**A Mobile Robot Controller Featuring Exploration and Navigation Behaviours**

*Adam Hubble (P17175774)*

***Abstract – Commonly, mobile robots employ random wandering behaviours for exploration and surveillance operations, meanwhile edge following as an evasive behaviour, is typically renowned for its application in navigation and object avoidance. The aims of this report are to explore the ways in which random wandering and edge following behaviours can be architected as a controller, for a mobile robot that exists within a structured environment.***

***Keywords – random wandering; exploration; edge following; evasive; navigation; object avoidance; controller; mobile robot; structured environment***

I. Introduction

Aligned with the layout and contents of the environment provided and the application of the Pioneer P3-DX [1] mobile robot, it was anticipated for the robot to initially explore or wander the environment randomly; up until an object, or wall, would be detected using its onboard sensory capabilities. Functionally, wandering was expected to feature forward traversal and rotary motions in sequence, which with randomised metrics, would determine the direction of angular offset and the durations of each movement pattern for the robot.

***Available in Appendix A***

Upon detecting a wall, the task understood for the robots undertaking was to actuate edge following behaviours for a given detected edge of an object, by rendering sensory data. This was assumed for avoiding collisions and allowing the robot to navigate itself around the accessible sides of said walls, at a set distance, that it would attempt to maintain over time. The expected mode of traversal whilst exhibiting said behaviour, was forwards, for a given direction that the robot initially intercepts the wall.

In extension to the robot’s edge following behaviours aforementioned and given the connected nature of the walls in the environment, it was further expected for the robot to follow the edges of the walls continually, where the robot would remain to exhibit behaviours specific to edge following but not random wandering. Understood as the inhibition of desired behaviour [2], it was acknowledged that both behaviours would contribute to the robots “levels of competence” [2]; but each of the behaviours would have to subsume one another, and allow the robot to exhibit behaviours “appropriate to the task it has to perform” [3]. This was figured for the robot’s expectation to transition between tasks, which subsequently inspired the subsumption architecture [3] of the robot’s controller and its inclusion of states and subsidiary states to accommodate control.

***Available in Appendix B***

Simplified, a mobile robot controller that supports the Pioneer P3-DX to wander randomly and follow the edges of objects it detects, interchangeably, was to be devised, assessed, and enhanced.

1. *Software implementation*

As the robot controller’s client, CoppeliaSim [4] was independently nominated for the robot controller’s implementation and application, due to its prior correspondence in the ‘Mobile Robotics’ undergraduate module. Whereby existing knowledge and applications of robot controllers in Lua, as CoppeliaSim’s native programming language, was more viable as opposed to using unfamiliar languages employed by the remote application programming interface(s) (API’s) recommended.

1. *Pioneer P3-DX overview*

As the mobile robot subjected to the tasks mentioned, understanding its composition and capabilities were mandatory for compiling a controller of the given specification.

Universally, the Pioneer P3-DX robot employs a two-motor differential drive and provides an array of Sound Navigation and Ranging (SONAR) sensors for its front faces; which as exteroceptive-active sensors [5], enables the robot to emit and intercept ultrasounds as a means of object detection and rendering sensory data.

***Available in Appendix C***

Featured within CoppeliaSim, the Pioneer P3-DX robot occupies sixteen SONAR sensors, that present a positional and angular arrangement, which provide a detectable potential that surrounds the entirety of the robot’s exterior; this was initially presumed useful for detecting objects that present obtuse motions for the robot.

***Available in Appendix D***

For the implementation of the controller, all SONAR sensors are utilized to engage the behavioural expectations of the robot.

***Available in Appendix E***

II. Random wandering

For addressing the implementation of random wandering, the robot was required to exhibit the following behaviours:

* Forward traversal
* Angular traversal

Constituting to the randomness of the behaviours listed, both required the use of randomised metrics to determine a distance for forward traversal, a turning direction, and an angular offset to accumulate. Given the dual functionality of the controller proposed, it was assumed that the requirement for the metrics listed would better the robot for “target searching” through a series of “continuous movements” [7]; which when simplified, is the sequence of forward and sideward motions. In focus of the environment that the robot is subjected to, random wandering aims to appropriate the robot’s capability of exploration to discover walls, as the robots targeted interest.

Given the distinction between the random wandering behaviours, it was sensible to componentize forward and angular traversal motions into separate subsidiary states, as each motion was imagined to be composed up of unique functionality. From an architectural standpoint, code base expansions were easier to implement and interpret over the course of the behaviour’s development. To determine the active subsidiary state of the random wandering behaviour, the robot can simply identify as rotating or not, from the implementation of a Boolean variable acting as a state determiner, alongside a series of conditional statements to regulate the functionality invocated.

For transitioning between each subsidiary state, the robot is required to accumulate the randomised targets for distance traversed and angular offset accumulated. By default, forward traversal has higher precedence and is thereby invoked before angular traversal at the start of program runtime; this was configured alongside with the robots wheel velocity, to enable the robot to explore its environment faster (comparatively) and to decrease the likelihood of collisions with objects, when transitioning from edge following behaviours. To default forward traversal as the initial random wandering behaviour, the state determining Boolean variable referred to prior, is reset, for the first frame when transitioned to the edge following state.

***Available in Appendix F***

1. *Forward traversal*

Aligned with exploration, forward traversal was configured with a randomised range of ‘0.1’ and ‘0.5’ metres, which is considered to be a range of short traversal distances; this range was configured to enable the robots traversal to be increasingly varied, as the robots angular traversal frequency would increase exponentially. Increased variation as the result of “shorter travelled paths”, assumed that the robot was to “provide more area coverage” [8] of its environment, for any given duration. Although a range exceeding a metre could be implemented and rendered as being functional for the focus environment, the range configured was more desirable for exploration purposes.

Assuming a traversal distance for the given range is generated, the robot’s motors for both wheels are actuated with a velocity of ‘0.2’ metres per second (default), allowing the robot to traverse forwards. Following its launch, the robot will continue to traverse forwards until the generated distance has been accumulated and acknowledged, by updating the state of the Boolean variable mentioned previously. The distance that the robot travels is calculated by finding the difference between the magnitude of the robots previous and current positions for the ‘X’ and ‘Y’ axes only, given that the environment is levelled; the difference in position calculated is then appended to a float variable, representing the distance that the robot has travelled over time.

1. *Angular traversal*

Regarding the configuration of angular traversal, a randomised range was once again populated but for the range of angular offset that the robot would accumulate, from rotating over time. For this motion, the range of ‘30’ and ‘90’ degrees was provided, which dictates that the robot cannot exceed an offset of an acute angle, at any given period. This was purposed for enabling the robot to turn away from the current area being explored, whilst preventing it from immediately turning back into the area explored prior; thus promoting the robot to explore “unknown surrounding space” [7].

***Available in Appendix G***

Additional to a given angular offset, the turning direction for the angular offset to be accumulated in was also required; said direction is utilized to determine the leftward or rightward motion of traversal for the robot. Also proposed as a randomised metric, the range ‘1’ to ‘100’ was nominated, which when applying the modulo operator, a direction can be determined based upon the generated numbers relation to being even (right), or odd (left). Supplying the range presented was purposed for enhancing the randomisation potential, as opposed to being binomial.

Assuming an angular offset and turning direction are generated for the ranges presented, then for the robots turning direction, the motor situated on the corresponding side of the robot decelerates to ‘0.1’ (default velocity halved) metres per second, whereas the opposing motor accelerates to ‘0.3’ (default velocity halved and added) metres per second. Said velocities enable the robot to sustain a cumulative velocity of ‘0.2’ metres per second, which enables the robots’ movements to be seamless whilst navigable. Through the alternation of wheel velocities presented, the robot gradually rotates whilst traversing forwards to maximise exploration efficiency, as opposed to rotating around its own axes, for the current position it resides at. In regard to the angular offset that the robot accumulates and parallel to calculating distance travelled, angular offset is calculated by accumulating the adjustments in the robots heading indicator; if the orientation is negative, is it negated, which enables the difference between the robots current and previous heading to be determined and interpreted by relevant conditional statements. Upon the difference being determined, it is then appended to a separate float variable, representing the angular offset that the robot has accumulated over time.

***Available in Appendix H***

III. Edge following

Regarding the implementation of edge following, the subsequent behaviours were required to be shown by the robot in simulation:

* Forward traversal
* Distance maintenance

Dissimilar to wandering behaviours, edge following requires the robot to apply metrics that are precise and calculated, as opposed to being totally randomised. As expected for the robots tasking, the robot would need to find a means for reliably transitioning from wandering behaviours, to edge following behaviours, using its sensory readings. Upon the transition expiring, the robot should then follow the detected wall with a forward mode of traversal, whilst adjusting its orientation to achieve a given set-point from the wall. It is necessary for the robot to rotate left and right overtime to achieve this, in result of initial offset from the set-point, overturning and SONAR sensor imprecision; this is a known error, fundamental to “ultrasonic wave propagation” [9], which in focus of the Pioneer P3-DX’s sensor arrangement, can be assumed as a factor of the numerous “active transmitters of acoustic waves of similar frequencies” that it employs. For the environment that the robot is subjected to, the robot has been configured to follow adjacent faces and perpendicular arrangements of walls, that present a range of angular adjustment for the robot; this allows the robot to avoid and navigate autonomously, whilst continuing to exhibit edge following behaviours as required.

***Available in Appendix I***

Given the difference in edge following behaviours, similar to the random wandering state, each behaviour was componentised into subsidiary states; where the states accommodate edge following for the left and right sides of the robot, as well as for its transition into either side of edge following. This organisation was necessary to detach the alignment functionality used for transitioning, from the distance adjustment functionality used to edge follow; discriminating between edge following for the robots left side and right side was sensible for the robustness of the controller, in the event the functional requirements would later differ, and for the robots motor velocities to be appropriated to the side that a wall is detected. To determine the active subsidiary state of the edge following behaviour, the robot can be identified as edge following left or right, or alternatively as transitioning, through the implementation of a series of Boolean variables and conditional statements; transitioning is further subsidised into transition left or right, which is calculated and used to determine the direction that best situates seamlessness from wandering, and is made relative to the orientations of the robot and detected wall(s).

To transition between each of the subsidiary states, the robot requires to detect a wall or finish following the edge of a currently detected wall; to initiate this transition and to invocate the edge following behaviours, all SONAR sensors that the robot operates are surveyed for readings, per frame passed. This enables the robot to react to stimuli instantaneously, thus preventing collisions. For the given state, there is no default behaviour assigned, as the subsidiary state that the robot invocates is determined situationally, which depends on the orientation of the robot, relative to the orientation of the object(s) it detects and the quantity and index of the sensor(s) detecting the object(s); this enables the robot to appropriate the transition from random wandering to edge following behaviours efficiently.

***Available in Appendix J***

1. *Wandering to following*

To achieve state transitioning from random wandering to edge following, for every frame that passes, all sixteen SONAR sensors are surveyed for object detection readings. By providing an undetectable range (as a float variable) that the robot can validate detections within, enables the robot to enter given proximities of objects before invocating edge following behaviours; this inflicts restraints on the set-point that the robot can be assigned to maintain over time, however. If any of the sensors return validated readings, the robot is tasked to edge follow; subsequently, random wandering behaviours cannot be inhibited (not executed).

For determining the subsidiary state of edge following to invocate, a criterion composed up of a series of ‘if-else’ statements is referred to. In focus of this criterion, to edge follow for the robots left side, it is required that the robots left-most front facing sensor (indexed ‘1’) detects a wall, whilst its corresponding front-left most facing sensor (indexed ’3’) does not. Said configuration enables the robot to cater for potential collisions ahead, given the forward mode of traversal that the robot actuates; therefore, it is not necessary for the robots back facing sensors to be accounted for whilst edge following. For this configuration, the side-most sensor is factored so that the distance maintenance behaviour can be invoked, given its adjacent relation to the wall that the robot is parallel to and following. Respectively, to edge follow for the robots’ right side, the same criteria pertains but for the robots right-most front facing and front-right most facing sensors alternatively (indexed ‘8’ and ‘6’).

***Available in Appendix K***

Meanwhile, to invocate the alignment (transition) behaviours of the edge following state, it is required that a wall is detected, but not on either side-most sensor (indexed ‘1’ and ‘8’) of the robots front face. This is necessary to avoid collisions preliminarily, then followed by the robot’s ability to align parallel with a detected wall, where it can then transition into edge following for the corresponding side. Otherwise, the robot can directly transition into edge following for a given side-most sensor. To note, the back facing sensors considered for the criteria are significant to the robot’s capability of following adjacent faces and perpendicular arrangements of walls, as mentioned prior; where the side-most and front-most sensors can no longer detect a wall. For the criterion explored, each condition is applied to control the activeness of each subsidiary state, through alternating the values of the Boolean variables representing them.

***Available in Appendix L***

1. *Transitioning*

Relating to the configuration of the transition subsidiary state, the activeness of the further subsidised states: transition left and transition right, are determined by calculating the direction of the object that is situated closest to the robot, or via randomisation. Randomised direction is defaulted to in the situation where the robot detects an object at equal lengths, from multiple sensors simultaneously; typically, this applies when the robot is adjacent to a wall (sensors indexed ‘4’ and ‘5’), and is generated with a binomial range for simplicity. Alternatively, for calculating the direction that an object resides closest in, the readings for both sides of sensors are accumulated as separate float variables and are then compared by size using a series of conditional statements; said statements are implemented to control and alternate the values of the subsidized states and the resulting behaviours exhibited. This method was also nominated for its simplicity.

Excluding randomisation, for objects detected in the robots front faces, the aligning direction is configured to be the opposing direction of where the closest object resides; this enables the robot to become parallel with the wall for a smaller incline or angular offset, which is better suited to the robots fluidity of movement and efficiency for exploration and navigation. Whereas, for objects detected in the robots back faces only, the aligning direction is configured to be identical to the direction of where the closest object resides. Using the back facing sensors to achieve this behaviour enables the robot to adjust its orientation, so that the robot maintains a constant detection with a given wall, until either front side-most sensor can sense the proceeding adjacent face of it; otherwise, for a perpendicular arrangement of walls, the subsequent wall will be detected. Inevitably, this technique is effective, given that it prevents random wandering from being invoked and allows the robot to continue navigating around the environment, uninterrupted.

Regarding the actuation of this behaviour, given the example that the subsidized state: transition left, is active, the robots left motor velocity accelerates to ‘0.2’ metres per second, meanwhile its right motor velocity accelerates to ‘0.2’ metres per second, negated. This enables the Pioneer P3-DX to rotate around its own axes for its current position, to ensure that the robot does not crash into walls with an interior perpendicular arrangement. Furthermore, the technique ensures that the robot remains within the detectable range of a given wall, as well as the set-point, at which the robot attempts to maintain from a wall, whilst explicitly edge following walls with an exterior perpendicular arrangement. When visualising the turning motion configured, the robot would turn right, away from the wall detected on its left side; thus, the robot and wall align with a parallel relation, where the wall detected on its left side can then be followed. For transitioning right, the robots left motor velocity accelerates to ‘0.2’ metres per second, negated, while the right motor velocity accelerates to ‘0.2’ metres per second.

