six sensors were activated and used to complete our tasks, this was decided in order to ensure no sensor readings will overlap and to help in debugging any experimental errors, furthermore the environment presented in Paragraph 1.2 does not require the use of all sixteen sensors as there are a relatively small number of obstacles.

Figure 7 - Pioneer 3DX - Front Sensors.

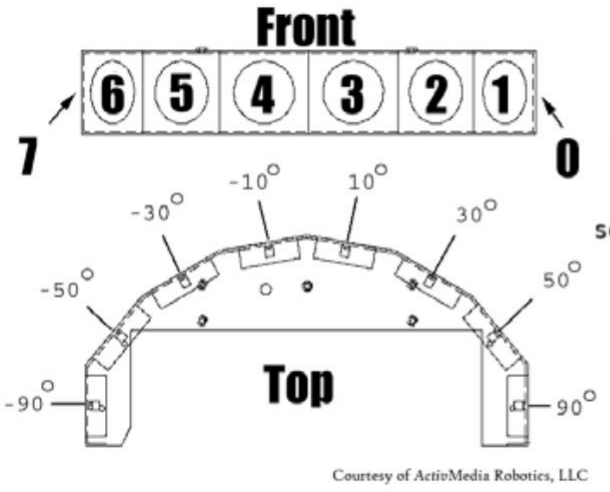
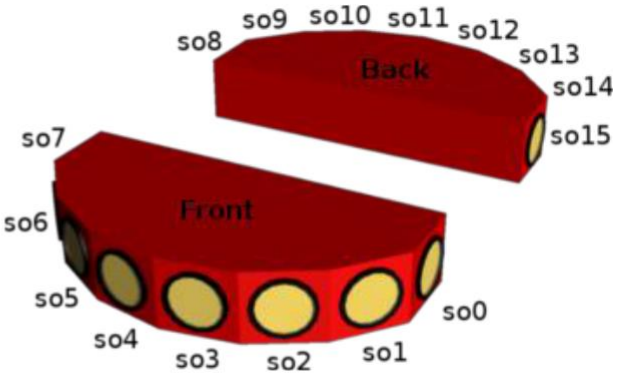


Figure 6 - Pioneer 3DX – Sensors (Cyberbotics.com, online).



3.2 Controlling the Pioneer 3DX Robot

In order to control our robot we are employing a client-server model, which is a way to transmit a script to the device (robot), in our case the robot (CoppeliaSim) is the server, and Python is the client through which we will be sending the commands. During the development process in Python we used the following libraries: **`sim`, `numpy`, `random`, `time`, `math` and `matplotlib`** to implement the functions used to control our robot, as well as the script available in Appendix 8.1 to initiate a connection with our environment CoppeliaSim.

Using the python remote API function **`vrep.simxGetObjectHandle`**, please see Appendix 8.2, which returns an error code and a handle to an object, we can obtain a handle on the robot's motors (Francisco , online). In our case below, we are using **`ClientID`** as a handle to the server for our function, the name of our robot (**`Pioneer_p3dx`**) and the operation mode (**`blocking mode`**).

```
res, self.leftMotor =
vrep.simxGetObjectHandle(clientID,'Pioneer_p3dx_leftMotor',vrep.simx_opmode_blocking)
res, self.rightMotor =
vrep.simxGetObjectHandle(clientID,'Pioneer_p3dx_rightMotor',vrep.simx_opmode_blocking)
```

