Random Wanderer and Wall Follower simulating Pioneer 3-DX robot in Coppeliasim

(Lab 4 Portfolio)

Student: Arpit Sharma

P Number: P2613237

Lecturer: Uzor C

Module: Mobile Robots

Course: Intelligent Systems and Robotics

Date Due: 20 November 2020 (Extension)

ABSTRACT

The report presents an approach to addressing a wall-following problem. A Pioneer 3-DX robot consisting of 16 inbuilt ultrasonic sensors is simulated with freely steering the virtual environment and not touching or bashing into the wall. The solution pulls the extensive functionality of the CoppeliaSim (previously known as V-REP) virtual robot experimentation platform [1] to provide the simulated physical environment layer, including wall and mobile robot objects. The programming has been done in Lua and edited in Visual Studio, which manages environment perception and navigation. A final assessment is made, using different approaches for making corrections to maintain a constant distance from the wall. It is realized that with the increased number of ultrasonic sensors, the turning of the robot smoothens, as there are more sensors to estimate the odometry, and as the robot has a large number(16) of inbuilt ultrasonic sensors, it becomes one of the best choices among mobile robots for wall-following. The loop was completed in 9 minutes and 47 s with and trying to maintain a constant distance of 24.8cm with the wall. Recommendations include more refined sensor calibration and further research into proven control methods.

INTRODUCTION

The goal of this study is to develop a simulate the Pioneer P3dx robot in the CoppeliaSim simulator to perform the following tasks:

1. Initiate robot wanders with random bearing. Make robot wander randomly in the scene
2. On-wall encounter proceed to autonomously follow the wall to maintain a constant distance.

The robot is a dedicated wall follower, and the only ambiguity in the environment are the walls. This paper is organized as follows. Section 1 details the simulated physical environment, while section 2 provides details on the robot model used. Sections 3 cover the control cycle. Section 4 provides some technical detail on the solution implementation. Section 5 presents the result of testing, with section 6 closing with the final evaluation and thoughts.

SECTION 1

SIMULATED ENVIRONMENT

Physical and geometric characteristics of the simulated world, including ‘collidable’, ‘static’, and ‘responsive’ walls and the scene (Fig. 1) are modeled in CoppeliaSim. The environment is 15m by15m with the only obstacles being the 80cm high walls that include 10 inner corners, 2 outer corners, and 2 edges. The robot can start anywhere except for the small,

enclosed square. The floor is flat and smooth. No constraints like friction, air pressure, power consumption is considered in the experiment.

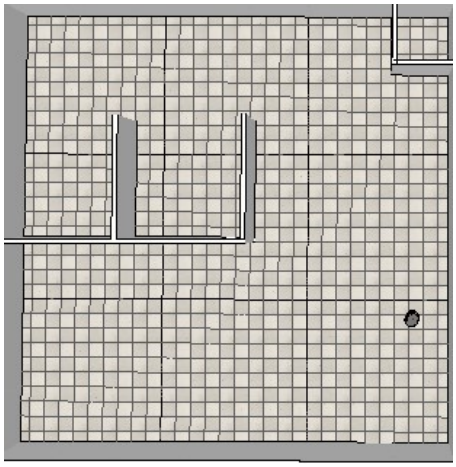

Fig.1 CoppeliaSim scene layout

Section 2

PIONEER P3DX ROBOT MODEL

Modelled in CoppeliaSim and used here is the well-known Pioneer P3DX robotics research platform [2], in basic configuration without arms or grippers. Pioneer P3DX robot (Figure 2) with 2 rear wheels, powered independently, and an omnidirectional caster wheel in the front, is modelled in the virtual environment provided by CoppeliaSim.The dimensions of the robot, shown in figure 3, is important for the calculations of the distance from the walls.