***Available in Appendix M***

1. *Distance maintenance*

To adhere to the requirement of maintaining a set distance from a wall, a proportional-integral-derivative (PID) controller was integrated, to approach the set distance by controlling the robots motor velocities progressively and steadily, over time, whilst either edge following left or edge following right subsidiary states are active. Given a PID controllers known “smoothness, performance and accuracy” [10] and support with “autonomous control systems and navigation systems”, it was assumed appropriate for use with a differential-drive robot, equipped with “motor drives” that could be directly affected.

For the implementation of the PID controller, as the robot’s motor control source, for both subsidiary states when edge following, the difference (error) in distance between the targeted set-point and robot, is calculated using the distance returned by the front side-most sensor, corresponding to the active subsidiary state of edge following. For every frame passed whilst edge following, the calculated error is stored into an array variable for the current count (index) of error generated; as the array size, ‘10’ was configured to consider the last ‘10’ errors generated by the robot, which is considered as a sensible sample for calculating the integral gain. To calculate PID, the current, previous, accumulated and sum of errors were required, to affect the velocity of the motor corresponding to the side of edge following active. Each of said metrics are stored as float variables, which loosely represent the proportional, derivative, and integral gains to the motor’s velocity, prior to their calculation. Upon PID being calculated, it is then appended to the default velocity (‘0.2’ metres per second), for the motor that corresponds to the side that the robot is edge following for; meanwhile the opposing motor maintains a velocity of ‘0.2’ metres per second to maintain a forward mode of traversal. From the alternation of the motor’s velocity, the robot can accelerate and decelerate aligned with the calculated error, to near and retain the set-point of ‘0.25’ metres, as a series of subtle oscillatory motions.

Regarding the gain values configured for the PID controller, the proportional gain was settled at ‘7’, integral ‘4’ and derivative ‘0.2’; for all of the gain values listed, collectively, they are used to enhance the rate at which the error between the robots distance from the wall and set-point, is marginalised. However, values exceeding the configuration presented were observed to affect the robot’s ability to align with a wall, upon the edge following state being entered initially. Where the robot could be described as performing a series of oscillatory motions, that are considerable in size. This behaviour hindered the seamlessness and efficiency between the transition from random wandering to edge following states, and so the apparent trade-off between correction enhancement over shorter durations, compared to fluidity in movement and desired behaviour exhibition, was studied. Evidently, desired behaviour was favoured, as sizable oscillatory motions within enclosed spaces of environments, could potentialize often occurrences of collision; whilst the robot still maintains to correct its distance from a wall, regardless.

IV. Controller Architecture

So that a “basic control system can be established” [2] for managing the transition between and activeness of the random wandering and edge following states, and their subsidiaries (as the controllers “levels of competence”), Rodney Brooks’ “subsumption architecture” was adopted. Resulting from this adoption, the controller features a series of finite-state machines (FSM’s), which “models the behaviours as states, transitions and associated actions” [11]. This enables each behaviour or “module” [2] to subsume one another, in the form of “disabling specific behaviours where their activity at a particular time or circumstance is undesirable”. For the context of the controller configured, behaviours are inhibited as opposed to being suppressed, which only permits one behaviour to be active at any period, and poses positive implications on computational performance, comparatively. Furthermore, given the incremental development process that the architecture supports, each of the behaviours could be implemented in isolation, which assured the robustness of the controller, that further enabled the development process to be hastened.

***Available in Appendix N***

As referred to throughout the prior sections of this document, state control is managed by a series of Boolean variables and ‘if-else’ statements, representing the FSM’s, major and subsidiary. Given the binomial values of Boolean variables, it was sensible to represent each major and subsidiary state, with one of said variables. Which when combined with the ‘if-else’ statements, could be applied to govern state transition, when the value of the variable(s) alternate(s); thus, allowing the robots actuation to be specific to the task determined. Meanwhile, for the purpose of robustness, random wandering and edge following behaviours are partitioned into separate functions. This allows the functionality of the only active major state: random wandering or edge following, to be invocated, whereas the opposing states functionality is ignored from execution. If additions were to be made to this controller in future periods, the process would be effortless, given the structure presented.

V. Testing

When compiling the final configurations for both random wandering and edge following states of the controller, a series of black-box test regimes were conducted, for enabling the robot to exhibit the behaviours required for each state, at their greatest capacity. For the testing procedure nominated, black-box testing was appropriated for its support for visual observations and numerical analysis of the robot’s actuation and was resultingly deemed sensible for identifying adjustments to the controller’s configuration. Regarding the approach to testing, the configurations for each behaviour was exploratory tested [12], given that the behaviours the robot exhibits can only be enhanced, and not rendered absolute; this is assumed by the inaccuracy of the robots SONAR reception, as previously mentioned, which remains a persistent hinderance to calculations concerning sensory data.

1. *Random wandering*

As the basis for testing random wandering behaviours, the randomised ranges provided for generating a forward traversal distance, angular traversal direction and offset, for which the robot accumulates, were adjusted for optimising its wandering capability. As previously described, the targeted outcome for the wandering configuration was to promote the exploration of unexplored areas, through preventing the robot from turning back into the area explored prior. Meanwhile, regarding forward traversal, it was desired for the robot to traverse shorter distances, for increasing its frequency of angular traversal, over time.

For the approach to testing random wandering behaviours, a series of black-box test cases were conducted, given the behaviours observational standpoint; where randomness cannot be measured numerically. However, supporting the results of the test cases and for determining the desirability of the actuation exhibited, a series of console output was configured to determine the subsidiary random wandering state, its target metric and the accumulation for the metric that the robot has achieved, from a programmatical stance. For the test regime of the random wandering behaviours, see ***Appendix O***.

1. *Edge following*

Relating to the edge following state, for edge following behaviours to be exhibited, a set-point and maximum distance value considered as the distance that the robot should and can be from a detected wall, had to be appropriated in direct relation to the robots undetectable distance of objects. This was mandatory for enabling the robot to follow the edge of a wall, detected by the robot initially and during its attempt to adjust its position to the given distance (set-point) from the wall, over time. The maximum distance from a wall, should cater for the mere oscillations in the robot’s movement patten, for its continual adjustment; otherwise, random wandering behaviours will be invoked in result of the robot overturning from its adjustment to the set-point. Meanwhile, the secondary aim for these characteristics was to enable the robot to be situated relatively close to the walls it follows, yet with a distance that enables manoeuvring to be collision-free; this was desired of the robot, for the scenario that it is subjected to other environments that feature enclosed spaces. Like the test approach used to better random wandering behaviours, the robot’s ability to initiate and maintain a detection with a wall, required its actuation to be observed, to indicate the active behaviour being exhibited; therefore the black-box method was selected for its suitability. Furthermore, console output was also configured for the behaviours, to determine and acknowledge the sensors with the closest detection to objects, the active major and subsidiary state for the behaviours invoked and the current oscillatory motion of the robot (turning inwards or turning outwards).

Meanwhile, for marginalising the robot’s adjustment to the set-point over time, the PID gain values that the robot employs within the PID controller, were fine-tuned for enhancing the rate at which the adjustment to calculated error occurs. In determination of this rate, root mean squared error (RMSE) could be calculated to numerically represent the difference between the current position of the robot and the set-point, for a given number of errors generated whilst the robot is following the edge of a wall. As mentioned previously, the configuration of the gain values were required to appropriate the transition from the random wandering behaviours, for preventing sizable oscillatory motions from occurring, which potentialize often collisions and hinder the seamlessness of said transition. By contrast, the rate of adjustment should also be enhanced for the robot to near the set-point sooner, which has the purpose of optimising the accuracy and efficiency of the adjustment actuated; enhancing this rate has proportional implications on the fluidity of the transition between behaviours, however. To identify the impact of alternating the gain values, the robot’s actuation required observation again, and so black-box testing was referred to once more. For the RMSE calculated, console output was also configured to output the value, for the purpose of numerically acknowledging the rate at which the error depreciates. Additional to this preparation, a graph object was also created, for plotting the RMSE value of every frame passed whilst edge following (time graph); visually representing RMSE was deemed more practical, for comprehending the robots rate of adjustment to the set-point. Meanwhile, another graph object was configured, for plotting the errors generated over time (time graph); collectively, these visual representations were used to correlate relations with error and RMSE in pursuit of time, which has enabled the final configuration of the gain values to be optimised. For the test regime of edge following behaviours, see ***Appendix P***.

***Available in appendix Q***

VI. Results

1. *Random wandering*

In direct correspondence to the results of the test cases produced, it is inevitable that the controller enables the robot to exhibit forward and angular traversal behaviours, for the random wandering state; each with randomised ranges to control the distance, angular offset and direction assigned to the relevant behaviour. Without doubt, the wandering behaviours of the controller can be considered random. Based upon the ranges subjected to testing, it was observed that a forward traversal distance ranging between ‘0.1’ and ‘0.5’ metres and an angular offset ranging between ‘30’ and ‘90’ degrees, were best adapted to exhibiting randomness and desired exploration patterns. Whereas ranges populated below and above this configuration, were identified as predominant causes for the robot exploring areas that it resided in prior. This was mostly apparent due to the large significance in angle that the robot accumulates, when exhibiting angular traversal behaviours, in conjunction with the insignificant distances assigned for forward traversal. Opposingly, the robot would merely offset from its previous orientation when assigned insignificant angles to accumulate, whilst traversing angularly. Thus, causing the robot to navigate around the walls ahead of it more frequently, in comparison to wandering the environment. This behaviour could also be correlated with longer distances travelled, for its forward mode of traversal.

1. *Edge following*

Relating to the results produced by the test cases concerning the edge following state, it is known that the controller facilitates edge following behaviours, where both left and right sides of the robot can be used to follow the edge of a wall, at a given distance parallel to it. Said behaviours are also proven to account for the walls presenting a perpendicular arrangement, that feature interior and exterior angles (relative to the detectable side of the walls), as well as for the adjacent faces of walls in isolation.

In focus of the configuration decided for the relation between the set-point, maximum and undetectable distances, it is believed to be optimal for the robot exhibiting desired behaviours, when transitioning between the random wandering and edge following states, and for the preservation of edge following behaviours over time. Gathered by the test cases conducted, for an undetectable distance of ‘0.35’ metres, a set-point of ‘0.25’ metres and maximum distance from a wall considered for edge following, of ‘0.275’ metres, it was determined that the configuration best promotes the prevention of sizable oscillatory motions (potential collisions avoided) when transitioning between behaviours, whilst maintaining a relatively close distance to a wall. This is evidenced by the contrasting nature of observations, obtained for the other configurations tested.

Meanwhile, relating to the configuration determined for the PID controller gain values, it is considered optimal, when considering the compromise between the fluidity of behaviour transition, and rate of adjustment to the set-point that the robot actuates over time. As determined by the series of test cases conducted, a proportional gain of ‘7’, integral of ‘4’ and derivative of ‘0.2’, promotes the actuation of the robot when transitioning from random wandering to edge following behaviours, to be more fluid, whilst enabling it to maintain a detection with the edge of the wall being followed and also being relatively efficient, for adjusting to the errors generated. Whereas gains exceeding the configuration presented, were often found to hinder the fluidity of behavioural transition, by promoting sizable oscillatory motions; however, the robot’s rate of adjustment to the set-point, becomes significantly sharper. Whilst gain values less significant than the configuration presented, would hinder the robot’s ability to maintain a detection with the edge of a wall being followed, thus meaning that the robot’s rate of adjustment to the set-point, depreciates also.

VII. Conclusion

In conclusion of implementing a controller for the Pioneer P3-DX mobile robot, capable of actuating random wandering and edge following behaviours, learning developments have been mostly focused on the behaviours constituting edge following. This is specific to my understanding and implementation of behaviours that satisfy the robots requirement to follow walls around the environment, as enabling the robot to follow the edges of walls that either form perpendicular arrangements, or are isolated, was the most challenging task required of the controller, in terms of implementation. Meanwhile, in relation to random wandering, my understanding of odometry has advanced, from the principles incorporated within the implementation of its behaviours.

For the configuration of the controller submitted, even finer tuning of the PID controller gains could promote an improved, yet compatible rate of distance adjustment for the robot, however I believe that such tuning would not be necessary, nor substantial to the difference in behaviours exhibited. Such tuning could also be offered to random wandering behaviours, for a similar magnitude of benefit to the controller. Moreover, granted the inclusion of an avoidance algorithm as future implementation, the transition between random wandering and edge following could be intermediated and made more effective, when compared to the implementation of the transition behaviours.

References

[1] Génération Robots (2020) *Pioneer P3-DX mobile robot*. [Online] Génération Robots. Available from: https://www.generationrobots.com/en/402395-robot-mobile-pioneer-3-dx.html [Accessed: 08/11/20].

[2] SIMPSON, J. and JCOBSEN, C.L. and JADUD, M.C. (2006) Mobile Robot Control the Subsumption Architecture and occam-pi. In: The 29th Communicating Process Architectures Conference, CPA 2006, organised under the auspices of WoTUG and the Napier University, Edinburgh. Berlin: ResearchGate, pp. 225-236.

[3] TOAL, D. and FLANAGAN, C. and JONES, C. and STRUNZ, B. (1995) Subsumption Architecture for the Control of Robots. [Online] Available from: https://www.researchgate.net/publication/244321030\_Subsumption\_Architecture\_for\_the\_Control\_of\_Robots [Accessed: 08/11/20].

[4] Coppelia Robotics (2020) *CoppeliaSim from the creators of V-REP*. [Online] Coppelia Robotics. Available from: https://www.coppeliarobotics.com [Accessed: 08/11/20].

[5] APARNA, K. and UMESH, B. (2013) Overview of Sensors for Robotics. *International Journal of Engineering Research and Technology*. [Online] 2 (Issue 3/ March). Available from: https://www.ijert.org/overview-of-sensors-for-robotics [Accessed: 08/11/20].

[6] KIM, K. and KIM, M. and CHONG, N.Y. (2010) RFID based collision-free robot docking in cluttered environment. In: *Progress in Electromagnetics Research.* Massachusetts: EWM Publishing, pp. 199-218.

[7] XIE, Y. and YAN, X. and CHEN, M. and CAI, J. and TANG, Y. (2019) An autonomous exploration algorithm using environment-robot interacted traversability analysis. In: 2019 IEEE/ RSJ International Conference on Intelligent Robots and Systems (IROS)*, Macau.* New York: IEEE, pp. 4885-4890.

[8] EL-HUSSIENY, H. and ASSAL, S. and ABDELLATIF, M. (2013) Improved Sensor-Based Mobile Robot Exploration of Novel Environments. In: *The Sixth International Conference on Intelligent Computing and Information Systems (ICICIS 2013), Cairo.* Berlin: ResearchGate, pp. 43-49.