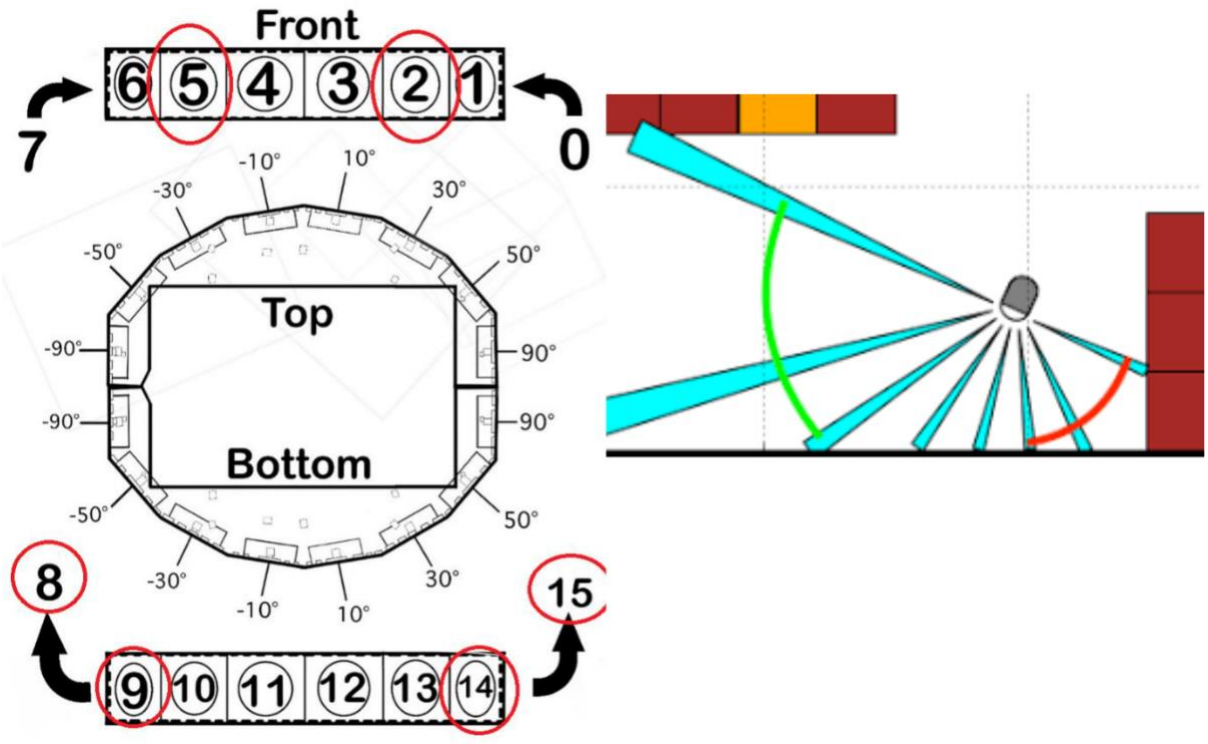
3.3 Path following

An overview of the robot's ultrasonic sensors is presented above in Figure 6, as previously mentioned we are only using six of the available sonars, those being:

- ***number 2 and number 5*** *for detection on the front of the robot,*
- ***number 15*** *for detection on the left side,*
- ***number 8*** *for detection on the right side,*
- ***number 9 and number 14*** *for detection on the back of the robot.*

A clear graphical representation of the sonars we will be using is available below in Figure 8 with a perspective from under the Pioneer 3DX mobile robot:

Figure 8 - Ultrasonic sensors implemented.



According to Francisco (online), sensors work by "detecting objects at a given point and providing information to check the distance and coordinates of the detected object with respect to the sensor", furthermore, sensors need to be initialized (first detection) and then used over and over again.

We continue by using the same function (**vrep.simxGetObjectHandle**) to obtain a handle on the robot's sensors, for the purpose of this example we are only displaying one of the sensors:

```
# obtain handle on sensor 5
res, self.frontLeftSonar = vrep.simxGetObjectHandle(clientID,'Pioneer_p3dx_ultrasonicSen-
sor5',vrep.simx_opmode_blocking)
# initialize sensor 5
res,detectionState,detectedPoint,detectedObjectHandle,detectedSurfaceNormalVector =
vrep.simxReadProximitySensor(clientID,self.frontLeftSonar,vrep.simx_opmode_streaming)
```

Once we obtained a handle on the sensor, we call the **simxReadProximitySensor** function which returns several variables that we can use to determine if the robot detected something: **detectedPoint**, **detectedObjectHandle** and **detectedSurfaceNormalVector**.

The **vrep.simx_opmode_blocking** and **vrep.simx_opmode_streaming** are operation modes of the API functions we are using, when we called the remote API functions those were translated into commands that travel to the server (our robot) and return as a command reply. The operation mode defines what happens to the command and the command reply (coppeliarobotics.com, online). For more information on operation modes please see Appendix 8.3.

Finally, we are using **simxSetJointTargetVelocity** to set the speed and turning angles of our robot, this was done by creating several functions that use **simxSetJointTargetVelocity** to retrieve readings from the sensor defined previously and adjust speed and angles in accordance to our parameters.

While we decided to manually initialize a limited number of sensors for reasons specified earlier, those can be setup and activated together as shown by Nikolai Kummer, the alternative method is available in Appendix 8.4.