Fig.2 Pioneer P3DX robot

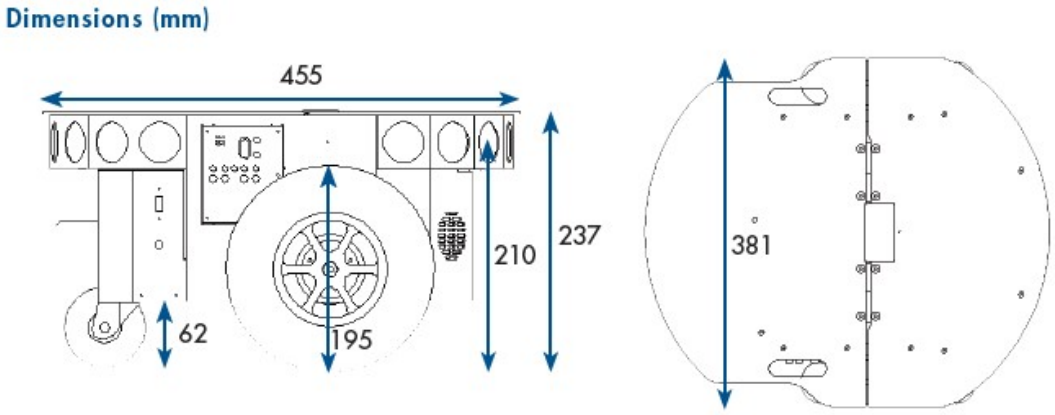

Fig.3 Dimensions of Pioneer P3DX robot

A. Actuators

The robot consists of 3 wheels in a triangle. Two larger and rear wheels are powered independently with bidirectional geared DC motors for getting the required torque, and the passive caster wheel in the front is omnidirectional. The combination makes the robot capable of rotating about its axis, moving and turning in any direction. In Coppeliasim, the motors are mimicked by the revolute joints which require input in radians (positive value for the forward direction, a negative value for reverse, and 0 for stopping).

B. Sensors

The robot consists of 16 inbuilt ultrasonic sensors, numbered as shown in figure 4. The sensors work by emitting sound waves and detecting the reflection received to calculate the distance from the reflecting body which enables the robot to detect the distance from any object around it in any direction because of sensors surrounding the robot. By default, these sensors are a cone type with max range 1m that “allows for the best and most precise modelling of most proximity sensors” [3]. When the robot is in contact with the wall, the sensor, perpendicular to the wall and facing it, reads 0.09(not 0) because of its offset position. So, the structure of the robot is important for calculating the threshold distance.

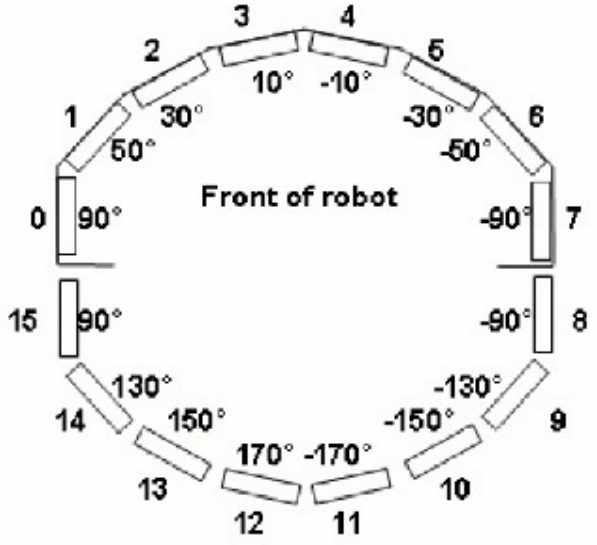

Fig.4 Numbering of ultrasonic sensors on the robot with angles

Section 3

CONTROL CYCLE

It is required for the experiment for the robot to execute four tasks in order: moving and turning(in both directions) randomly in the environment, till it detects a wall, and then, finally follow the wall making turns when needed. The principle of sense-think-act control has been used in the study.

The controller of the robot cycles through the tasks sequence, and continuously performing the 'sense' part to have the latest perception of the environment, by updating both, its internal and external state attributes. Hence, the values of all ultrasonic sensors are updated continuously. The values are converted into the distance from the objects in the environment. This information is stored in the internal state and is needed by the controller to decide to turn its revolute joints in which direction and with what speed. The absolute position of the robot is maintained by the external state.

The robot performs the 'think' part by using the state information, which can include, changing the task, upon a change in the values of the ultrasonic sensors. The internal state is updated continuously.

The last part, 'act' is implanted on the internal state of the robot, by updating the velocities of the revolute joint by values calculated in the 'think' part to rotate, turn, move in a curved or a straight path, or adjusting its alignment with the wall by turning little, left or right to maintain a constant distance. When a task is complete its status is updated, the control cycle proceeds with the next task.