[9] MAJCHZRAK, J. and MICHALSKI, M. and WICZYNSKI, G. (2009) Distance Estimation with a Long-Range Ultrasonic Sensor System. *IEEE Sensors Journal*. [Online] 9 (7). Available from: https://www.researchgate.net/publication/224492926\_Distance\_Estimation\_With\_a\_Long-Range\_Ultrasonic\_Sensor\_System [Accessed: 11/11/20].

[10] LEKKALA, K. and MITTAL, V.K. (2014) PID Controlled 2D Precision Robot. [Online] Available from: https://www.researchgate.net/publication/288424236\_PID\_controlled\_2D\_precision\_robot [Accessed: 11/11/20].

[11] ZHANG, P. (2008) Application Software for Industrial Control. In: ZHANG, P. (ed) *Industrial Control Technology.* Norwich: William Andrew Publishing, pp. 569-673.

[12] Agile Alliance (2020) *Exploratory testing*. [Online] Agile Alliance. Available from: https://www.agilealliance.org/glossary/exploratory-testing/#q=~(infinite~false~filters~(postType~(~'page~'post~'aa\_book~'aa\_event\_session~'aa\_experience\_report~'aa\_glossary~'aa\_research\_paper~'aa\_video)~tags~(~'exploratory\*20testing))~searchTerm~'~sort~false~sortDirection~'asc~page~1) [Accessed: 11/11/20].

Appendices

***Appendix A:***

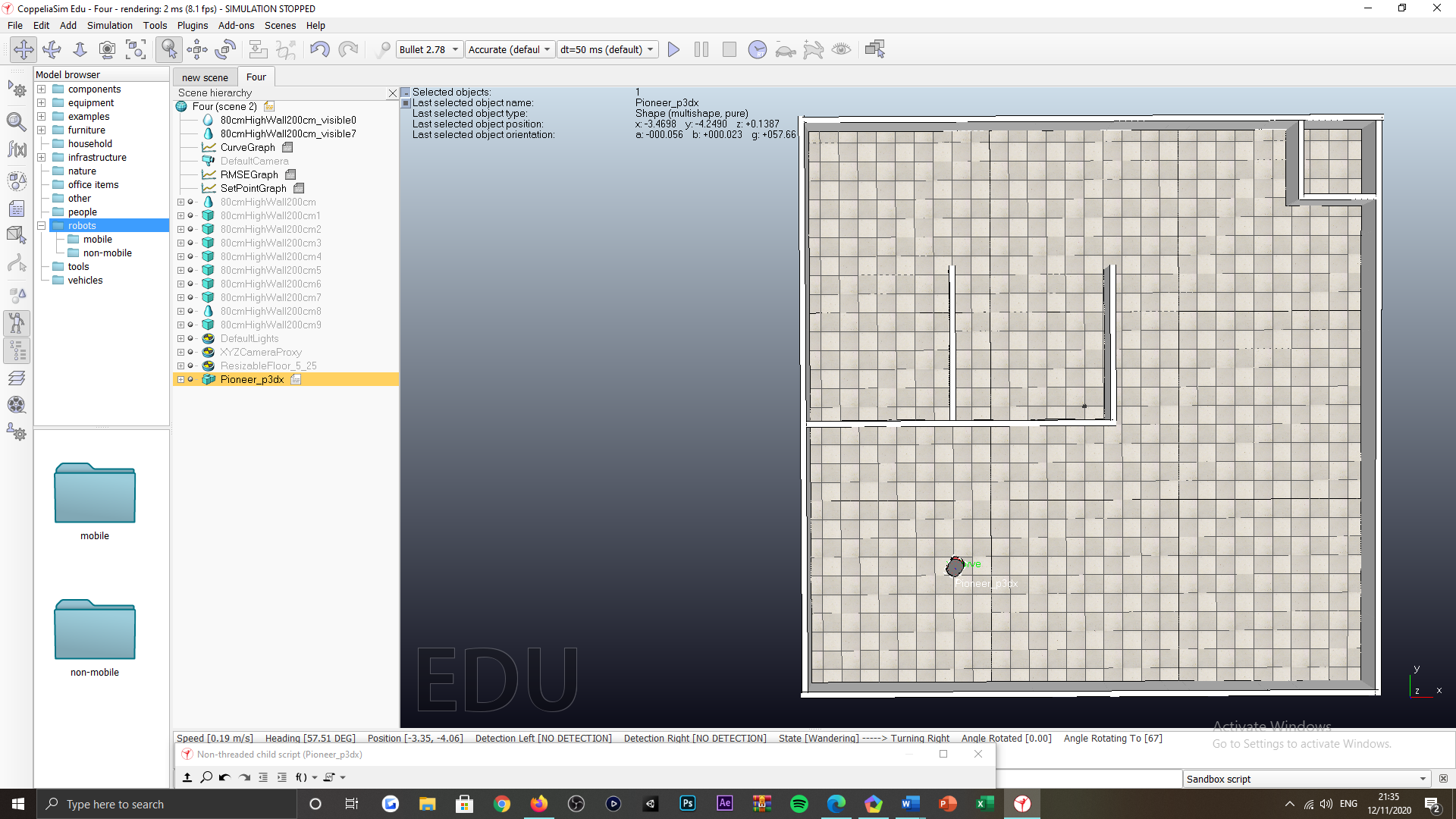


Figure : Pioneer P3-DX subjected environment.

***Appendix B:***

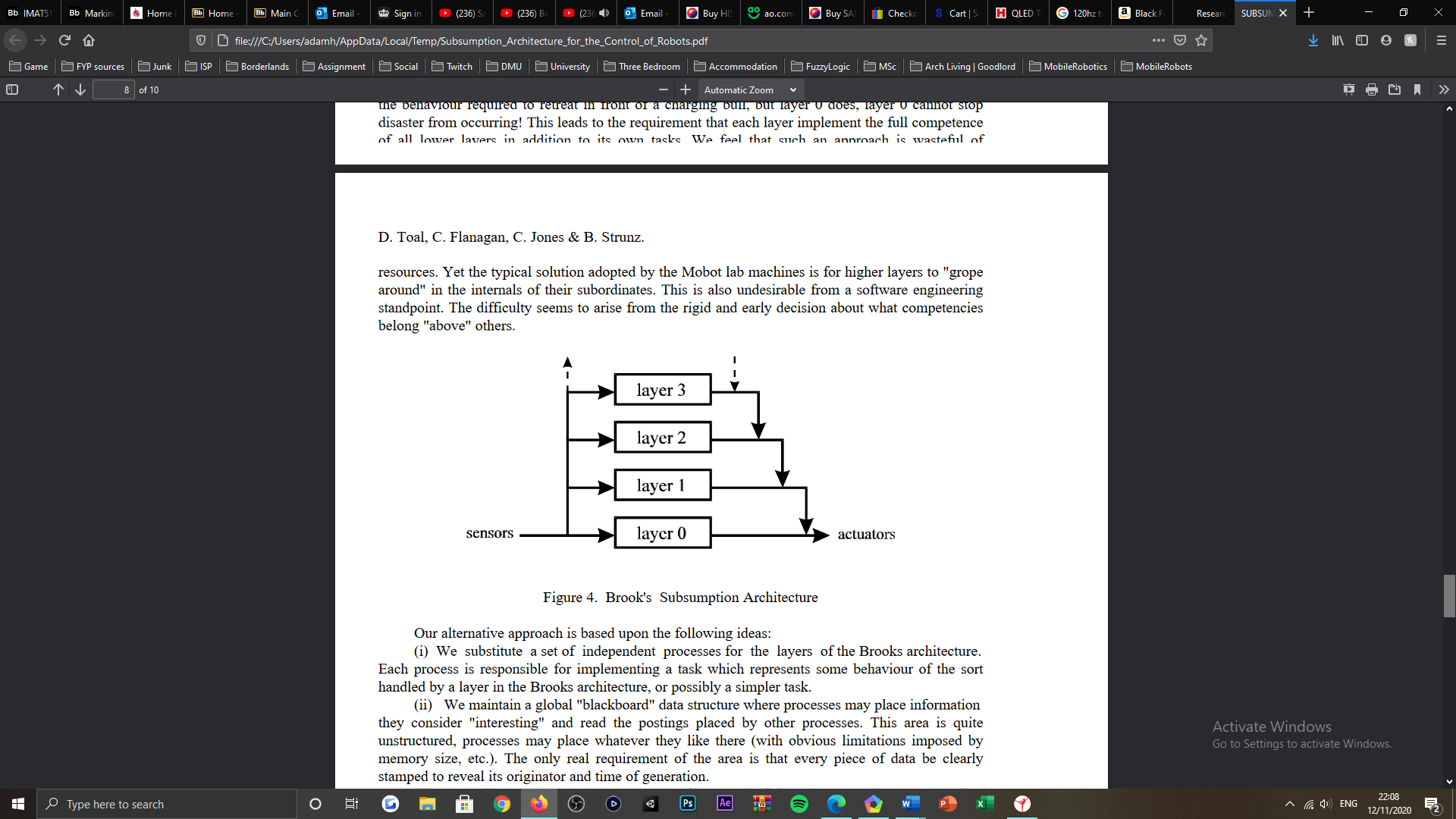


Figure : Rodney Brooks' subsumption architecture [3].

***Appendix C:***



Figure : Pioneer P3-DX mobile robot [1].

***Appendix D:***



Figure : Obtuse motions presented to the Pioneer P3-DX mobile robot, when actuating edge following behaviours.

***Appendix E:***

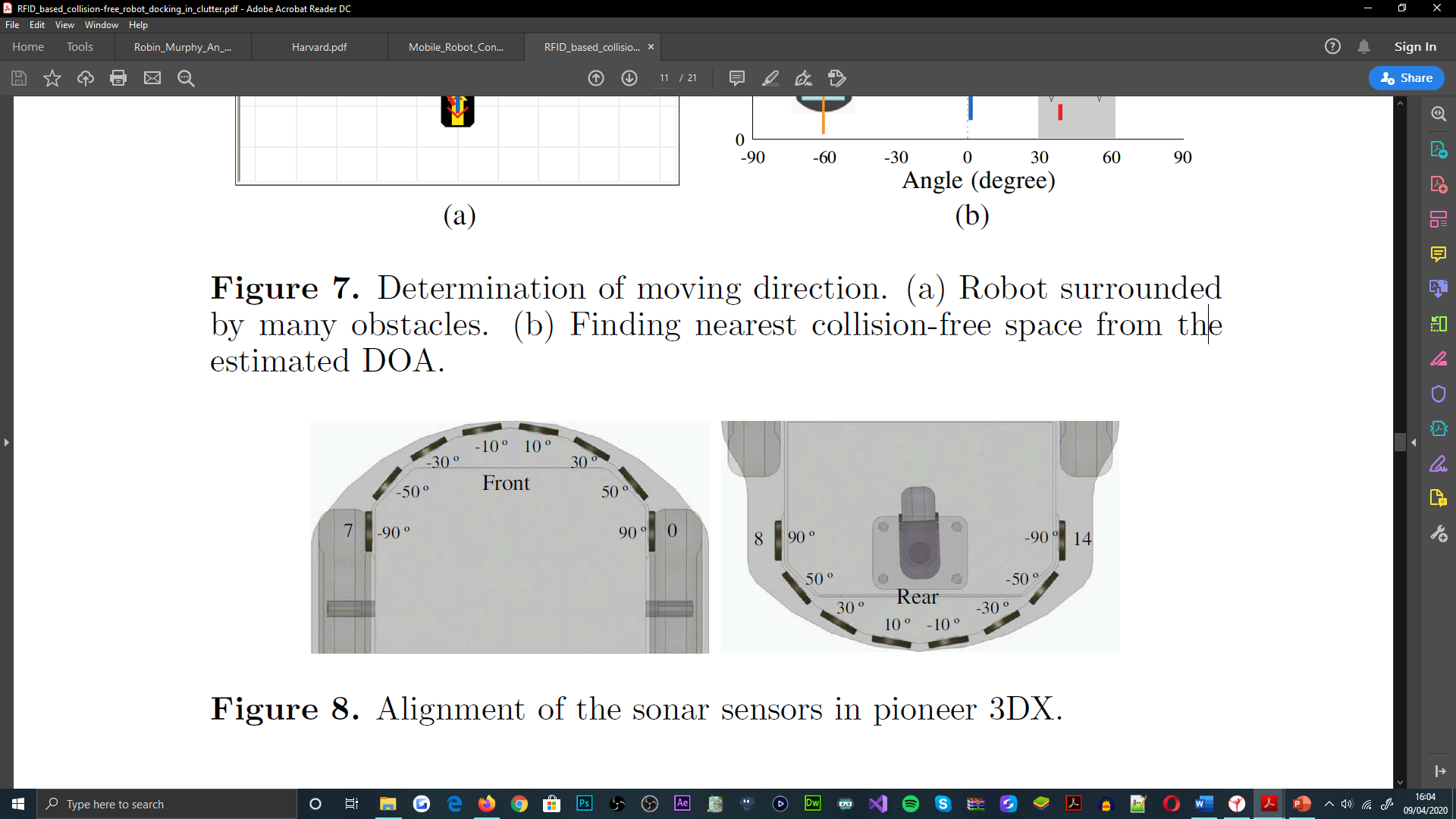


Figure : Pioneer P3-DX SONAR sensor arrangement [6].

***Appendix F:***

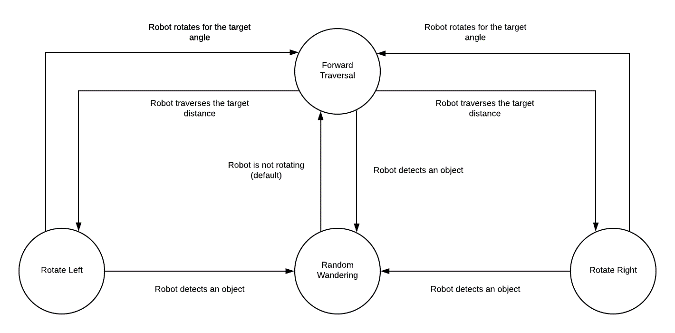


Figure : Random wandering, finite-state machine (FSM).

***Appendix G:***

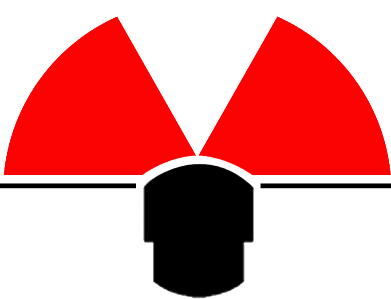


Figure : Random wandering, angular traversal offset potential, for the range ‘30’ to ‘90’ degrees (red zones).

***Appendix H:***

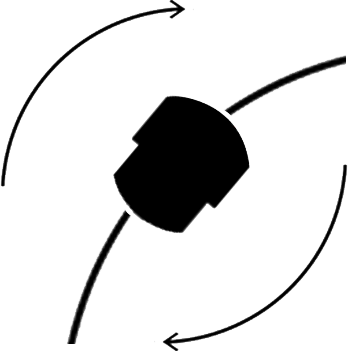


Figure : Pioneer P3-DX, angular traversal visualisation.