3.4 Random Wander

In order to create random movement of the robot in accordance with our task, we imported the python built-in **random** library, which implements pseudo-random number generators.

```
randomNumber = random.randint(0, 100)
```

The **randint()** method returns an integer number selected element from the specified range, in our case the starting position was **1** and the ending position was **100**.

```
robot.turnArc(randomNumber  - 1, randomNumber + 1)
```

As we can see above, we applied the random method to an earlier defined function in our program (**turnArc**) that takes two inputs as the velocity for the left and right motors, this dictates the speed and angle our robot will start to move at, which will be of a random value from 1 to 100.

3.5 Path Following & Random Wander Implementation

Using the handles and sensors initiated in the previous two paragraphs, we created a **nested while loop** containing an **if..elif..else statement**, which we are using to define the rules and parameters by which the robot will follow our determined path. Below we are displaying the first rule and ending statement:

```
while True:
        # the robot will move untill it detects an
        object; robot.move(1)
        # if.. elif ..else statements that allows us to check for
multiple expressions;
        if getDistanceReading(robot.frontLeftSonar) <= 0.5:        # threshold value
            robot.turnArc(-2, 2)                                   # velocity adjustment
            print("Wall detected in front")
            # the robot will trun if a wall is detected;

.

.

       else:
            robot.turnArc(randomNumber - 1, randomNumber + 1)
            # The robot will take a random turn untill it detects an object;
```

In the code presented above, the robot will retrieve and check the sensor reading from the front left sonar, if a wall is detected within **0.5 meters** it will turn at a 90 degrees angle in order to follow the wall, rules that adjust the movement were declared for each of the six sonars in order to achieve path following (please see the full program on GitHub). The speed and angle at which our robot moves is inputted in **`robot.turnArc(-2, 2)`**, -2 (negative value >0) is equivalent with taking a right turn and 2 (positive value <0) is equivalent with taking a left turn.

If no object is detected within the distance we inputted and no rule meets the parameters, than the robot will take a random turn using the function described in the previous paragraph, this is continued until a wall is detected.

4 Testing Results

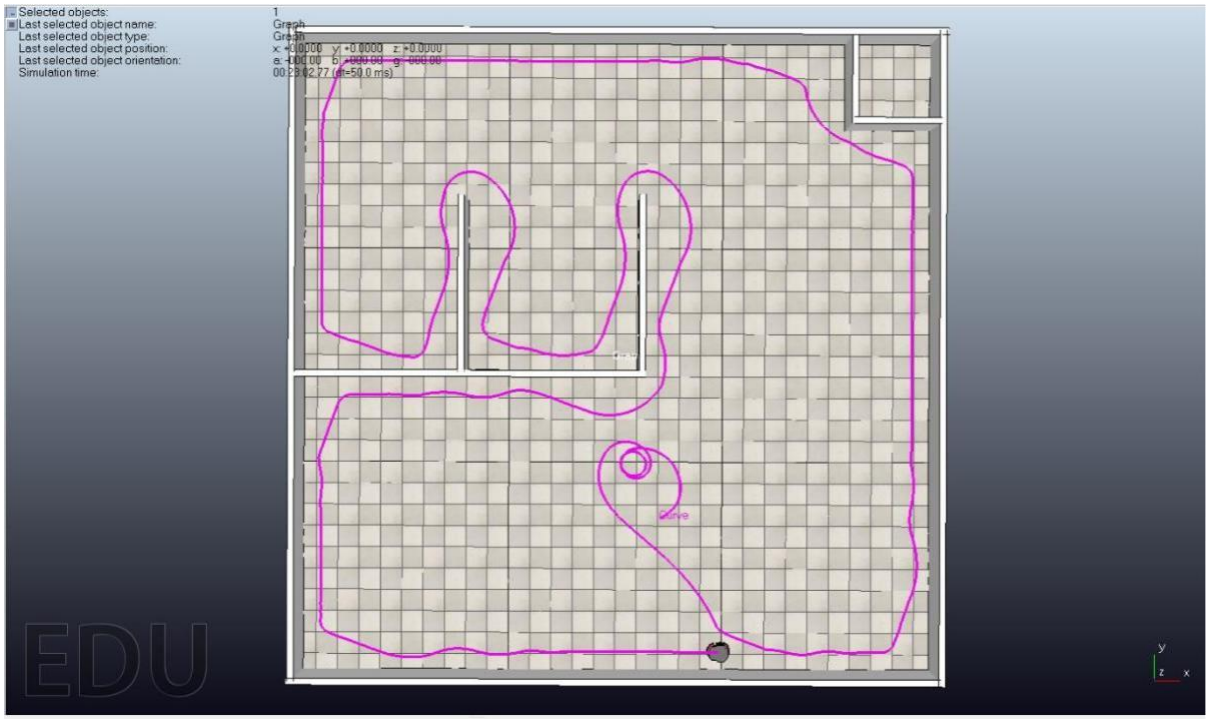
Using the system designed in the previous chapter, we implement the following distance thresholds on which our robot should move if an object is detected:

- ***Pioneer 3DX Front Left Sensor (ultrasonicSensor5) <= 1***
- ***Pioneer 3DX Front Right Sensor (ultrasonicSensor2) <= 1***
- ***Pioneer 3DX Left Side Sensor (ultrasonicSensor15) <= 0.3 & <= 1***
- ***Pioneer 3DX Right Side Sensor (ultrasonicSensor8) <= 0.3 & <= 1***
- ***Pioneer 3DX Back Left Sensor (ultrasonicSensor14) <= 2***
- ***Pioneer 3DX Back Right Sensor (ultrasonicSensor9) <= 2***

Consequently, values ranging from **-2** up to **2** were assigned as velocity adjustments of the speed and angle to each individual rule, for our presentation and contrast of results we will be focusing on the thresholds values (please refer to the program source code for full parameters list).

Using the values presented above we obtained the following results from our first simulation (while we ran the simulation, we used the graph functionality in CoppeliaSim to record the trajectory and time throughout the simulation):

Figure 9 - Simulation 01.



Looking at Figure 9, we can see the starting point of the robot and its random movement up until a wall was detected, it then continued following the wall and adjusting its direction until the simulation was interrupted. The robot was successful in completing the task, the simulation ran for approximately 23 minutes before manually stopped.

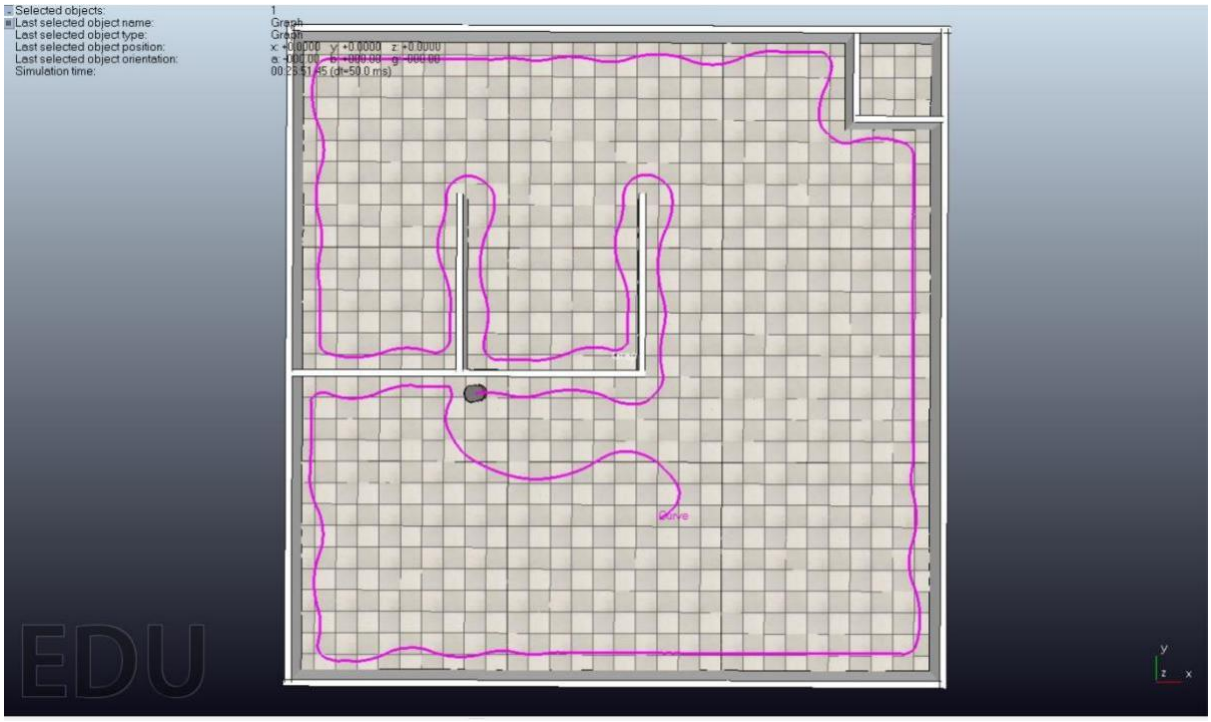
The data registered by the graph for our first simulation was saved for later analysis and comparison with our second simulation.