Section 4

Technical details of the solution implemented

1) Sense Phase

The movement of the robot is dependent on the sensor readings. That is why sensor values are updated after every control cycle to get its current position and orientation in the environment on which it can decide the next action.

In the beginning, when the robot is in the state of 'wander randomly' if the sensors on the front side of the robot detect an obstacle(wall), the wandering state is turned off for the entire program, and the robot enters the state of 'follow wall'.

2) Move

Differential steering is used in the robot which leads to a curvature in the path if the left wheel velocity and right wheel velocities are different. So to move straight, both the velocities should be exactly equal. The velocities need to be in the range specified in the manual of the robot because if it exceeds, it can cause wear and tear, and insufficient torque for a real robot. The code is given in figure 5 below.

```
-- move straight
function move(velocity)
    sim.setJointTargetVelocity(motorLeft, velocity)
    sim.setJointTargetVelocity(motorRight, velocity)
end
```

Fig.5 Code for moving straight

3) Turning about its axis

When the left and wheel velocities are of equal magnitude, but opposite in direction, the robot starts performing pure rotation, thanks to the omnidirectional caster wheel in the front. The code for it is given in the figure 6 below.it is required for aligning with the wall and in case, an inner corner is detected.

```
-- function for making a turn using differential system as the motors are independently powered
function turn(turnVelocity)
    sim.setJointTargetVelocity(motorLeft, -turnVelocity)
    sim.setJointTargetVelocity(motorRight, turnVelocity)
end
```

Fig.6 Code for rotating about its own axis

4) Making an arc

The robot turns an arc when the left and wheel velocities are of different magnitude but in direction. The robot makes an arc, the direction and curvature depending on the ratio of their velocities, thanks to the omnidirectional caster wheel in the front. The code for it is given in the figure below.it is needed in the case, an outer corner is detected, or a wall edge is detected. In Lua, the code looks like given in figure 7.

```
-- for making an arc , speeds of left and right wheels must be different
function turnArc(leftMotorVelocity, rightMotorVelocity)
    sim.setJointTargetVelocity(motorLeft, leftMotorVelocity)
    sim.setJointTargetVelocity(motorRight, rightMotorVelocity)
    delay(arcTime)
end
```

Fig.7 Code for making an arc

5) Introducing a delay in the program

As an indefinite while loop cannot be used in Lua because the script gets blocked and the program crashes. For setting a timer, to make an arc, or turning 90 degrees, simulation time is stored in a variable before starting a task and hence, in the task, the simulation time is continuously monitored, and hence, the difference in time is calculated between the current time and last stored time. Till the difference is lesser than the delay, that is needed, all flags are turned to block execution. this gives a perfect delay for the time, that is required in seconds. The code is given in figure 8 below.

```
function delay(time)
    if (timeSetFlag == 0) then
        runningTime = time
        timeSetFlag = 1
        isDelayRunning = true

        lastTime = sim.getSimulationTime()
    -- --print("Timer Started")
    end

    if sim.getSimulationTime() - lastTime > runningTime then
        -- --print("Timer Stopped")
        timeSetFlag = 0
        isDelayRunning = false
    else
    end
end
```

Fig.8 Code for introducing a timer in the program

6) Random Wandering

Two approaches can be used for random wandering: one without a timer and the other with a timer. Both use a random number generator.

First Approach: A random number is generated, and the movement of the robot is decided purely on the range of that number. This method gives the path of the robot, a random curvature every time, (like shown in the figure), and when the left turn and right turn come out to be consecutive commands, the robot appears stopped. The code and the output are shown in figure 9 and figure 10, respectively.

```
function wandering()
    -- generate a random number between 1 and 100
    randomNumber = math.random2(1, 100)
    -- please change seed for trying different random patterns
    -- math.randomseed2(24)
    -- --print(randomNumber)
    if randomNumber > 90 then
        --turn left
        turn(velocity)
    elseif randomNumber > 70 then
        --turn right
        turn(-velocity)
    else
        -- move straight
        move(velocity / 2)
    end
end
```