***Appendix I:***

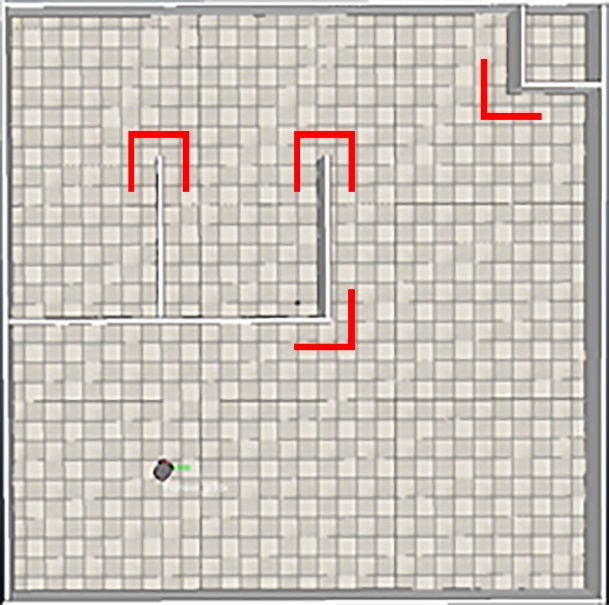


Figure : Exampling walls with perpendicular arrangements and adjacent faces exposed, for the subjected environment.

***Appendix J:***

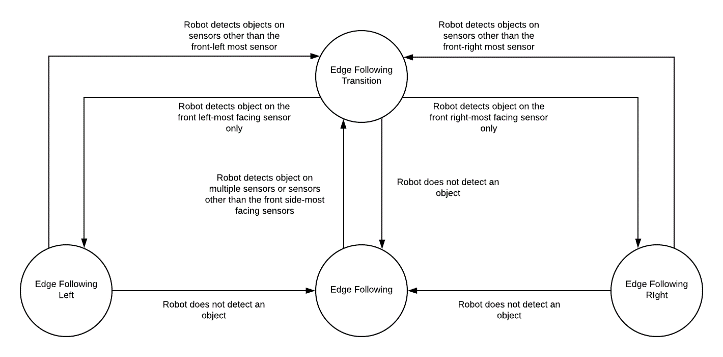


Figure : Edge following, finite-state machine (FSM).

***Appendix K:***

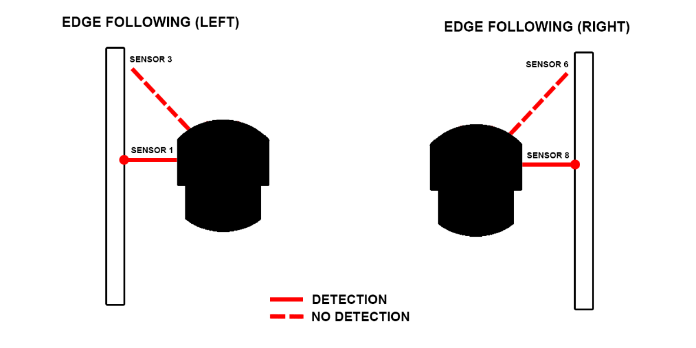


Figure : Pioneer P3-DX SONAR sensors, used to condition the invocation of either edge following subsidiary state.

***Appendix L:***

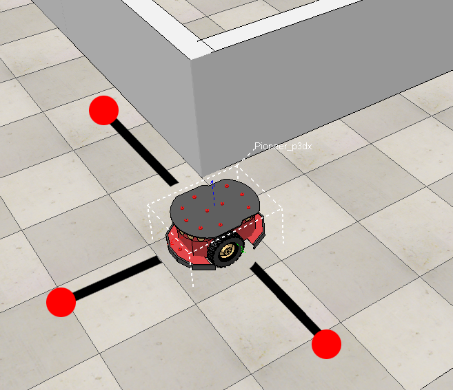


Figure : Pioneer P3-DX front, left and right-most sensors do not detect an object, at a perpendicular arrangement of walls in the subjected environment.

***Appendix M:***

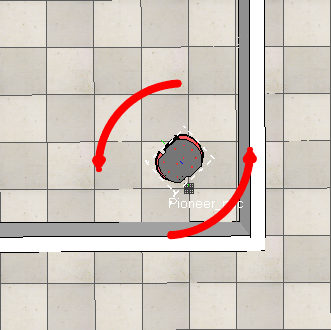


Figure : Edge following, transition left subsidiary state, actuation visualisation (arrows representing the acceleration (right) and deceleration (left) of the motor components of the Pioneer P3-DX.

***Appendix N:***

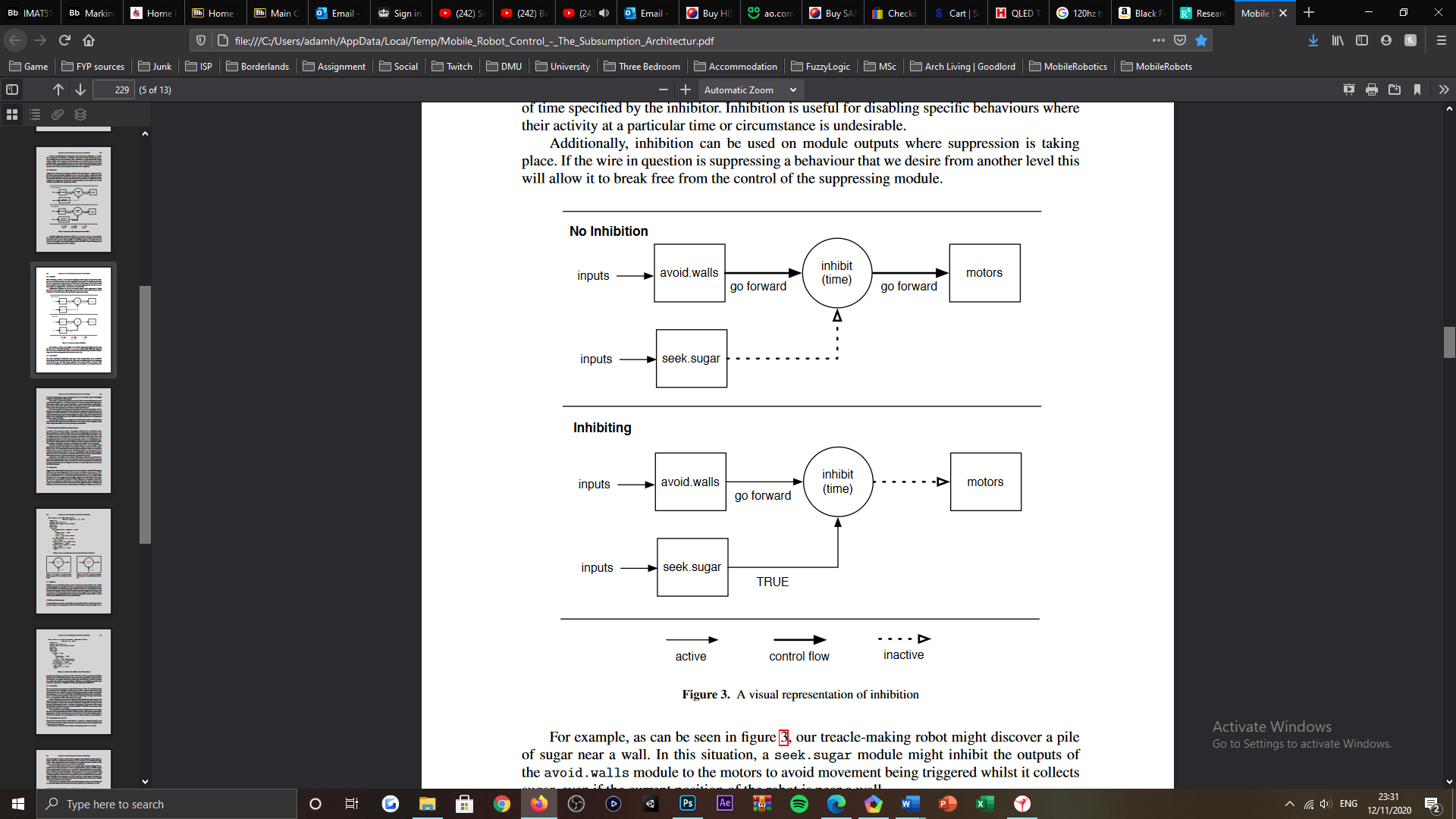


Figure : Behavioural inhibition, visualisation [2].

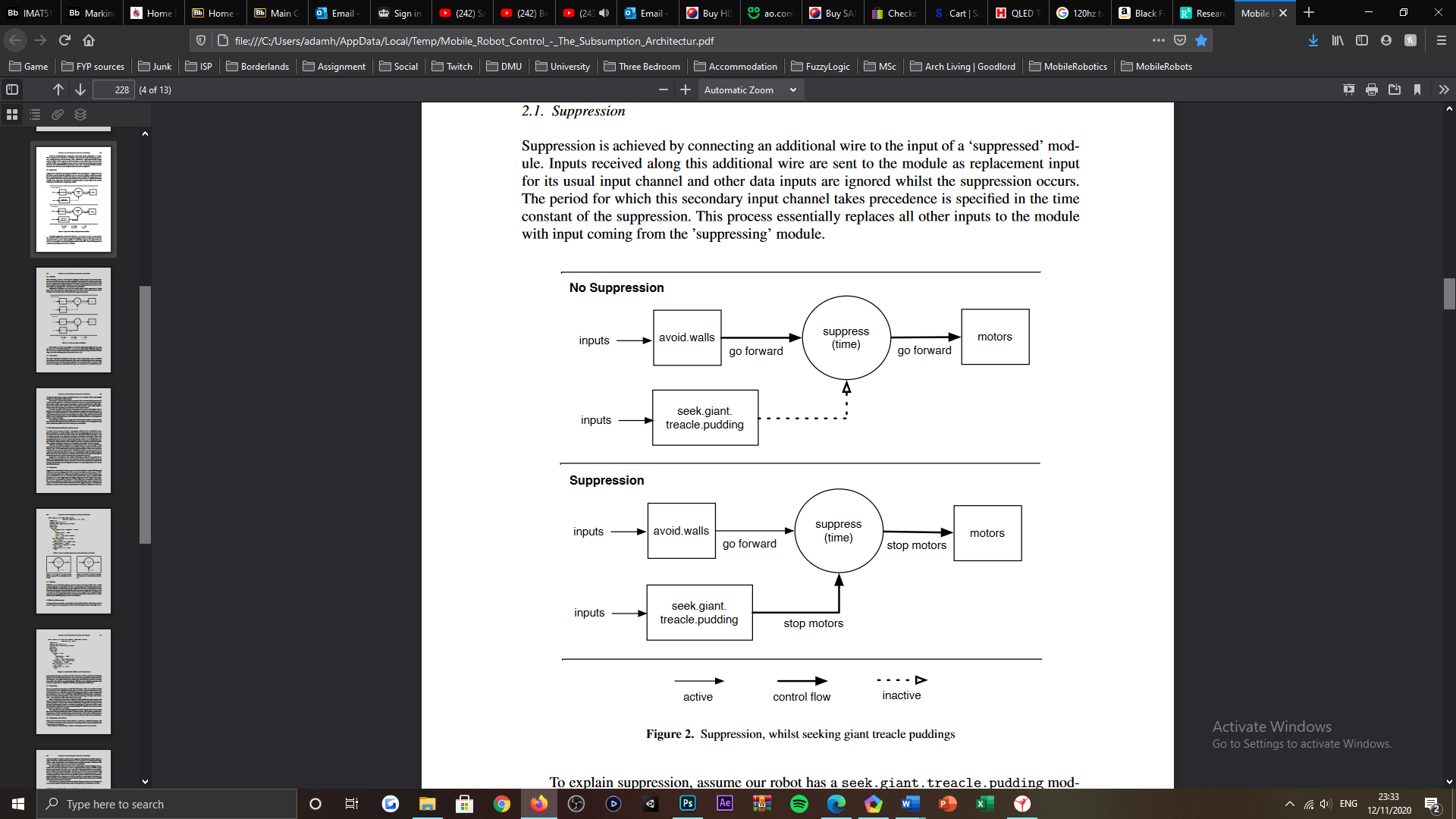


Figure : Behavioural suppression, visualisation [2].

***Appendix O:***

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| ***Random wandering state, relation between randomisation ranges*** | | | | | |
| ***Case*** | ***Variable*** | ***MIN*** | ***MAX*** | ***Observation*** | ***Final*** |
| ***1*** | robotWanderingDistance | 1 | 10 | *Robot mostly fails to rotate before detecting a wall. Robots traversal is not randomised for the scale of environment.* | No |
| robotRotationDirection | 1 | 2 |
| robotWanderingAngle | 1 | 360 |
| ***2*** | robotWanderingDistance | 1 | 5 | *Robot is mostly able to rotate before detecting a wall. Robots traversal is merely random for the scale of environment.* | No |
| robotRotationDirection | 1 | 2 |
| robotWanderingAngle | 1 | 360 |
| ***3*** | robotWanderingDistance | 1 | 2 | *Robot rotates couple iterations before detecting a wall. Robots traversal is somewhat random for the scale of the environment.* | No |
| robotRotationDirection | 1 | 2 |
| robotWanderingAngle | 1 | 360 |
| ***4*** | robotWanderingDistance | 0 | 1 | *Robot mostly stays within starting position, turning multiple iterations with minimal to no forward traversal.* | No |
| robotRotationDirection | 1 | 2 |
| robotWanderingAngle | 1 | 360 |
| ***5*** | robotWanderingDistance | 0.1 | 1 | *Robot demonstrates adequate traversal, not desired for smaller environments however.* | No |
| robotRotationDirection | 1 | 2 |
| robotWanderingAngle | 1 | 360 |
| ***6*** | robotWanderingDistance | 0.1 | 0.5 | *Robot fails to leave starting area, overturns into already explored area.* | No |
| robotRotationDirection | 1 | 2 |
| robotWanderingAngle | 1 | 360 |
| ***7*** | robotWanderingDistance | 0.1 | 0.5 | *Robots overturns into prior area, often. Robot rarely explores unknown areas.* | No |
| robotRotationDirection | 1 | 2 |
| robotWanderingAngle | 1 | 180 |
| ***8*** | robotWanderingDistance | 0.1 | 0.5 | *Robot turns merely, often, where the robot essentially traverses forward continually/* | No |
| robotRotationDirection | 1 | 2 |
| robotWanderingAngle | 1 | 90 |
| ***9*** | robotWanderingDistance | 0.1 | 0.5 | *Robot turns too similarly between each turning transition. Robot uncommonly explores unknown areas.* | No |
| robotRotationDirection | 1 | 10 |
| robotWanderingAngle | 45 | 90 |
| ***10*** | robotWanderingDistance | 0.1 | 0.5 | *Robot demonstrates frequent and randomised turning and traversal. Robot explores unexplored areas more frequently.* | Yes |
| robotRotationDirection | 1 | 100 |
| robotWanderingAngle | 30 | 90 |

Table : Random wandering, randomisation ranges for forward traversal, angular traversal direction and offset, test cases.