The task was successfully completed in simulation 01, now we proceed with changing the parameters used in order to test and improve our program, the following threshold distance values were amended (note that speed and turning angles were not changed):

- ***Pioneer 3DX Front Left Sensor (ultrasonicSensor5) <= 0.5***
- ***Pioneer 3DX Front Right Sensor (ultrasonicSensor2) <= 0.5***
- ***Pioneer 3DX Left Side Sensor (ultrasonicSensor15) <= 0.3 & <= 0.5***
- ***Pioneer 3DX Right Side Sensor (ultrasonicSensor8) <= 0.3 & <= 0.5***
- ***Pioneer 3DX Back Left Sensor (ultrasonicSensor14) <= 0.5***
- ***Pioneer 3DX Back Right Sensor (ultrasonicSensor9) <= 0.5***

From our first simulation we observed that the robot performs wide angle turnings when corner walls are detected, in order to correct this we attempted several threshold distances. The above parameters provide the highest efficiency level from the values we tested. The results from our second simulation are presented below:

Figure 10 - Simulation 02.



Our second simulation (Figure 10) shows a visible improvement in the robot's path following, the path the robot takes is smoother and more accurate in comparison to our first simulation, furthermore the angles at which the robot turns when corner walls are detected is significantly improved. The simulation ran for approximately 25 minutes before it was manually stopped, and the graph data was saved for analysis.

We used MATLAB to plot the data gathered from the two simulation, the results are as follows:

Figure 12 - Plotting data from simulation 01.

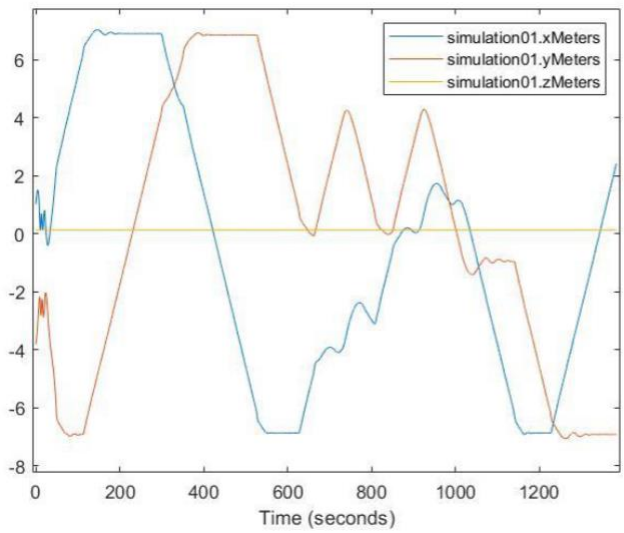
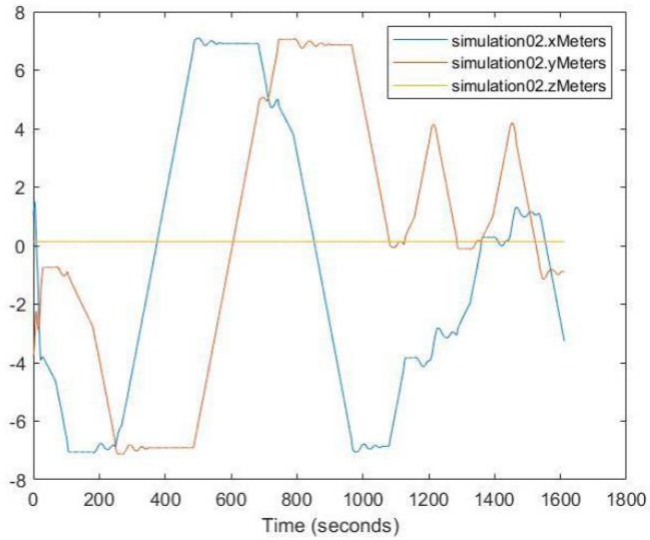


Figure 11 – Plotting data from simulation 02.



Analysing the data from Figure 12 and Figure 11, we can see that changing the threshold did not significantly improved the time spend overshooting corrections nor the near misses. It is however an improvement from the first simulation.