Fig.9 Code for the first approach for randomly wandering

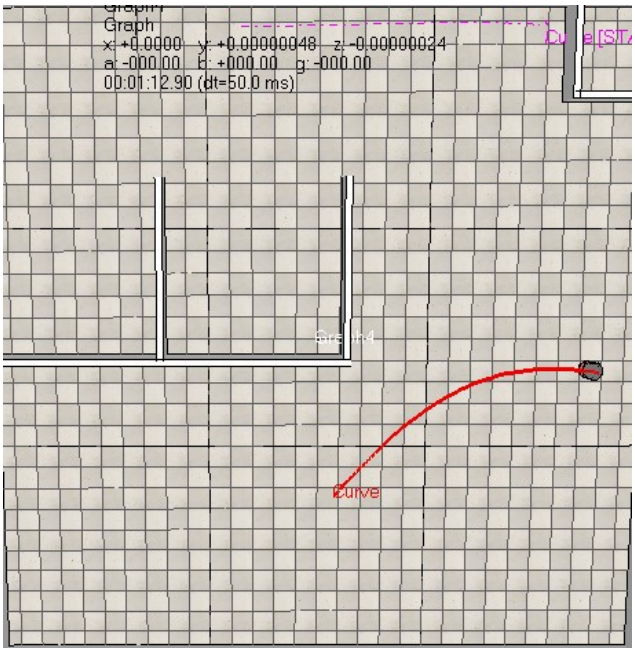

Fig.10 Output for the first approach for randomly wandering resulting in a random curved path

Second Approach: In this approach, instead of the random number just determining the movement, it also determines the period for which that movement should continue. So, instead of curvature, as in the previous case, we get a robot moving and rotating in random directions for a random period as shown in figure 12, and the code is given in figure 11.

```
function wandering()
    if isDelayRunning == false then
        timeSetFlag = 0
        -- generate a random number between 1 and 10
        -- please change seed for trying different random patterns
        -- math.randomseed2(24)
        randomNumber = math.random2(1, 10)
        print(randomNumber)
        if randomNumber == 2 then
            --turn left
            turn(velocity)
        elseif randomNumber == 3 then
            --turn right
            turn(-velocity)
        else
            -- move straight
            move(velocity / 2)
        end
        delay(randomNumber)
        isDelayRunning = true
    else
        randomNumber = math.random2(1, 5)
        delay(randomNumber)
    end
end
```

Fig. 11 Code for the second approach for randomly wandering

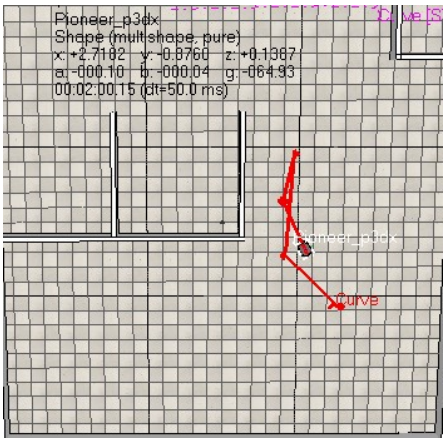

Fig. 12 Output for the second approach for randomly wandering

7) Wall Follow

Once the robot detects a wall, it enters and stays in the 'follow wall' state for the rest of the program. So, as the four sensors in the front (numbered 2, 3, 4, and 5 in the specification) are detecting a wall, it continues to rotate till every one of them reads a value lesser than the threshold distance, so that robot is somewhat aligned with the wall, and the rest of the alignment job is done by the 'turnCorrections' function which continuously keeps cycling so that the robot maintains a constant distance from the wall. So, after this happens, the robot starts moving immediately in a straight path. As there are three situations, which demand turning (90° inner corner, 90° outer corner, and the 180° wall edges). The code for the 'followWall' function is given below in figure 7.

```
function followWall()
    read_sensors()
    -- if timer running, dont do anything else
    if isDelayRunning == true then
        delay(runningTime)
    else
        -- read all sensors ad print readings for testing purposes
        -- readSensorsAndPrintValues()
        if wallOver() or uTurn() then
            turnArc(2 * velocity, velocity)
        elseif checkObstacleInFrontSide() then
            turn90(turningVelocity)
        elseif isDelayRunning == false then
            move(velocity)
            -- detect[7]>0 added because wallOver and turnCorrection interfere otherwise
            if detect[7] > 0 then
                -- function for making little turns to adjust itself along the wall at a constant distance
                turnCorrection()
            end
        end
    end
end
```