***Appendix P:***

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ***Edge following state, set-point and undetectable distance relation*** | | | | |
| ***Case*** | ***Variable*** | ***Value*** | ***Observation*** | ***Final*** |
| ***1*** | setPointPID | 0.3 | *Robot edge follows walls with interior angled perpendicular wall arrangements but not exterior angled perpendicular wall arrangements and adjacent faces.* | No |
| maximumDistancePID | 0.325 |
| undetectableDistance | 0.35 |
| ***2*** | setPointPID | 0.3 | *Robot edge follows walls with interior and exterior angled perpendicular wall arrangements but not adjacent faces.* | No |
| maximumDistancePID | 0.35 |
| undetectableDistance | 0.35 |
| ***3*** | setPointPID | 0.275 | *Robot edge follows walls with interior and exterior angled perpendicular wall arrangements but adjacent faces partially.* | No |
| maximumDistancePID | 0.35 |
| undetectableDistance | 0.35 |
| ***4*** | setPointPID | 0.25 | *Robot edge follows walls with interior and exterior angled perpendicular wall arrangements, as well as adjacent faces. Robot spaces from wall when edge following adjacent faces.* | No |
| maximumDistancePID | 0.35 |
| undetectableDistance | 0.35 |
| ***5*** | setPointPID | 0.25 | *Robot edge follows walls with interior and exterior angled perpendicular wall arrangements, as well as adjacent faces.* | Yes |
| maximumDistancePID | 0.275 |
| undetectableDistance | 0.35 |
| ***6*** | setPointPID | 0.2 | *Robot edge follows walls with interior and exterior angled perpendicular wall arrangements, as well as adjacent faces. Robot actuates sizable oscillatory motions when entering edge following behaviours.* | No |
| maximumDistancePID | 0.275 |
| undetectableDistance | 0.35 |
| ***7*** | setPointPID | 0.225 | *Robot edge follows walls with interior and exterior angled perpendicular wall arrangements, as well as adjacent faces. Robot actuates sizable oscillatory motions when entering edge following behaviours.* | No |
| maximumDistancePID | 0.275 |
| undetectableDistance | 0.35 |
| ***8*** | setPointPID | 0.235 | *Robot edge follows walls with interior and exterior angled perpendicular wall arrangements, as well as adjacent faces. Robot actuates noticeable oscillatory motions when entering edge following behaviours.* | No |
| maximumDistancePID | 0.275 |
| undetectableDistance | 0.35 |
| ***9*** | setPointPID | 0.245 | *Robot edge follows walls with interior and exterior angled perpendicular wall arrangements, as well as adjacent faces. Robot actuates subtle oscillatory motions when entering edge following behaviours sometimes.* | No |
| maximumDistancePID | 0.275 |
| undetectableDistance | 0.35 |
| ***10*** | setPointPID | 0.2475 | *Robot edge follows walls with interior and exterior angled perpendicular wall arrangements, as well as adjacent faces. Robot actuates subtle oscillatory motions when entering edge following behaviours rarely.* | No |
| maximumDistancePID | 0.275 |
| undetectableDistance | 0.35 |

Table : Edge following, set-point, maximum distance and undetectable distance value relationship, test cases.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| ***Edge following, PID controller gain value relation*** | | | | | |
| ***Case*** | ***Variable*** | ***Value*** | ***Observation*** | ***RMSE*** | ***Final*** |
| ***1*** | gainProportionalPID | 20 | *Robot maintains set-point but enters edge following with sizable oscillatory motions preventing the robot edge follow for some time.* | 0.00249 | No |
| gainIntegralPID | 10 |
| gainDerivativePID | 5 |
| ***2*** | gainProportionalPID | 10 | *Robot maintains set-point and does not exhibit noticeable oscillatory motions when entering edge following. Seamless transition.* | 0.00497 | No |
| gainIntegralPID | 5 |
| gainDerivativePID | 2.5 |
| ***3*** | gainProportionalPID | 15 | *Robot maintains set-point but does exhibit noticeable oscillatory motions when entering edge following. Does not hinder transition significantly.* | 0.00308 | No |
| gainIntegralPID | 7.5 |
| gainDerivativePID | 3.5 |
| ***4*** | gainProportionalPID | 12.5 | *Robot maintains set-point but does exhibit noticeable oscillatory motions when entering edge following sometimes. Does not hinder transition significantly.* | 0.00450 | No |
| gainIntegralPID | 6 |
| gainDerivativePID | 3 |
| ***5*** | gainProportionalPID | 12.5 | *Robot maintains set-point but does exhibit noticeable oscillatory motions when entering edge following sometimes. Does not hinder transition significantly.* | 0.00555 | No |
| gainIntegralPID | 6 |
| gainDerivativePID | 1 |
| ***6*** | gainProportionalPID | 12.5 | *Robot maintains set-point but does exhibit noticeable oscillatory motions when entering edge following sometimes. Does not hinder transition significantly.* | 0.00516 | No |
| gainIntegralPID | 3 |
| gainDerivativePID | 1 |
| ***7*** | gainProportionalPID | 10 | *Robot maintains set-point but does exhibit noticeable oscillatory motions when entering edge following sometimes. Does not hinder transition significantly.* | 0.00666 | No |
| gainIntegralPID | 3 |
| gainDerivativePID | 1 |
| ***8*** | gainProportionalPID | 5 | *Robot does not maintain set-point sometimes and enters random wandering but does not exhibit noticeable oscillatory motions when entering edge following.* | 0.00902 | No |
| gainIntegralPID | 3 |
| gainDerivativePID | 1 |
| ***9*** | gainProportionalPID | 7.5 | *Robot does not maintain set-point sometimes and enters random wandering but does not exhibit noticeable oscillatory motions when entering edge following.* | 0.00736 | No |
| gainIntegralPID | 3 |
| gainDerivativePID | 1 |
| ***10*** | gainProportionalPID | 7.5 | *Robot maintains set-point and does not exhibit noticeable oscillatory motions when entering edge following.* | 0.00661 | No |
| gainIntegralPID | 5 |
| gainDerivativePID | 1 |
| ***11*** | gainProportionalPID | 7.5 | *Robot maintains set-point but does exhibit subtle oscillatory motions when entering edge following sometimes. Does not hinder transition significantly.* | 0.00641 | No |
| gainIntegralPID | 6 |
| gainDerivativePID | 1 |
| ***12*** | gainProportionalPID | 8 | *Robot maintains set-point but does exhibit subtle oscillatory motions when entering edge following sometimes. Does not hinder transition significantly.* | 0.00613 | No |
| gainIntegralPID | 5 |
| gainDerivativePID | 1 |
| ***13*** | gainProportionalPID | 8 | *Robot maintains set-point but does exhibit subtle oscillatory motions when entering edge following sometimes. Does not hinder transition significantly.* | 0.00633 | No |
| gainIntegralPID | 5 |
| gainDerivativePID | 0.5 |
| ***14*** | gainProportionalPID | 8 | *Robot maintains set-point but does exhibit subtle oscillatory motions when entering edge following sometimes. Does not hinder transition significantly.* | 0.00699 | No |
| gainIntegralPID | 5 |
| gainDerivativePID | 0.1 |
| ***15*** | gainProportionalPID | 7 | *Robot maintains set-point and does not exhibit oscillatory motions when entering edge following sometimes.* | 0.00650 | Yes |
| gainIntegralPID | 4 |
| gainDerivativePID | 0.2 |

Table : Edge following, PID controller gain variable value relationship, test cases.

***Appendix Q:***



Figure : Time graph plots representing PID error (left) and RMSE (right) improving trends, over time.

***Appendix R***:

function sysCall\_init() -- Start of program runtime

do -----[ ROBOT VARIABLES ]-----

pioneerRobot = sim.getObjectHandle("Pioneer\_p3dx") -- Pioneer robot object

sonarSensor = { -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1} -- Robots sonar sensor component array

for i = 1, 16, 1 do -- For all of the robots sonar sensors, do the following

sonarSensor[i] = sim.getObjectHandle("Pioneer\_p3dx\_ultrasonicSensor".. i) -- Store the robots sonar sensor components

end -- End iteration

motorLeftWheel = sim.getObjectHandle("Pioneer\_p3dx\_leftMotor") -- Store the robots motor component for the left wheel

motorRightWheel = sim.getObjectHandle("Pioneer\_p3dx\_rightMotor") -- Store the robots motor component for the right wheel

undetectableDistance = 0.35 -- Robots distance that an object cannot be detected at or beyond

distanceObjectDetected = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0} -- Robots sonar sensor detection readings as distances to the objects detected

defaultWheelVelocity = 2 -- Robots default wheel velocity

end -----[ ROBOT VARIABLES ]-----

do -----[ WANDERING VARIABLES ]-----

robotWanderingAngle = 0 -- Robots wandering angle to rotate towards

robotWanderingDistance = 0 -- Robots wandering distance as forward traversal

robotPositionCurrent = { 0, 0, 0 } -- Robots current position array

robotPositionPrevious = { 0, 0, 0 } -- Robots previous position array

distanceForwardAccumulated = 0 -- Robots distance moved forward accumulated

robotWanderingDistanceSet = false -- Determine whether the robots forward traversal distance has been set

robotRotationDifference = 0 -- Robots rotational difference between its current and prior headings

robotRotationCurrent = { 0, 0, 0 } -- Robots current rotation offset array

accumulatedAngleRotated = 0 -- Robots angle accumulated whilst rotating

robotHeadingCurrent = 0 -- Robots heading for the current frame

robotHeadingPrevious = 0 -- Robots heading for the previous frame

robotWanderingReset = false -- Determine whether the wandering state variables are reset

end -----[ WANDERING VARIABLES ]-----

do -----[ EDGE FOLLOWING VARIABLES ]-----

setPointPID = 0.25 -- PID controller set-point to maintain from objects (walls)

maximumDistancePID = 0.275 -- PID controller maximum distance allowed to be considered for edge following objects (walls)

gainProportionalPID = 7 -- PID controller wheel velocity proportional gain

gainIntegralPID = 4 -- PID controller wheel velocity integral gain

gainDerivativePID = 0.2 -- PID controller wheel velocity derivative gain

errorLeftSumPID = { } -- PID controller edge following left error array

errorRightSumPID = { } -- PID controller edge following right error array

errorLeftCounterPID = 1 -- PID controller error counter for edge following left (accumulator)

errorRightCounterPID = 1 -- PID controller error counter for edge following right (accumulator)

integralThresholdPID = 10 -- PID controller error integral threshold (averaging amount)

errorLeftCurrentPID = 0 -- PID controller edge following left current error

errorLeftPreviousPID = 0 -- PID controller edge following left previous error

errorRightCurrentPID = 0 -- PID controller edge following right current error

errorRightPreviousPID = 0 -- PID controller edge following right previous error

errorLeftCountPID = 0 -- PID controller number of left errors obtained

errorRightCountPID = 0 -- PID controller number of right errors obtained

edgeFollowingLeft = false -- Determine whether the robot is edge following an object on its left side

edgeFollowingRight = false -- Determine whether the robot is edge following an object on its right side

edgeFollowingTransition = false -- Determine whether the robot will transition into edge following upon detecting an object

edgeFollowingTransitionLeft = false -- Determine whether the robot will transition into edge following left, upon detecting an object

edgeFollowingTransitionRight = false -- Determine whether the robot will transition into edge following right, upon detecting an object

edgeFollowingTransitionReset = false -- Determine whether the robot edge following transition variables will be reset

accumulatedLeftErrorRMSE = 0 -- Store the root mean squared error accumulated error (squared) for the robots left side

accumulatedRightErrorRMSE = 0 -- Store the root mean squared error accumulated error (squared) for the robots right side

rootMeanSquaredErrorPID = 0 -- Robot edge following root mean squared error value

graphRMSE = sim.getObjectHandle("RMSEGraph") -- RMSE graph object

graphSetPoint = sim.getObjectHandle("SetPointGraph") -- Set point graph object

end -----[ EDGE FOLLOWING VARIABLES ]-----

do -----[ CONSOLE OUTPUT ]------

robotHeadingOutput = 0 -- Robots heading representing facing direction (degrees)

robotMovementSpeed = 0 -- Robots velocity (meters per second)

robotTravelledDistance = 0 -- Robots travelled distance (metres)

robotCurrentPosition = { 0, 0, 0 } -- Robots current position in world space

robotPreviousPosition = { 0, 0, 0 } -- Robots previous position in world space

objectDetectedLeftString = "" -- Store the distance detected closest to the robot, relative to the left of its body

objectDetectedRightString = "" -- Store the distance detected closest to the robot, relative to the right of its body

edgeFollowingTransitionString = "" -- Store the robots edge following transition criteria

end -----[ CONSOLE OUTPUT ]------

do -----{ STATE VARIABLES )-----

robotEdgeFollowing = false -- Determine whether the robot is edge following an object

robotRotating = false -- Determine whether the robot is rotating around its own axes

end -----[ STATE VARIABLES ]------

end -- End function

function sysCall\_cleanup() -- End of program runtime

-- Write to external document potential

end -- End function

function sysCall\_sensing() -- Robot sensory functionality

do -----[ ROBOT POSITION AND ORIENTATION ]-----

robotPositionCurrent = sim.getObjectPosition(pioneerRobot,-1) -- Store the robots current position (all axes)

robotRotationCurrent = sim.getObjectOrientation(pioneerRobot,-1) -- Store the robots current orientation (all axes)

robotHeadingCurrent = math.deg(robotRotationCurrent[3]) -- Store the current heading of the robot (Z axis)

robotHeadingOutput = robotHeadingCurrent -- Store the current heading of the robot for console output

if (robotHeadingCurrent < 0) then -- If the current heading of the robot is negative, do the following

robotHeadingCurrent = -(robotHeadingCurrent) -- Set the current heading of the robot to the current heading of the robot, negated (positive)

end -- End condition

end -----[ ROBOT POSITION AND ORIENTATION ]-----

do -----[ SENSING ]-----

do -----[ CHECK ALL SENSORS }-----

accumulatedDetectedDistance = 0 -- Reset the distance accumulated by the robots sensor detections

for i = 1, 16, 1 do -- For all of the robots sonar sensors, do the following

detectionStatus, distanceReturned = sim.readProximitySensor(sonarSensor[i]) -- Store the detection status and the distance an object was detected at (if detected) from the robots position for the currently iterated sonar sensor

if (detectionStatus > 0) and (distanceReturned < undetectableDistance) then -- If the currently iterated sonar sensor detected an object within the robots detectable range, do the following

distanceObjectDetected[i] = distanceReturned -- Store the curently iterated sonar sensor reading into the current index of the detected object distance array

else -- If the currently iterated sonar sensor has not detected an object, do the following

distanceObjectDetected[i] = 0 -- Set the currently iterated detected object distance to '0' (an object was not detected)