5 Conclusion

In this report we are presenting the implementation of a python-based robot controller for the Pioneer 3DX mobile robot that starts at a random position on our virtual environment and takes random turns until a wall is detected and continues by following the wall on a determined direction. All aspects of our proposed solution have been considered and the obtained results have been discussed. While we were successful in completing the task, the program developed was a basic solution to our task and is still in development phase, the results obtained rely heavily on the parameters inputted by us. With further testing and calculations, the program can be improved by using more accurate parameter values.

6 Limitations & Future Work

The work presented in this report completes the task given but requires further work in order to increase efficiency. In the current state our program will ran indefinitely unless manually stopped, future work can include a method for the program to stop the robot automatically once the robot reached his goal. The use of manually inputted parameters can be replaced with an algorithm (such as the ones we mentioned earlier in our report) to automatically detect obstacles and follow the pre-determined path. Extra sensors can be added or activated (if already incorporated on the pioneer robot) for better detection results.

7 References

Anderson, S.J. et al. (2010) An optimal-control-based framework for trajectory planning, threat assessment, and semi-autonomous control of passenger vehicles in hazard avoidance scenarios. *International Journal of Vehicle Autonomous Systems*, 8(2/3/4), p. 190.

Batti, H., Jabeur, C.B. and Seddik, H. (2019) Mobile Robot Obstacle Avoidance in labyrinth Environment Using Fuzzy Logic Approach. In: *2019 International Conference on Control, Automation and Diagnosis (ICCAD)*. 2019 International Conference on Control, Automation and Diagnosis (ICCAD). Grenoble, France: IEEE, pp. 1–5.

Francisco , I. *AI and Robotics*. [Online] Francisco Iacobelli's Academic Website. Available from : http://fid.cl/courses/ai-robotics/vrep-tut/pythonBubbleRob.pdf [Accessed 11/11/20].

Ghosh, S., Panigrahi, P.K. and Parhi, D.R. (2017) Analysis of FPA and BA meta-heuristic controllers for optimal path planning of mobile robot in cluttered environment. *IET Science, Measurement & Technology*, 11(7), pp. 817–828.

Lim Chee Wang, Lim Ser Yong and Ang, M.H. (2002) Hybrid of global path planning and local navigation implemented on a mobile robot in indoor environment. In: *Proceedings of the IEEE Internatinal Symposium on Intelligent Control*. International Symposium on Intelligent Control. Vancouver, BC, Canada: IEEE, pp. 821–826.

Lumelsky, V.J. and Stepanov, A.A. (1987) Path-planning strategies for a point mobile automaton moving amidst unknown obstacles of arbitrary shape. *Algorithmica*, 2(1–4), pp. 403–430.

Minguez, J. and Montano, L. (2000) Nearness diagram navigation (ND): a new real time collision avoidance approach. In: *Proceedings. 2000 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2000) (Cat. No.00CH37113)*. Proceedings. 2000 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2000) (Cat. No.00CH37113). pp. 2094–2100 vol.3.

Orozco-Rosas, U., Montiel, O. and Sepúlveda, R. (2019) Mobile robot path planning using membrane evolutionary artificial potential field. *Applied Soft Computing*, 77, pp. 236–251.

Pioneer P3-DX mobile robot. [Online] Génération Robots. Available from : https://www.generationrobots.com/en/402395-robot-mobile-pioneer-3-dx.html [Accessed 10/11/20a].

Remote API Constants. Available from : https://www.coppeliarobotics.com/helpFiles/en/remoteApiConstants.htm [Accessed 12/11/20b].

Remote API functions (Python). Available from : https://www.coppeliarobotics.com/helpFiles/en/remoteApiFunctionsPython.htm [Accessed 10/11/20c].

Remote API modus operandi. Available from : https://www.coppeliarobotics.com/helpFiles/en/remoteApiModusOperandi.htm [Accessed 12/11/20d].

Robot simulator CoppeliaSim: create, compose, simulate, any robot - Coppelia Robotics. Available from : https://www.coppeliarobotics.com/ [Accessed 10/11/20e].

Terzimehic, T. et al. (2011) Path finding simulator for mobile robot navigation. In: *2011 XXIII International Symposium on Information, Communication and Automation Technologies*. 2011 XXIII International Symposium on Information, Communication and Automation Technologies. pp. 1–6.

Webots:. Available from : https://cyberbotics.com/doc/guide/pioneer-3dx [Accessed 11/11/20f].

What is Python? Executive Summary. [Online] Python.org. Available from : https://www.python.org/doc/essays/blurb/ [Accessed 11/11/20g].