Fig.13 The code for the followWall function

8) TurnCorrection

Once the robot starts following a wall, to keep it aligned and maintain a constant distance error, turn correction is introduced, The difference between the sensor(s) on the front side and the sensor(s) on the rear side of the same side as the wall is calculated. As the difference will be positive if the right wall is being followed, and the front side is leaning away from the wall and vice-versa. So, when the difference is positive, turns slightly right, and left,

otherwise. The problem with this approach is that, with this simple approach, the robot starts oscillating between turning slightly left and slightly right. That is why the use of the concept of 'hysteresis' is used. The action is taken only if the difference is below turnCorrectionTolerance(Lower hysteresis condition), or above turnCorrectionTolerance(upper hysteresis condition), that is if it lies in the shaded grey area in the graph in figure 8, no correction is made. otherwise, little adjustments are made by making minute turns to maintain a constant distance from the wall. 0 is the set point for the hysteresis.

The difference is taken between the 7th and 8th sensors (numbering according to figure 4), but, to get much more accuracy, the 6th and 9th sensors can be used, as their difference will be more per radian, but the robot will start making corrections most of the time instead of moving forward. So, it is a trade-off between accuracy and speed here. The other thing of concern in this approach is that if the turns for corrections are bigger to an extent, that in one control cycle, the robot skips the grey region, then, the robot gets stuck in an infinite loop of turning left and right. So, making turns, smaller is a necessary part of this approach. The values are dependent on the turning velocity and can be calculated from the rotation step size in one control cycle in the console of Coppeliasim. Figure 9 shows the actual code.

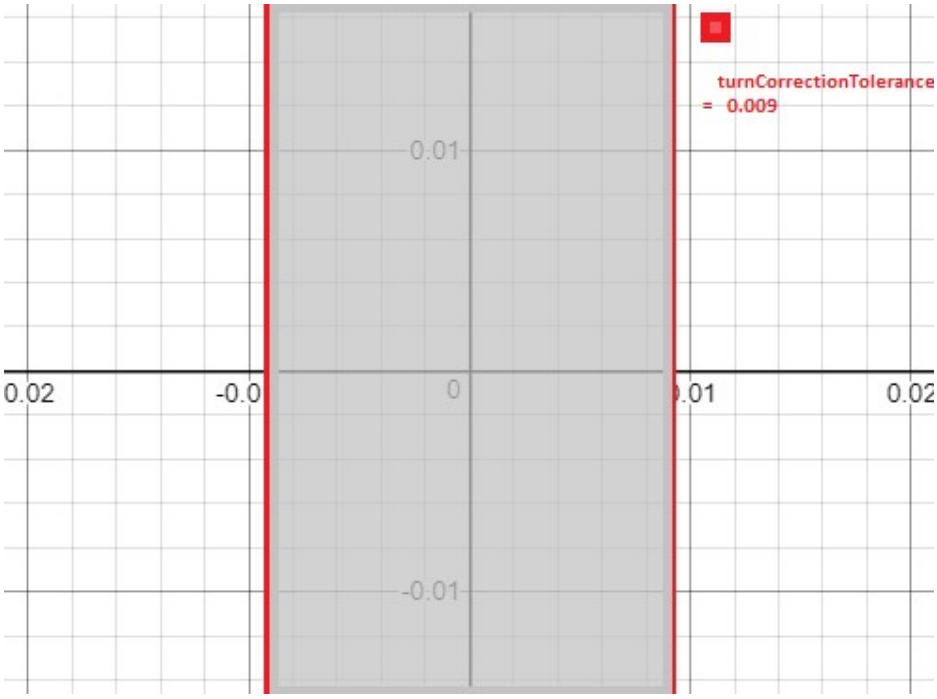

Fig.14 Using hysteresis for maintaining a constant distance from the wall and avoiding oscillations

```
function turnCorrection()
    -- using hysterisis, if the  diff is between -turnCorrectionTolerance and +turnCorrectionTolerance, do nothing
    -- otherwise, if diff is between below, or beyond  the turnCorrectionTolerance,
    --  make adjustments by making little turns to maintain a constant distance from the wall
    -- if the robot is about to move away from the wall, diff becomes negative, hence it is made to turn right and vice-versa
    local diff = detect[8] - detect[9]

    read_sensors()
     if detect[8] > 0 and detect[9] > 0 then
         if (diff > turnCorrectionTolerance) then
            -- --print("TOO CLOSE/////////")
            --print("Correction-turn left")
            sim.setJointTargetVelocity(motorLeft, -correctionVelocity)
            sim.setJointTargetVelocity(motorRight, correctionVelocity)
        elseif (diff < -turnCorrectionTolerance) then
            -- --print("TOO FAR.........")
            --print("Correction-turn right")
            sim.setJointTargetVelocity(motorLeft, correctionVelocity)
            sim.setJointTargetVelocity(motorRight, -correctionVelocity)
        end
    end
end
```