end -- End condition

end -- End iteration

end -----[ CHECK ALL SENSORS )-----

do -----[ EDGE FOLLOWING ]-----

for i = 1, 16, 1 do -- For all of the robots sonar sensors, do the following

if (distanceObjectDetected[i] > 0) then -- If an object has been detected by the currently iterated sonar sensor, do the following

accumulatedDetectedDistance = accumulatedDetectedDistance + distanceObjectDetected[i] -- Add and equal the accumulated distance detected with the distance detected by the currently iterated sonar sensor

end -- End condition

end -- End iteration

if (accumulatedDetectedDistance > 0) then -- If an object has been detected, do the following

if (distanceObjectDetected[1] > 0 and distanceObjectDetected[3] == 0) then -- If the robots left-most front facing sensor has detected an object and a front-left sensor has not detected an object (parallel to object), do the following

edgeFollowingRight = false -- The robot will not enter edge following for its right side

sensorDetectedObjectCounter = 0 -- Store the number of sonar sensors with a detected distance

for i = 5, 8, 1 do -- For the robots front-right facing sensors, do the following

if (distanceObjectDetected[i] > 0) then -- If the currently iterated sonar sensor has detected an object, do the following

sensorDetectedObjectCounter = sensorDetectedObjectCounter + 1 -- Increment the number of front-right sensors that have detected an object

end -- End condition

end -- End iteration

if (sensorDetectedObjectCounter == 0) then -- If the robots front-right facing sonar sensors have not detected an object, do the following

edgeFollowingLeft = true -- Set the robot to edge follow for its left side

else -- If the robots front-right facing sensors have detected an object, do the following

edgeFollowingLeft = false -- The robot will not edge follow for its left side

end -- End condition

elseif (distanceObjectDetected[8] > 0 and distanceObjectDetected[6] == 0) then -- If the robots right-most front facing sensor has detected an object and a front-right sensor has not detected an object (parallel to object), do the following

edgeFollowingLeft = false -- The robot will not enter edge following for its left side

sensorDetectedObjectCounter = 0 -- Store the number of sonar sensors with a detected distance

for i = 1, 4, 1 do -- For the robots front-left facing sensors, do the following

if (distanceObjectDetected[i] > 0) then -- If the currently iterated sensor has detected an object, do the following

sensorDetectedObjectCounter = sensorDetectedObjectCounter + 1 -- Increment the number of front-left sensors that have detected an object

end -- End condition

end -- End iteration

if (sensorDetectedObjectCounter == 0) then -- If the robots front-left facing sonar sensors have not detected an object, do the following

edgeFollowingRight = true -- Set the robot to edge follow for its right side

else -- If the robots front-left facing have detected an object, do the following

edgeFollowingRight = false -- The robot will not edge follow for its right side

end -- End condition

else -- If the robots left-most or right-most front facing sensor does not detect an object when their other corresponding front facing sensor does not detect an object, do the following

edgeFollowingLeft = false -- The robot will not edge follow for its left side

edgeFollowingRight = false -- The robot will not edge follow for its right side

edgeFollowingTransition = true -- The robot will transition into edge following as an object was detected by the robot

end -- End condition

if (edgeFollowingLeft == true or edgeFollowingRight == true or edgeFollowingTransition == true) then -- If the robot has been set to transition into or to edge follow, do the following

robotEdgeFollowing = true -- The robot will edge follow

else -- If the robot has not been set to transition into or to edge follow, do the following

robotEdgeFollowing = false -- The robot will wander

end -- End condition

else -- If the robot has not detected an object, do the following

robotEdgeFollowing = false -- The robot will wander

end -- End condition

end -----[ EDGE FOLLOWING ]-----

end -----[ SENSING ]-----

do -----[ CONSOLE OUTPUT NEAREST OBJECT ]-----

nearestDetectionLeft = { 0, 0 } -- Store the index of the sonar sensor and the distance detected to the closest object, relative to the robots left side

nearestDetectionRight = { 0, 0 } -- Store the index of the sonar sensor and the distance detected to the closest object, relative to the robots right side

nearestDetectionLeftSet = false -- Determine whether a sonar sensor and its reading have been set (initialisation), so that readings can be compared for objects detected on the robots left side

nearestDetectionRightSet = false -- Determine whether a sonar sensor and its reading have been set (initialisation), so that readings can be compared for objects detected on the robots right side

objectsDetectedLeft = 0 -- Store the number of sonar sensors that detect an object on the robots left side

objectsDetectedRight = 0 -- Store the number of sonar sensors that detect an object on the robots right side

do -----{ CHECK SENSOR DETECTIONS }-----

do -----{ ACCUMULATE DETECTIONS LEFT }-----

for i = 1, 4, 1 do -- For the robots front-left facing sensors, do the following

if (distanceObjectDetected[i] > 0) then -- If the currently iterated sonar sensor detected an object, do the following

objectsDetectedLeft = objectsDetectedLeft + 1 -- Increment the object detected left counter

end -- End condition

end -- End iteration

for i = 13, 16, 1 do -- For the robots back-left facing sensors, do the following

if (distanceObjectDetected[i] > 0) then -- If the currently iterated sonar sensor detected an object, do the following

objectsDetectedLeft = objectsDetectedLeft + 1 -- Increment the object detected left counter

end -- End condition

end -- End iteration

end -----{ ACCUMULATE DETECTIONS LEFT }-----

do -----{ ACCUMULATE DETECTIONS RIGHT }-----

for i = 5, 12, 1 do -- For the robots right facing sensors, do the following

if (distanceObjectDetected[i] > 0) then -- If the currently iterated sonar sensor detected an object, do the following

objectsDetectedRight = objectsDetectedRight + 1 -- Increment the object detected left counter

end -- End condition

end -- End iteration

end -----{ ACCUMULATE DETECTIONS RIGHT }-----

end -----{ CHECK SENSOR DETECTIONS }-----

do -----{ DETERMINE CLOSEST OBJECTS }-----

if (objectsDetectedLeft + objectsDetectedRight > 0) then -- If the robot has detected an object, do the following

for i = 1, 16, 1 do -- For all of the robots sonar sensors, do the following

if (distanceObjectDetected[i] > 0) then -- If the currently iterated sonar sensor has detected an object, do the following

if (i >= 1 and i <= 4) then -- If the currently iterated sonar sensor is a front-left facing sensor, do the following

if (nearestDetectionLeftSet == false) then -- If the sonar sensor and nearest distance detected by the robots left sensors has not been set, do the following

nearestDetectionLeft[1] = i -- Store the index of the currently iterated sonar sensor that has detected the object

nearestDetectionLeft[2] = distanceObjectDetected[i] -- Store the distance detected to the object by the currently iterated sonar sensor

nearestDetectionLeftSet = true -- The sonar sensor and nearest distance detected by the robots left sensors has been set

else -- If the sonar sensor and nearest distance detected by the robots left sensors has been set, do the following

if (distanceObjectDetected[i] < nearestDetectionLeft[2]) then -- If the distance detected is closer than the closest distance detected to an object for the currently iterated sonar sensor, do the following

nearestDetectionLeft[1] = i --- Store the index of the currently iterated sonar sensor that has detected the object

nearestDetectionLeft[2] = distanceObjectDetected[i] -- Store the distance detected to the object by the currently iterated sonar sensor

end -- End condition

end -- End condition

elseif (i >= 5 and i <= 12) then -- If the currently iterated sonar sensor is a right facing sensor, do the following

if (nearestDetectionRightSet == false) then -- If the sonar sensor and nearest distance detected by the robots right sensors has not been set, do the following

nearestDetectionRight[1] = i -- Store the index of the currently iterated sonar sensor that has detected the object

nearestDetectionRight[2] = distanceObjectDetected[i] -- Store the distance detected to the object by the currently iterated sonar sensor

nearestDetectionRightSet = true -- The sonar sensor and nearest distance detected by the robots right sensors has been set

else -- If the sonar sensor and nearest distance detected by the robots right sensors has not been set, do the following

if (distanceObjectDetected[i] < nearestDetectionRight[2]) then -- If the distance detected is closer than the closest distance detected to an object for the currently iterated sonar sensor, do the following

nearestDetectionRight[1] = i -- Store the index of the currently iterated sonar sensor that has detected the object

nearestDetectionRight[2] = distanceObjectDetected[i] -- Store the distance detected to the object by the currently iterated sonar sensor

end -- End condition

end -- End condition

elseif (i >= 13 and i <= 16) then -- If the currently iterated sonar sensor is a back-left facing sensor, do the following

if (distanceObjectDetected[i] < nearestDetectionLeft[2]) then -- If the distance detected is closer than the closest distance detected to an object for the currently iterated sonar sensor, do the following

nearestDetectionLeft[1] = i -- Store the index of the currently iterated sonar sensor that has detected the object

nearestDetectionLeft[2] = distanceObjectDetected[i] -- Store the distance detected to the object by the currently iterated sonar sensor

end -- End condition

end -- End condition

end -- End condition

end -- End iteration

end -- End condition

end -----{ DETERMINE CLOSEST OBJECTS }-----

do -----[ STRINGIZE ]-----

if (objectsDetectedLeft == 0) then -- If the robot did not detect an object on its left side, do the following

objectDetectedLeftString = "NO DETECTION" -- Set the object detected left string to 'no detection' (object not detected)

else -- If the robot did detect an object on its left side, do the following

-- Set the object detected left string to the sonar sensor and its corresponding reading, that was the closest distance detected to an object on the robots left side

objectDetectedLeftString = "Sensor [" .. nearestDetectionLeft[1] .. "] at " .. string.format("%.2f", nearestDetectionLeft[2]) .. " m"

end -- End condition

if (objectsDetectedRight == 0) then -- If the robot did not detect an object on its right side, do the following

objectDetectedRightString = "NO DETECTION" -- Set the object detected right string to 'no detection' (object not detected)

else -- If the robot did detect an object on its right side, do the following

-- Set the object detected left string to the sonar sensor and its corresponding reading, that was the closest distance detected to an object on the robots right side

objectDetectedRightString = "Sensor [" .. nearestDetectionRight[1] .. "] at " .. string.format("%.2f", nearestDetectionRight[2]) .. " m"

end -- End condition

end -----{ STRINGIZE }-----

end -----[ CONSOLE OUTPUT NEAREST OBJECT ]-----

do -----[ CONSOLE ROBOT VELOCITY CALCULATIONS ]-----

robotDistanceDifference = { 0, 0, 0 } -- Store the difference between the current and previous positions of the robot

robotCurrentPosition = sim.getObjectPosition(pioneerRobot,-1) -- Store the current position of the robot

for i = 1, 3, 1 do -- For the size of the array, do the following

robotDistanceDifference[i] = robotCurrentPosition[i] - robotPreviousPosition[i] -- Calculate the difference between the robots current and previous positions (last frame)

end -- End iteration

-- Set the distance travelled of the robot to the magnitude of the difference between the previous and current position vectors

robotTravelledDistance = math.sqrt((robotDistanceDifference[1] \* robotDistanceDifference[1]) + (robotDistanceDifference[2] \*

robotDistanceDifference[2]) + (robotDistanceDifference[3] \* robotDistanceDifference[3]))

robotSpeed = robotTravelledDistance / sim.getSimulationTimeStep() -- Calculate the distance travelled by the robot since the last frame was made (speed = distance / time)

robotTravelledDistance = 0 -- Reset the distance travelled by the robot

robotMovementSpeed = robotSpeed -- Set the robots movement speed to the speed calculated

robotPreviousPosition = robotCurrentPosition -- Set the previous position to the current position of the robot (end of frame)

end -----[ CONSOLE ROBOT VELOCITY CALCULATIONS ]-----

end -- End function

function sysCall\_actuation() -- Robot actuation functionality

if (robotEdgeFollowing == true) then -- If the robot is edge following, do the following

robotFollowing() -- Function call, invocate edge following behaviours

else -- If the robot is not edge following, do the following

robotWandering() -- Function call, invocate random wandering behaviours

end -- End condition

end -- End function

function robotWandering() -- Robot wander functionality

if (robotWanderingReset == true) then -- If the robots wandering state variables require to be reset, do the following

do -----[ RESET WANDERING VARIABLES ]-----

robotWanderingAngle = 0 -- Reset the robots targeted angle to rotate towards

robotWanderingDistance = 0 -- Reset the robots targeted forward traversal distance

robotPositionCurrent = { 0, 0, 0 } -- Reset the robots current position array

robotPositionPrevious = { 0, 0, 0 } -- Reset the robots previous position array

distanceForwardAccumulated = 0 -- Reset the distance the robot has traversed forwards for

robotWanderingDistanceSet = false -- The robots targeted forward traversal distance has not been set

robotRotationDifference = 0 -- Reset the robots rotational difference

robotRotationCurrent = { 0, 0, 0 } -- Reset the robots current rotation array

accumulatedAngleRotated = 0 -- Reset the angle the robot has rotated for

robotHeadingCurrent = 0 -- Reset the current heading of the robot

robotHeadingPrevious = 0 -- Reset the previous heading of the robot

robotRotating = false -- The robot is not rotating (forward traversal inversely)

end -----[ RESET WANDERING VARIABLES ]-----

robotWanderingReset = false -- The robots wandering state variables have been reset

else -- If the robots wandering state variables do not require to be reset, do the following

if (robotRotating == false) then -- If the robot is not rotating, do the following

do -----[ RESET ROTATING VARIABLES ]-----

robotRotationDifference = 0 -- Reset the robots rotational difference

robotRotationDirectionSet = false -- Reset the robots rotary direction

robotLeftTurning = false -- The robot is not turning left

robotRightTurning = false -- The robot is not turning right

robotWanderingAngle = 0 -- Reset the robots targeted angle to rotate towards

end -----[ RESET ROTATING VARIABLES ]-----

if (robotWanderingDistanceSet == false) then -- If the robots forward traversal distance has not been set, do the following

robotWanderingDistance = math.random(1, 5) -- Generate a forward traversal distance for the robot to accumulate

do -----[ RESET POSITION VARIABLES ]-----

distanceForwardAccumulated = 0 -- Reset the distance the robot has traversed forwards for

positionDifference = { 0, 0, 0 } -- Reset the robots difference in position array

robotPositionCurrent = { 0, 0, 0 } -- Reset the robots current position array

robotPositionPrevious = { 0, 0, 0 } -- Reset the robots previous position array

end -----[ RESET POSITION VARIABLES ]-----

robotWanderingDistanceSet = true -- The robots targeted forward traversal distance has been set

end -- End condition

if (robotPositionPrevious[1] > 0 or robotPositionPrevious[2] > 0 or robotPositionPrevious[3] > 0) then -- If the robots previous position is not the centre of the world space, do the following

for i = 1, 2, 1 do -- For the size of the array, do the following

positionDifference[i] = robotPositionCurrent[i] - robotPositionPrevious[i] -- Set the difference in position to be the difference between the current and previous positions of the robot

end -- End iteration

-- Set the robots distance travelled to the magnitude of the difference between the robots current and previous position vectors

distanceForwardAccumulated = distanceForwardAccumulated + math.sqrt((positionDifference[1] \* positionDifference[1]) +