Yu, J., Su, Y. and Liao, Y. (2020) The Path Planning of Mobile Robot by Neural Networks and Hierarchical Reinforcement Learning. *Frontiers in Neurorobotics*, 14, p. 63.

8 Appendices

8.1 Appendix 1 – Python API Connection Script

```
import sim as vrep # access all the VREP elements
vrep.simxFinish(-1) # just in case, close all opened connections
clientID=vrep.simxStart('127.0.0.1', 19999, True, True, 5000, 5) # start a connection
if clientID!=-1:
    print ("Connected  to remote API server")
else:
    print("Not connected to remote  API server")
sys.exit("Could not connect")
```

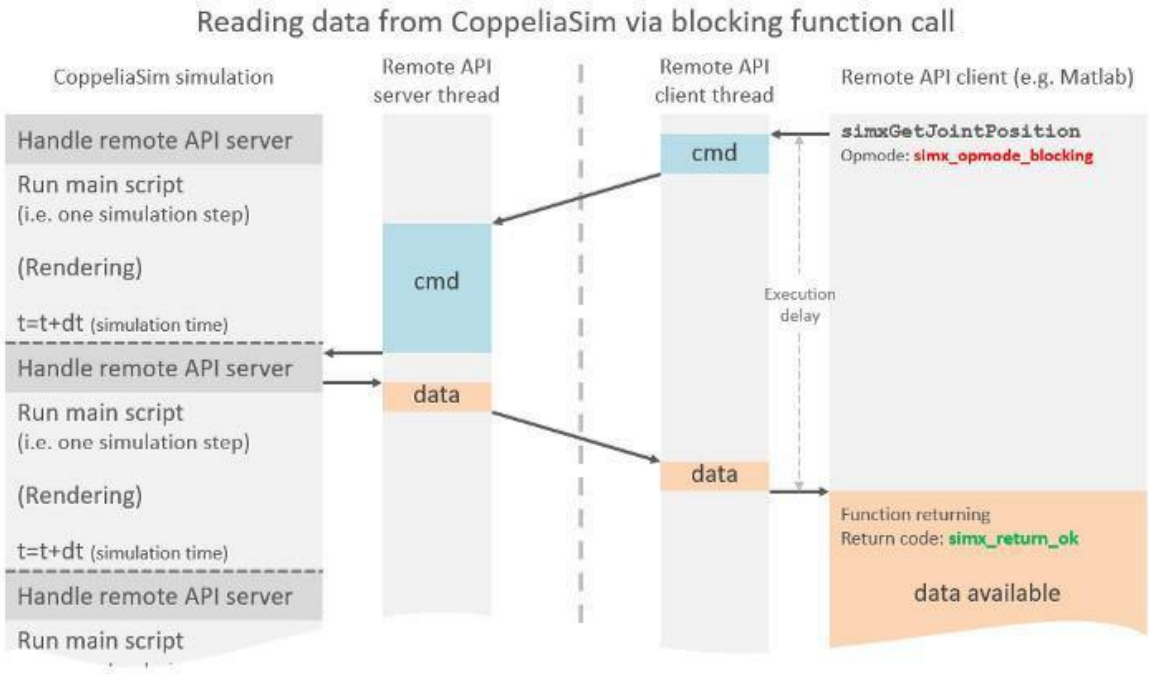
8.2 Appendix 2 – Remote Python API Functions

Full details and information available on Remote API Functions for Python and their parameters can be found at: http://www.coppeliarobotics.com/helpFiles/en/remoteApiFunctionsPython.htm.