Fig.15 The code for the Corrections to maintain a constant distance from the wall

9) Algorithm

The algorithm for a control cycle of the main program and the ‘follow wall’ state is given in figure 16 and figure 17 respectively. This cycle keeps looping throughout the simulation.

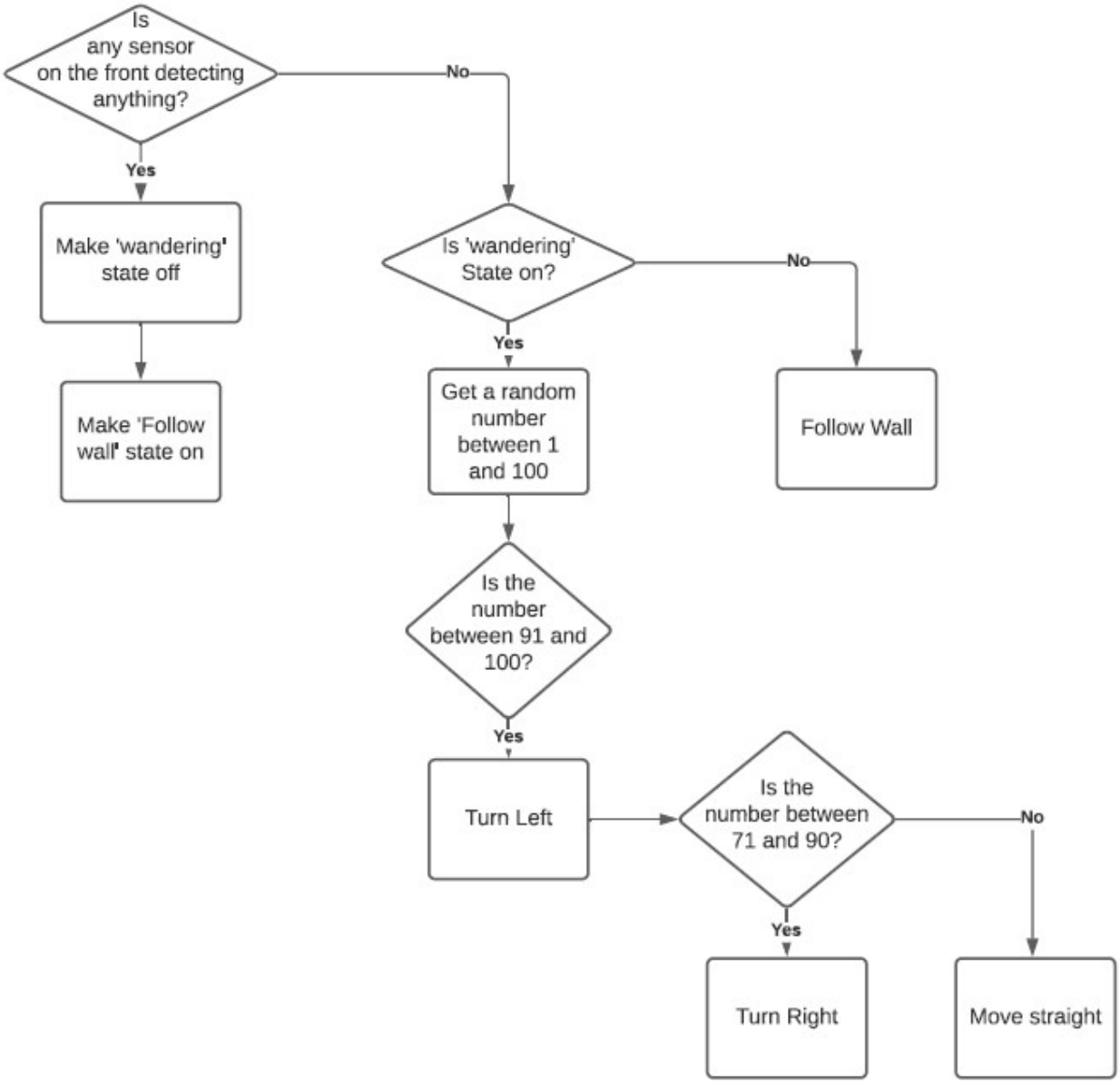

Fig.16 Algorithm for a single control cycle of the main program

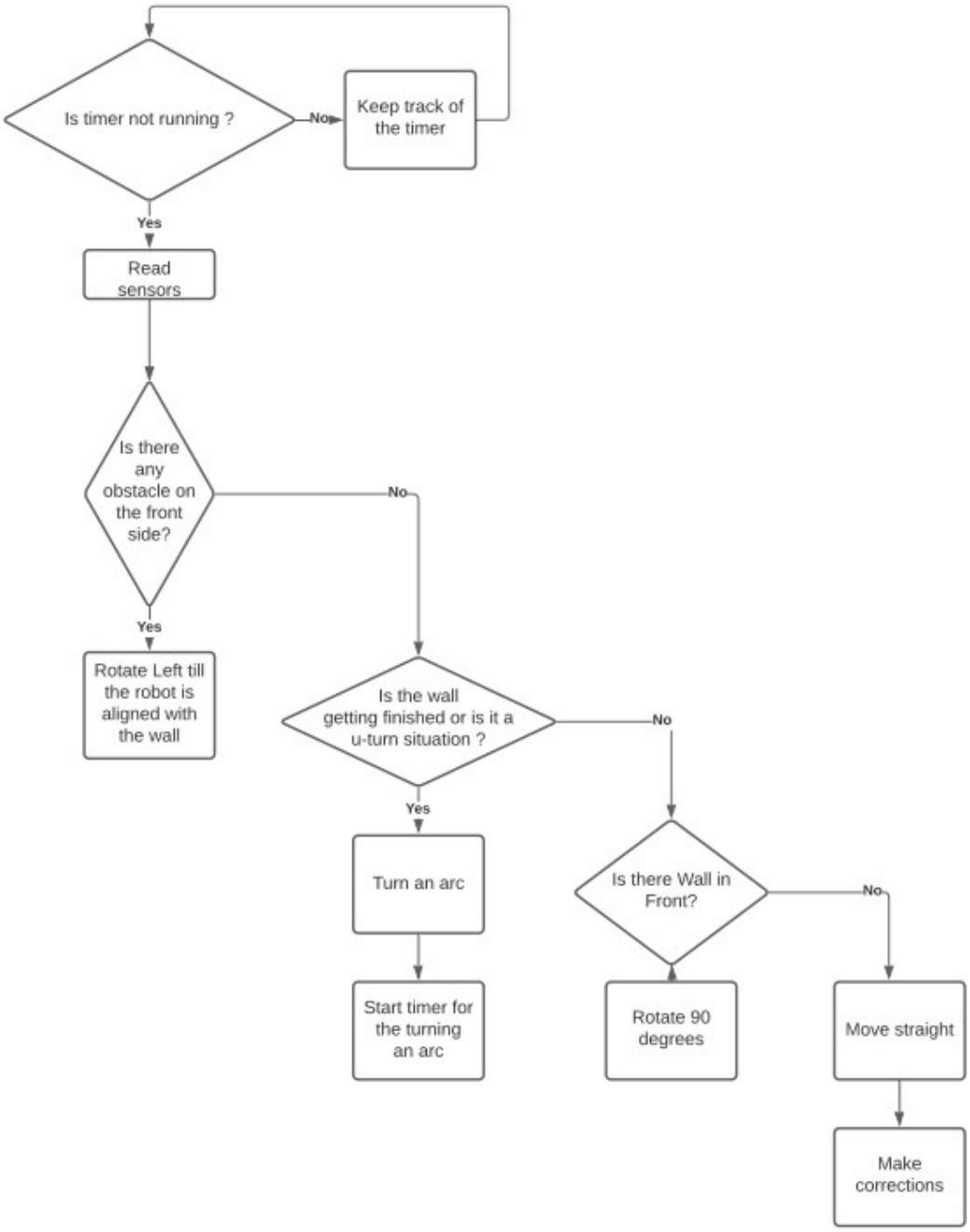

Fig.17 Algorithm for a single control cycle of the follow wall state

Section 5

Results and Conclusions

Initially, the values read by sensors to drive the robot with the differential motor yielded unwanted results, and large oscillations were seen because of high deviations from the correct path, which often resulted in the robot straying even further from the path, or the robot getting stuck especially around inside corners.