(positionDifference[2] \* positionDifference[2]))

if (robotWanderingDistanceSet == true) then -- If the robots forward traversal distance has been set, do the following

if (distanceForwardAccumulated >= (robotWanderingDistance / 10)) then -- If the robots distance travelled is equal to or larger than the targeted distance of traversal, do the following

robotWanderingDistanceSet = false -- The robots targeted forward traversal distance has not been set (reset)

robotRotating = true -- Set the robot to rotate

end -- End condition

end -- End condition

end -- End condition

sim.setJointTargetVelocity(motorLeftWheel, defaultWheelVelocity) -- Set the left wheel motor velocity of the robot to the default velocity

sim.setJointTargetVelocity(motorRightWheel, defaultWheelVelocity) -- Set the right wheel motor velocity of the robot to the default velocity

do -----[ CONSOLE OUTPUT ]-----

printf("Speed [" .. string.format("%.2f", robotMovementSpeed) .. " m/s] Heading [" .. string.format("%.2f", robotHeadingOutput) .. " DEG] "

.. "Position [" .. string.format("%.2f", robotPositionCurrent[1]) .. ", " .. string.format("%.2f", robotPositionCurrent[2]) .. "] "

.. "Detection Left [" .. objectDetectedLeftString .. "] " .. "Detection Right [" .. objectDetectedRightString .. "] "

.. "State [Wandering] -----> Moving Forward Distance Travelled [" .. string.format("%.2f", distanceForwardAccumulated)

.. "] Distance Travelling To [" .. (robotWanderingDistance / 10) .. "]") -- Output the robot entered the 'moving forward' state

end -----[ CONSOLE OUTPUT ]-----

robotPositionPrevious = robotPositionCurrent -- Set the previous position to the current position of the robot (end of frame)

else -- If the robot is rotating, do the following

if (robotRotationDirectionSet == false) then -- If the rotary direction of the robot has not been set, do the following

accumulatedAngleRotated = 0 -- Reset the angle the robot has rotated for

robotRotationDirection = math.random(1, 100) -- Generate a rotary direction for the robot to turn in

robotWanderingAngle = math.random(30, 90) -- Generate a targeted angle for the robot to rotate towards

if (robotRotationDirection % 2 == 0) then -- If the rotation direction is an even number (no remainder), do the following

robotRightTurning = true -- The robot will turn right

else -- If the rotation direction is an odd number (has a remainder), do the following

robotLeftTurning = true -- The robot will turn left

end -- End condition

robotRotationDirectionSet = true -- The robots rotary direction has been set

else -- If the rotary direction of the robot has been set, do the following

robotRotationDifference = robotHeadingCurrent - robotHeadingPrevious -- Set the robots rotary difference to the difference between the robots current heading and the robots previous heading

if (robotRotationDifference < 0) then -- If the robots rotary difference is negative, do the following

accumulatedAngleRotated = accumulatedAngleRotated + -(robotRotationDifference) -- Add and equal the robots rotary difference negated

else -- If the robots rotary difference is positive, do the following

accumulatedAngleRotated = accumulatedAngleRotated + robotRotationDifference -- Add and equal the robots rotary difference

end -- End condition

if (robotLeftTurning == true) then -- If the robot is rotating left, do the following

if (accumulatedAngleRotated >= robotWanderingAngle) then -- If the angle that the robot has rotated is equal to or larger than the targeted angle, do the following

robotRotating = false -- Set the robot to traverse forwards

end -- End condition

sim.setJointTargetVelocity(motorLeftWheel, defaultWheelVelocity / 2) -- Set the left wheel motor velocity of the robot to '2' times less the default velocity

sim.setJointTargetVelocity(motorRightWheel, defaultWheelVelocity \* 1.5) -- Set the right wheel motor velocity of the robot to '1.5' times more the default velocity

do -----[ CONSOLE OUTPUT ]-----

printf("Speed [" .. string.format("%.2f", robotMovementSpeed) .. " m/s] Heading [" .. string.format("%.2f", robotHeadingOutput) .. " DEG] "

.. "Position [" .. string.format("%.2f", robotPositionCurrent[1]) .. ", " .. string.format("%.2f", robotPositionCurrent[2]) .. "] "

.. "Detection Left [" .. objectDetectedLeftString .. "] " .. "Detection Right [" .. objectDetectedRightString .. "] "

.. "State [Wandering] -----> Turning Left Angle Rotated [" .. string.format("%.2f", accumulatedAngleRotated)

.. "] Angle Rotating To [" .. robotWanderingAngle .. "]") -- Output the robot entered the 'turning left' state

end -----[ CONSOLE OUTPUT ]-----

elseif (robotRightTurning == true) then -- if the robot is turning right, do the following

if (accumulatedAngleRotated >= robotWanderingAngle) then -- If the angle that the robot has rotated is equal to or larger than the targeted angle, do the following

robotRotating = false -- Set the robot to traverse forwards

end -- End condition

sim.setJointTargetVelocity(motorLeftWheel, defaultWheelVelocity \* 1.5) -- Set the left wheel motor velocity of the robot to '1.5' times more the default velocity

sim.setJointTargetVelocity(motorRightWheel, defaultWheelVelocity / 2) -- Set the right wheel motor velocity of the robot to '2' times less the default velocity

do -----[ CONSOLE OUTPUT ]-----

printf("Speed [" .. string.format("%.2f", robotMovementSpeed) .. " m/s] Heading [" .. string.format("%.2f", robotHeadingOutput) .. " DEG] "

.. "Position [" .. string.format("%.2f", robotPositionCurrent[1]) .. ", " .. string.format("%.2f", robotPositionCurrent[2]) .. "] "

.. "Detection Left [" .. objectDetectedLeftString .. "] " .. "Detection Right [" .. objectDetectedRightString .. "] "

.. "State [Wandering] -----> Turning Right Angle Rotated [" .. string.format("%.2f", accumulatedAngleRotated)

.. "] Angle Rotating To [" .. robotWanderingAngle .. "]") -- Output the robot entered the 'turning right' state

end -----[ CONSOLE OUTPUT ]-----

end -- End condition

end -- End condition

robotHeadingPrevious = robotHeadingCurrent -- Set the previous heading to the current heading of the robot (end of frame)

end -- End condition

end -- End condition

end -- End function

function robotFollowing() -- Robot edge following functionality

-- If the robot is edge following an object on its left or right sides and the edge following transition variables have been reset, do the following

if (edgeFollowingTransitionReset == true and edgeFollowingLeft == true or edgeFollowingTransitionReset == true and edgeFollowingRight == true) then

edgeFollowingTransition = false -- Reset the edge following transition state

edgeFollowingTransitionRight = false -- Reset the edge following transition state for the right side of the robot

edgeFollowingTransitionLeft = false -- Reset the edge following transition state for the left side of the robot

edgeFollowingTransitionReset = false -- The edge following transition variables have been reset

robotWanderingReset = true -- The robots wandering variables require to be reset (interrupted)

end -- End condition

if (edgeFollowingLeft == true) then -- If the robot has been set to edge follow an object on its left side, do the following

leftSensorDetectionStatus, leftSensorDetectedDistance = sim.readProximitySensor(sonarSensor[1]) -- Store the sonar sensors object detection status and the distance detected at (left-most sensor)

if (leftSensorDetectionStatus > 0) and (leftSensorDetectedDistance ~= setPointPID) then -- If an object was detected and the robot is not at the set-point, do the following

if (leftSensorDetectionStatus < 0) then -- If the left-most sensor malfunctioned, do the following

leftSensorDetectedDistance = maximumDistancePID -- Set the detected distance to the maximum distance allowed for edge following

end -- End condition

do -----[ CALCULATE LEFT ERROR ]-----

errorLeft = setPointPID - leftSensorDetectedDistance -- Set the left error to the difference between the robots distance to the object and the set-point

errorLeftSumPID[errorLeftCounterPID] = errorLeft -- Store the error into the left error array, indexed at the current left error count

errorLeftCurrentPID = errorLeft -- Store the current left error calculated

errorLeftCounterPID = errorLeftCounterPID + 1 -- Increment the left error counter

errorLeftCountPID = errorLeftCountPID + 1 -- Increment the number of left errors obtained count

if (errorLeftCounterPID > integralThresholdPID) then -- If the left error counter is greater than the size of the error array, do the following

errorLeftCounterPID = 1 -- Reset the left error counter (for future iterations)

end -- End condition

if (errorLeftCounterPID == 1) then -- If the left error counter is equal to '1', do the following

errorLeftPreviousPID = errorLeftSumPID[table.getn(errorLeftSumPID)] -- Set the previous left error to the last element in the error array, relative to the intial size of the error array

else -- If the left error counter is not equal to '1', do the following

errorLeftPreviousPID = errorLeftSumPID[errorLeftCounterPID - 1] -- Set the previous left error to the error previously stored in the error array

end -- End condition

errorLeftAccumulated = 0 -- Store the sum of left errors

for i = 1, table.getn(errorLeftSumPID), 1 do -- For the size of the array, do the following

errorLeftAccumulated = errorLeftAccumulated + errorLeftSumPID[i] -- Add and equal the currently iterated error to the error sum

end -- End iteration

do -----{ RMSE }-----

accumulatedLeftErrorRMSE = accumulatedLeftErrorRMSE + (errorLeft)^2 -- Add and equal the currently iterated left error, squared, to the accumulated left RMSE value

rootMeanSquaredErrorPID = math.sqrt(accumulatedLeftErrorRMSE / errorLeftCountPID) -- Calculate the root mean squared error for the number of errors obtained

sim.setGraphUserData(graphRMSE, "RMSE", rootMeanSquaredErrorPID) -- Plot the RMSE data to the RMSE graph

sim.handleGraph(graphRMSE, sim.getSimulationTime() + sim.getSimulationTimeStep()) -- Explicitly handle the graph (allows reset)

end -----{ RMSE }-----

sim.setGraphUserData(graphSetPoint, "Error", errorLeft) -- Plot the error (distance) data to the set point graph

sim.handleGraph(graphSetPoint, sim.getSimulationTime() + sim.getSimulationTimeStep()) -- Explicitly handle the graph (allows reset)

end -----[ CALCULATE LEFT ERROR ]-----

do -----[ APPLY PID GAINS TO WHEEL MOTORS ]-----

-- Set the motor velocity of the robots left wheel to the default velocity added with the calculated PID gains

leftWheelVelocity = defaultWheelVelocity +

(gainProportionalPID \* errorLeft) + -- Calculate the proportional gain

(gainIntegralPID \* (errorLeftAccumulated / integralThresholdPID)) + -- Calculate the integral gain

(gainDerivativePID \* (errorLeftPreviousPID - errorLeftCurrentPID)) -- Calculate the derivative gain

rightWheelVelocity = defaultWheelVelocity -- Set the motor velocity of the robots right wheel to the default velocity

sim.setJointTargetVelocity(motorLeftWheel, leftWheelVelocity) -- Set the left wheel motor velocity of the robot to the calculated velocity

sim.setJointTargetVelocity(motorRightWheel, rightWheelVelocity) -- Set the right wheel motor velocity of the robot to the calculated velocity

end -----[ APPLY PID GAINS TO WHEEL MOTORS ]-----

do -----[ CONSOLE OUTPUT ]-----

if (errorLeft > 0) then -- If the left error is larger than '0' (too close to the object), do the following

printf("Speed [" .. string.format("%.2f", robotMovementSpeed) .. " m/s] Heading [" .. string.format("%.2f", robotHeadingOutput) .. " DEG] "

.. "Position [" .. string.format("%.2f", robotPositionCurrent[1]) .. ", " .. string.format("%.2f", robotPositionCurrent[2]) .. "] "

.. "Detection Left [" .. objectDetectedLeftString .. "] " .. "Detection Right [" .. objectDetectedRightString .. "] "

.. "State [Edge Following] Left -----> Turning Outwards [RMSE " .. string.format("%.5f", rootMeanSquaredErrorPID) .. "]") -- Output the robot entered the 'turning outwards' state

else -- If the left error is smaller than '0' (too far from the object), do the following

printf("Speed [" .. string.format("%.2f", robotMovementSpeed) .. " m/s] Heading [" .. string.format("%.2f", robotHeadingOutput) .. " DEG] "

.. "Position [" .. string.format("%.2f", robotPositionCurrent[1]) .. ", " .. string.format("%.2f", robotPositionCurrent[2]) .. "] "

.. "Detection Left [" .. objectDetectedLeftString .. "] " .. "Detection Right [" .. objectDetectedRightString .. "] "

.. "State [Edge Following] Left -----> Turning Inwards [RMSE " .. string.format("%.5f", rootMeanSquaredErrorPID) .. "]") -- Output the robot entered the 'turning inwards' state

end -- End condition

end -----[ CONSOLE OUTPUT ]-----

end -- End condition

elseif (edgeFollowingRight == true) then -- If the robot has been set to edge follow an object on its right side, do the following

rightSensorDetectionStatus, rightSensorDetectedDistance = sim.readProximitySensor(sonarSensor[8]) -- Store the sonar sensors object detection status and the distance detected at (right-most sensor)

if (rightSensorDetectionStatus > 0) and (rightSensorDetectedDistance ~= setPointPID) then -- If an object was detected and the robot is not at the set-point, do the following

if (rightSensorDetectionStatus < 0) then -- If the right-most sensor malfunctioned, do the following

rightSensorDetectedDistance = maximumDistancePID -- Set the detected distance to the maximum distance allowed for edge following

end -- End condition

do -----[ CALCULATE RIGHT ERROR ]-----

errorRight = setPointPID - rightSensorDetectedDistance -- Set the right error to the difference between the robots distance to the object and the set-point

errorRightSumPID[errorRightCounterPID] = errorRight -- Store the error into the right error array, indexed at the current right error count

errorRightCurrentPID = errorRight -- Store the current right error calculated

errorRightCounterPID = errorRightCounterPID + 1 -- Increment the right error counter

errorRightCountPID = errorRightCountPID + 1 -- Increment the number of right errors obtained count

if (errorRightCounterPID > integralThresholdPID) then -- If the right error counter is greater than the size of the error array, do the following

errorRightCounterPID = 1 -- Reset the right error counter (for future iterations)

end -- End condition

if (errorRightCounterPID == 1) then -- If the right error counter is equal to '1', do the following

errorRightPreviousPID = errorRightSumPID[table.getn(errorRightSumPID)] -- Set the previous right error to the last element in the error array, relative to the intial size of the error array

else -- If the right error counter is not equal to '1', do the following

errorRightPreviousPID = errorRightSumPID[errorRightCounterPID - 1] -- Set the previous right error to the error previously stored in the error array

end -- End condition

errorRightAccumulated = 0 -- Store the sum of right errors

for i = 1, table.getn(errorRightSumPID), 1 do -- For the size of the array, do the following

errorRightAccumulated = errorRightAccumulated + errorRightSumPID[i] -- Add and equal the currently iterated error to the error sum

end -- End iteration

do -----{ RMSE }-----

accumulatedRightErrorRMSE = accumulatedRightErrorRMSE + (errorRight)^2 -- Add and equal the currently iterated right error, squared, to the accumulated right RMSE value

rootMeanSquaredErrorPID = math.sqrt(accumulatedRightErrorRMSE / errorRightCountPID) -- Calculate the root mean squared error for the number of errors obtained

sim.setGraphUserData(graphRMSE, "RMSE", rootMeanSquaredErrorPID) -- Plot the RMSE data to the RMSE graph

sim.handleGraph(graphRMSE, sim.getSimulationTime() + sim.getSimulationTimeStep()) -- Explicitly handle the graph (allows reset)

end -----{ RMSE }-----

sim.setGraphUserData(graphSetPoint, "Error", errorRight) -- Plot the error (distance) data to the set point graph

sim.handleGraph(graphSetPoint, sim.getSimulationTime() + sim.getSimulationTimeStep()) -- Explicitly handle the graph (allows reset)

end -----[ CALCULATE RIGHT ERROR ]-----

do -----[ APPLY PID GAINS TO WHEEL MOTORS ]-----

-- Set the motor velocity of the robots right wheel to the default velocity added with the calculated PID gains

rightWheelVelocity = defaultWheelVelocity +

(gainProportionalPID \* errorRight) + -- Calculate the proportional gain

(gainIntegralPID \* (errorRightAccumulated / integralThresholdPID)) + -- Calculate the integral gain