8.3 Appendix 3 - Remote API Modus Operandi

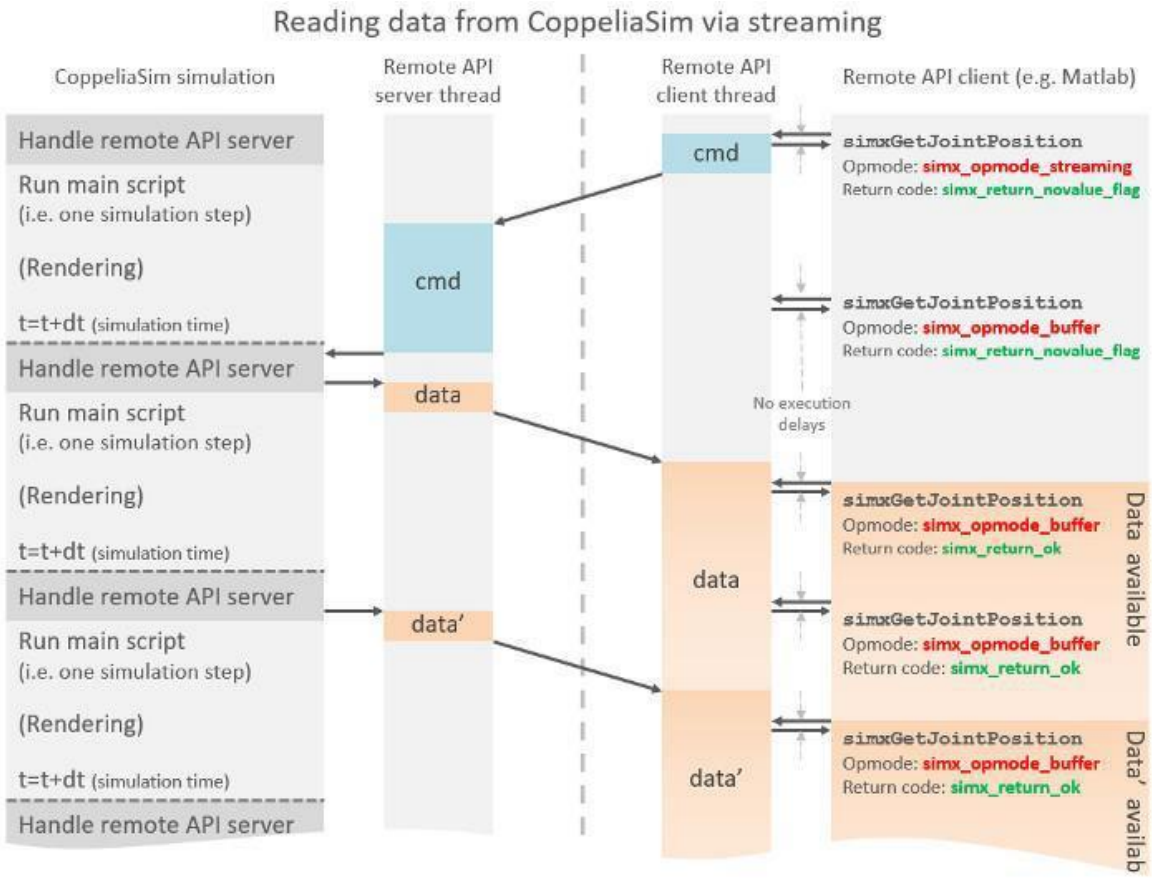
"**Blocking function calls**: a blocking function call is the naive or regular approach and is meant for situations where we cannot afford not to wait for a reply from the server" (coppeliarobotics.com, online).

Figure 13 - Blocking function calls (coppeliarobotics.com, online).



"Data streaming: the server can anticipate what type of data the client requires, for that to happen, the client has to signal this desire to the server with a "streaming" or "continuous" operation mode flag , this can be seen as a command/message subscription from the client to the server, where the server will be streaming the data to the client" (coppeliarobotics.com, online).

Figure 14 - Data Streaming function calls (coppeliarobotics.com, online).



8.4 Appendix 4 – Sensors Initialization, Alternative Method

```
sensor_h=[] #empty list for handles
sensor_val=np.array([]) #empty array for sensor measurements

#orientation of all the sensors:
sensor_loc=np.array([-PI/2, -50/180.0*PI,-30/180.0*PI,-
10/180.0*PI,10/180.0*PI,30/180.0*PI,50/180.0*PI,PI/2,PI/2,130/180.0*PI,150/180.0*PI,17
0/180.0*PI,-170/180.0*PI,-150/180.0*PI,-130/180.0*PI,-PI/2])

#for loop to retrieve sensor arrays and initiate
sensors for x in range(1,16+1):

errorCode,sensor_handle=vrep.simxGetObjectHandle(clientID,'Pioneer_p3dx_ultrasonicSens
or'+str(x),vrep.simx_opmode_oneshot_wait)
        sensor_h.append(sensor_handle) #keep list of handles

errorCode,detectionState,detectedPoint,detectedObjectHandle,detectedSurfaceNormalVecto
r=vrep.simxReadProximitySensor(clientID,sensor_handle,vrep.simx_opmode_streaming)
        sensor_val=np.append(sensor_val,np.linalg.norm(detectedPoint)) #get list of
values
```

The code above was made available by Nikolai Kummer and shows an alternative method to initialize and use all sensors, the full program can be found at: [http://34.208.13.223/VREP/04PythonTutorial/.](http://34.208.13.223/VREP/04PythonTutorial/)