The hardest problem to tackle was in the case of 90 degrees turn. The robot would take 90 degrees turn towards left and then, do corrections to align with the wall again by turning right, hence, reaching where it started, and getting stuck in these two situations.

The problem was solved by printing sensor values to the console, making the robot stationary, and placing the robot in both problem causing positions, noting down the sensor values and then, coming up with such a combination of sensors and tolerances, which would not allow any two tasks(especially consecutive tasks) to interfere with each other. So, the formulations of the combinations required the use of the multi-sensor approach and large number of iterations.

The other problem was in making corrections for aligning the locomotive along the wall. In the beginning, the magnitude of the corrections was so high that the robot would start oscillating between turning slightly left and slightly right, hence, getting stuck. But, finally, this was solved using the concept of ‘hysteresis’, allowing a region in the middle, where no corrections take place, and the robot moves straight.

So, finally after trying every required combination of multiple sensors, and tolerance values, to minimize task interference between each other, and efficient task execution, the output was obtained that is shown in figure 19 and the position graph for it is shown in figure 20.

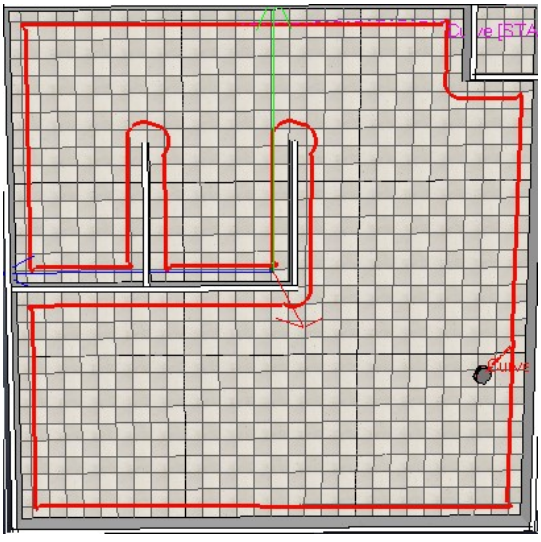

Fig.19 Output

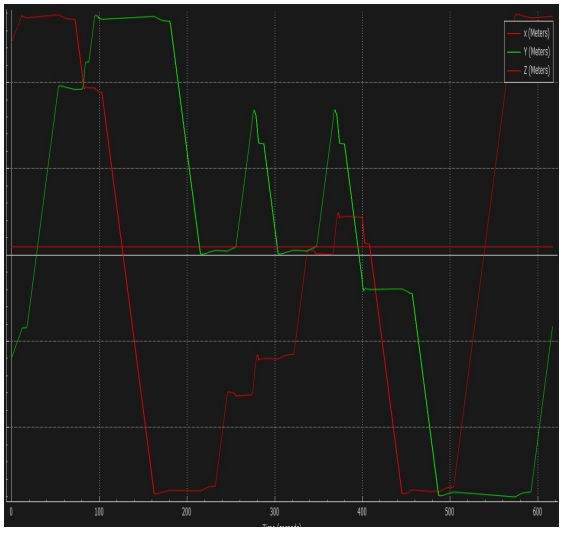

Fig.20 Output graph for the wall following state

Section 6

Final Thoughts

The efficiency of the robot can be increased by trying new approaches,trying more combinations of sensors, but ultimately, there is a trade-off between accuracy and speed, as more accuracy means, more corrections in much smaller steps, which results in more control cycles for the corrections, resulting in low speed and more time.

Bibliography

[1] VREP, “V-REP Virtual Robot Experimentation Platform Home Page,” [Online . Available: http:/'www.coppeliarobotics.com.

[2] P. P3DX, “Pioneer P3DX Rev A Data Sheet.” Adept Mobile Robots, [Online]. Available:
https: www. Generationrobots.com/media/Pioneer3DX-P3DX-Rev A. pdf.

[3] http:/www.coppeliarobotics.com, “Proximity sensor types and mode of operation,” [Online]. Available:
https://www.coppeliarobotics.com/helpFiles/en/proximitySensors.htm