(gainDerivativePID \* (errorRightPreviousPID - errorRightCurrentPID)) -- Calculate the derivative gain

leftWheelVelocity = defaultWheelVelocity -- Set the motor velocity of the robots left wheel to the default velocity

sim.setJointTargetVelocity(motorLeftWheel, leftWheelVelocity) -- Set the left wheel motor velocity of the robot to the calculated velocity

sim.setJointTargetVelocity(motorRightWheel, rightWheelVelocity) -- Set the right wheel motor velocity of the robot to the calculated velocity

end -----[ APPLY PID GAINS TO WHEEL MOTORS ]-----

do -----[ CONSOLE OUTPUT ]-----

if (errorRight > 0) then -- If the right error is larger than '0' (too close to the object), do the following

printf("Speed [" .. string.format("%.2f", robotMovementSpeed) .. " m/s] Heading [" .. string.format("%.2f", robotHeadingOutput) .. " DEG] "

.. "Position [" .. string.format("%.2f", robotPositionCurrent[1]) .. ", " .. string.format("%.2f", robotPositionCurrent[2]) .. "] "

.. "Detection Left [" .. objectDetectedLeftString .. "] " .. "Detection Right [" .. objectDetectedRightString .. "] "

.. "State [Edge Following] Right -----> Turning Outwards [RMSE " .. string.format("%.5f", rootMeanSquaredErrorPID) .. "]") -- Output the robot entered the 'turning outwards' state

else -- If the right error is smaller than '0' (too far from the object), do the following

printf("Speed [" .. string.format("%.2f", robotMovementSpeed) .. " m/s] Heading [" .. string.format("%.2f", robotHeadingOutput) .. " DEG] "

.. "Position [" .. string.format("%.2f", robotPositionCurrent[1]) .. ", " .. string.format("%.2f", robotPositionCurrent[2]) .. "] "

.. "Detection Left [" .. objectDetectedLeftString .. "] " .. "Detection Right [" .. objectDetectedRightString .. "] "

.. "State [Edge Following] Right -----> Turning Inwards [RMSE " .. string.format("%.5f", rootMeanSquaredErrorPID) .. "]") -- Output the robot entered the 'turning inwards' state

end -- End condition

end -----[ CONSOLE OUTPUT ]-----

end -- End condition

elseif (edgeFollowingTransition == true) then -- If the robot has been set to transition into edge following, do the following

if (edgeFollowingTransitionReset == false) then -- If the edge following transition variables have not been reset, do the following

edgeFollowingTransitionReset = true -- The edge following transition variables will be reset

do -----[ RESET EDGE FOLLOWING VARIABLES }-----

errorRightCountPID = 0 -- Reset the number of right errors obtained

errorRightCounterPID = 1 -- Reset the current right error counter

errorRightSumPID = { } -- Reset the right error array

accumulatedRightErrorRMSE = 0 -- Reset the accumulated right RMSE value

errorLeftCountPID = 0 -- Reset the number of left errors obtained

errorLeftCounterPID = 1 -- Reset the current left error counter

errorLeftSumPID = { } -- Reset the left error array

accumulatedLeftErrorRMSE = 0 -- Reset the accumulated left RMSE value

sim.resetGraph(graphRMSE) -- Reset the RMSE graph data plots

sim.resetGraph(graphSetPoint) -- Reset the set point graph data plots

end -----{ RESET EDGE FOLLOWING VARIABLES }-----

end -- End condition

frontleftMostSensorDetectionStatus, frontLeftMostSensorDistance = sim.readProximitySensor(sonarSensor[1]) -- Store object detection and object distance (left-most sensor)

frontLeftSensorDetectionStatus, frontLeftSensorDistance = sim.readProximitySensor(sonarSensor[3]) -- Store object detection and object distance (front-left sensor)

frontRightMostSensorDetectionStatus, frontRightMostSensorDistance = sim.readProximitySensor(sonarSensor[8]) -- Store object detection and object distance (left-most sensor)

frontRightSensorDetectionStatus, frontRightSensorDistance = sim.readProximitySensor(sonarSensor[6]) -- Store object detection and object distance (front-right sensor)

if (edgeFollowingTransitionLeft == false and edgeFollowingTransitionRight == false) then -- If the robot has not been assigned a direction to transition into edge following, do the following

if (frontLeftSensorDetectionStatus > 0 and frontRightSensorDetectionStatus > 0) then -- If the robots front-most sensors detect objects, do the following

if (frontleftMostSensorDetectionStatus > 0 and frontRightMostSensorDetectionStatus > 0) then -- If the robots front-most side sensors detect objects, do the following

closestObjectLeft = frontLeftSensorDistance + frontLeftMostSensorDistance -- Store and accumulate the distances detected on the robots front-left most and side most sensors

closestObjectRight = frontRightSensorDistance + frontRightMostSensorDistance -- Store and accumulate the distances detected on the robots front-right most and side most sensors

if (closestObjectLeft < closestObjectRight) then -- If the robot detects object(s) closer to its left side, do the following

edgeFollowingTransitionLeft = true -- The robot will transition into edge following for its left side

edgeFollowingTransitionString = "Sensors [SIDE AND FRONT] ---> Space Right" -- Set the edge following transition string to 'space right' (objects further on right)

elseif (closestObjectRight < closestObjectLeft) then -- If the robot detects object(s) closer to its right side, do the following

edgeFollowingTransitionRight = true -- The robot will transition into edge following for its right side

edgeFollowingTransitionString = "Sensors [SIDE AND FRONT] ---> Space Left" -- Set the edge following transition string to 'space left' (objects further on left)

elseif (closestObjectLeft == closestObjectRight) then -- If the robot detects object(s) at equal lengths around its front side, do the following

transitionDirection = math.random(1, 2) -- Generate a randomised direction for the robot to transition into edge following

if (transitionDirection == 1) then -- If the transition direction is '1', do the following

edgeFollowingTransitionLeft = true -- The robot will transition into edge following for its left side

else -- If the transition direction is '2', do the following

edgeFollowingTransitionRight = true -- The robot will transition into edge following for its right side

end -- End condition

edgeFollowingTransitionString = "Sensors [SIDE AND FRONT] ---> Equal Distance (Random)" -- Set the edge following transition string to 'equal distance' (object detected equal lengths)

end -- End condition

elseif (frontleftMostSensorDetectionStatus > 0 and frontRightMostSensorDetectionStatus == 0) then -- If the robots front-left side sensor detects an object but the front-right side sensor does not detect an object, do the following

edgeFollowingTransitionLeft = true -- The robot will transition into edge following for its left side

edgeFollowingTransitionString = "Sensors [SIDE AND FRONT] ---> Non-detection Right" -- Set the edge following transition string to 'non-detection right' (object not detected right)

elseif (frontleftMostSensorDetectionStatus == 0 and frontRightMostSensorDetectionStatus > 0) then -- If the robots front-right side sensor detects an object but the front-left side sensor does not detect an object, do the following

edgeFollowingTransitionRight = true -- The robot will transition into edge following for its right side

edgeFollowingTransitionString = "Sensors [SIDE AND FRONT] ---> Non-detection Left" -- Set the edge following transition string to 'non-detection left' (object not detected left)

else -- If the robot does not detect an object on either of its front-most side sensors (object directly ahead), do the following

transitionDirection = math.random(1, 2) -- Generate a randomised direction for the robot to transition into edge following

if (transitionDirection == 1) then -- If the transition direction is '1', do the following

edgeFollowingTransitionLeft = true -- The robot will transition into edge following for its left side

edgeFollowingTransitionString = "Sensors [FRONT] ---> Equal Distance (Random)" -- Set the edge following transition string to 'equal distance' (object detected equal lengths)

else -- If the transition direction is '2', do the following

edgeFollowingTransitionRight = true -- The robot will transition into edge following for its right side

edgeFollowingTransitionString = "Sensors [FRONT] ---> Equal Distance (Random)" -- Set the edge following transition string to 'equal distance' (object detected equal lengths)

end -- End condition

end -- End condition

elseif (frontRightSensorDetectionStatus > 0 and frontLeftSensorDetectionStatus == 0) then -- If the robot detects an object on its front-right sensor but not its front-left sensor, do the following

edgeFollowingTransitionRight = true -- The robot will transition into edge following for its right side

edgeFollowingTransitionString = "Sensors [FRONT] ---> Non-detection Front-left" -- Set the edge following transition string to 'non-detection front-left' (object not detected front-left)

elseif (frontLeftSensorDetectionStatus > 0 and frontRightSensorDetectionStatus == 0) then -- If the robot detects an object on its front-left sensor but not its front-right sensor, do the following

edgeFollowingTransitionLeft = true -- The robot will transition into edge following for its left side

edgeFollowingTransitionString = "Sensors [FRONT] ---> Non-detection Front-right" -- Set the edge following transition string to 'non-detection front-right' (object not detected front-right)

else -- If the robot does not detect an object using its front-most sensors, do the following

do -----[ CHECK ALL SENSORS }-----

leftSensorDistance = 0 -- Create and intialise a variable for storing the accumulated distance of the robots left sensors that detected an object

for i = 1, 4, 1 do -- For all of the robots sonar sensors, do the following

detectionStatus, distanceReturned = sim.readProximitySensor(sonarSensor[i]) -- Store object detection and object distance for the currently iterated sensor

if (detectionStatus > 0) then -- If the currently iterated sonar sensor detected an object, do the following

leftSensorDistance = leftSensorDistance + distanceReturned -- Add and equal the left sensor distance to the distance detected by the currently iterated sensor

end -- End condition

end -- End iteration

for i = 12, 16, 1 do -- For all of the robots sonar sensors, do the following

detectionStatus, distanceReturned = sim.readProximitySensor(sonarSensor[i]) -- Store object detection and object distance for the currently iterated sensor

if (detectionStatus > 0) then -- If the currently iterated sonar sensor detected an object, do the following

leftSensorDistance = leftSensorDistance + distanceReturned -- Add and equal the left sensor distance to the distance detected by the currently iterated sensor

end -- End condition

end -- End iteration

rightSensorDistance = 0 -- Create and intialise a variable for storing the accumulated distance of the robots right sensors that detected an object

for i = 5, 12, 1 do -- For all of the robots sonar sensors, do the following

detectionStatus, distanceReturned = sim.readProximitySensor(sonarSensor[i]) -- Store object detection and object distance for the currently iterated sensor

if (detectionStatus > 0) then -- If the currently iterated sonar sensor detected an object, do the following

rightSensorDistance = rightSensorDistance + distanceReturned -- Add and equal the right sensor distance to the distance detected by the currently iterated sensor

end -- End condition

end -- End iteration

end -----[ CHECK ALL SENSORS }-----

if (leftSensorDistance < rightSensorDistance) then -- If the accumulated distance detected on the robots left sensors is smaller than the accumulated distance detected on the robots right sensors, do the following

edgeFollowingTransitionLeft = true -- The robot will transition into edge following for its left side

edgeFollowingTransitionString = "Sensors [ALL] ---> Right Corner" -- Set the edge following transition string to 'right corner' (object corner to right)

elseif (rightSensorDistance < leftSensorDistance) then

edgeFollowingTransitionRight = true -- The robot will transition into edge following for its right side

edgeFollowingTransitionString = "Sensors [ALL] ---> Left Corner" -- Set the edge following transition string to 'left corner' (object corner to left)

end -- End condition

end -- End condition

elseif (edgeFollowingTransitionLeft == true) then -- If the robot is transitioning into edge following for its left side, do the following

sim.setJointTargetVelocity(motorLeftWheel, defaultWheelVelocity) -- Set the left wheel motor velocity of the robot to the default velocity

sim.setJointTargetVelocity(motorRightWheel, -(defaultWheelVelocity)) -- Set the right wheel motor velocity of the robot to the default velocity, negated

do -----[ CONSOLE OUTPUT ]-----

printf("Speed [" .. string.format("%.2f", robotMovementSpeed) .. " m/s] Heading [" .. string.format("%.2f", robotHeadingOutput) .. " DEG] "

.. "Position [" .. string.format("%.2f", robotPositionCurrent[1]) .. ", " .. string.format("%.2f", robotPositionCurrent[2]) .. "] "

.. "Detection Left [" .. objectDetectedLeftString .. "] " .. "Detection Right [" .. objectDetectedRightString .. "] "

.. "State [Edge Following] Transition Left -----> " .. edgeFollowingTransitionString) -- Output the robot entered the 'transition' state

end -----[ CONSOLE OUTPUT ]-----

elseif (edgeFollowingTransitionRight == true) then -- If the robot is transitioning into edge following for its right side, do the following

sim.setJointTargetVelocity(motorLeftWheel, -(defaultWheelVelocity)) -- Set the left wheel motor velocity of the robot to the default velocity, negated

sim.setJointTargetVelocity(motorRightWheel, defaultWheelVelocity) -- Set the right wheel motor velocity of the robot to the default velocity

do -----[ CONSOLE OUTPUT ]-----

printf("Speed [" .. string.format("%.2f", robotMovementSpeed) .. " m/s] Heading [" .. string.format("%.2f", robotHeadingOutput) .. " DEG] "

.. "Position [" .. string.format("%.2f", robotPositionCurrent[1]) .. ", " .. string.format("%.2f", robotPositionCurrent[2]) .. "] "

.. "Detection Left [" .. objectDetectedLeftString .. "] " .. "Detection Right [" .. objectDetectedRightString .. "] "

.. "State [Edge Following] Transition Right -----> " .. edgeFollowingTransitionString) -- Output the robot entered the 'transition' state

end -----[ CONSOLE OUTPUT ]-----

end -- End condition

end -- End condition

end -- End function