De Montfort University

IMAT 3404

Mobile Robotics

Implementation of a Robot Controller

Pamela Hardaker and Elizabeth Felton

Authored by

Adam L. Hubble

P17175774
P17175774@my365.dmu.ac.uk

Abstract

This report explores the application of an autonomous robot controller for the Pioneer 3-DX mobile robot, in focus of the controller's architecture, exhibiting behaviours in the form of actuation and their related strategies for enabling the robot to solve complex predefined tasks. Throughout this document, the development of the robot's controller will be detailed and where necessary, justified; there will be particular emphasis on the controller's behavioural strategies and the robots resulting actuation, this will be evidenced with relevant testing.

Introduction

Autonomous systems within robotics allows "tasks to be performed without the requirement for human intervention" [1], autonomy within system control flow can be achieved by implementing "sets of parameters" [1] or states in the form of a finite-state machine (FSM) for example; this enables robots to "decide and act" [1] on its own accord, which corresponds with Robin Murphy's "hybrid deliberative/ reactive paradigm" [2], for a robot to plan, sense and then actuate.

Available in appendix A

Autonomous robots are considered to be capable of "collecting information from its surroundings, to perceive its surrounding environment, to localize itself, to decide on actions and to execute the necessary actuation" [1]. In the context of the robots behavioural requirements: to avoid, edge follow and map obstacles in its environment, as well as to roam its environment arbitrarily, the robot would be considered autonomous; given that its behaviours could be invoked interchangeably in result of self-governance. Thereby in the proceeding sections of this document, the outlined behavioural requirements of the robot will be discussed as separate strategies, as well as their bound architecture that allows said autonomy to be achieved.

Avoidance Strategy

Robot avoidance is a "crucial behaviour for numerous robotic systems"[4], whereby a robot should be able to avoid "obstacles within its workspace"[4]; the configuration for this behaviour was designed for enabling the robot to evade objects in a range of scenarios and environments, that it may be subjected to. Respectfully, not only does the robot adjust its position and orientation based upon the positions of detected objects, it is also able to navigate into and out of confined spaces, as well as to reverse away from objects that it detects ahead; for the actuations listed, the ultrasonic sensors equipped by the Pioneer 3-DX robot are utilised.

In relation to the strategy's fundamental behaviour, Valentino Braitenberg's avoidance algorithm [5] was adapted for the simplicity of setup and transitioning to and from the robot traversing forwards or backwards and turning left or right; the algorithms application was particularly useful for the Pioneer 3-DX robot, in which the readings from the robots sensors could be applied to "directly affect its movement" [5], for evasive purposes. These behaviours are invoked when an object is detected, and the robot is neither reversing nor turning away from an object. For the implementation of these behaviours, it was only necessary for the robot's front-facing sensors to be considered, given that the robot traverses forwards mostly.

Available in appendix B

When deciding the direction to turn to, all of the robot's front-facing sensor readings are accumulated for the left and right sides of the robot and are then compared in relation to their magnitude; the robot will turn to the direction of where objects are situated further away (lower magnitude), this enables the robot to proportionally manoeuvre away from nearby objects, successfully.

Available in appendix C

Furthermore, given the scenario that multiple objects are detected at equal distances from the robot, provided that the front-most sensors do not detect an object at similar distances, the robot will traverse forwards. Whereas if an object is detected in the facing direction of the robot using the front-most sensors and the difference in distance between the sensor readings is smaller than '0.005' metres, the robot will enter the subsidiary state 'reversing', required that the robot is not within the subsidiary states 'turning' or 'stuck'. This configuration is necessary for preventing the robot from oscillating and eventually colliding with an object when an insignificant difference between the sensor readings is calculated. Upon entering the state, the robot will reverse away from the object until it is no longer detectable, or alternatively when another object is detected by any of the back-facing sensors; this enables the robot to avoid objects positioned behind it, as it attempts to evade objects in front.

In advancement of 'reversing', the robot then transitions to the subsidiary state 'turning', where it will turn for '1' or '2' seconds randomly, in a calculated or randomised direction; similar to prior calculations, the robot turns in the direction of where objects are situated further away, but considers the back-facing sensor readings to determine the direction also. If the difference between the accumulated distance detected by the sensors on each side of the robot is equal, the robot's direction is randomised binarily to avert oscillation and eventual collision, once more.

Available in appendix D

Accounting for the readings of all sensors enables the robot to consider object positions in a multidirectional manner, in the scenario that objects surround it, the robot will be able to further transition to the subsidiary state 'stuck', whereby the robot will pivot around its own axis until sensors four and five do not detect an object. Upon exiting the 'turning' state, as the robot remains 'stuck', the robot will be able to emerge from the space where no object resides using the Braitenberg avoidance algorithm. Entering the 'stuck' state requires the robot to return six or more sensor readings with a distance metric, otherwise the robot will simply turn for the randomly selected amount of time. Turning aims to prevent the robot from re-entering the 'reversing' state and allows the robot to exit the 'avoiding' state when no objects are detected. Meanwhile, the 'stuck' state exists to enable the robot to explore and map its environment entirely, without colliding and entrapping itself; the values assigned to the Braitenberg 'noDetectionDistance' and 'maxDetectionDistance' variables, allow for this.

Wandering Strategy

Robot exploration is "crucial for achieving tasks such as environment modelling, target searching and auto navigation" [7], wandering as a form of exploration aims to make the robots "unknown surrounding space" [7], known. For which, the configuration proposed for this behaviour attempts to enable the robot to explore its environment, in a "series of continuous movements" [7]; forwards, left and right.

For maximising the area explored, the robot navigates itself into the unexplored areas of an environment, in a randomised order; this is regulated as the initial wandering behaviour of the robot, which prioritises mapping and exploration efficiency. Implementing the navigation strategy required the robot's environment to be sectioned into areas, each assigned a position relative to the global coordinate space. The robot traverses forwards and rotates towards its targeted area, by calculating the angular difference between its position and the areas position; the facing direction of the robot is also accounted for and is used for determining the accumulated angle that it has rotated for. Upon all areas being explored, the robots wandering strategy invokes random traversal behaviours, for the purpose of exploring sub-sections of areas that may not be known to the robot.

Relating to said behaviours, for forward traversal, the robot travels for a randomised distance ranging between '0.1' and '0.5' meters and is calculated by comparing the magnitude of the distance between the robots current and previous positions. Travelling short distances allows the robots exploration to be more varied, as its frequency of sideward traversal increases exponentially; this assumes that a larger area of an environment can be explored within a given period of time, as "shorter travelled paths provide more area coverage" [8].

Regarding sideward traversal, the robot adjusts its facing direction until the angular difference between the robots initial and current heading accumulates a randomly generated angle, ranging between '30' and '90' degrees. Adjusting to the range provided enables the robot to turn away from its current area, whilst preventing it from overturning into the same area. Due to the obscurity of CoppeliaSim's object orientation layout, the angle the robot accumulates is calculated similarly to the distance it has travelled, but in the context of heading adjustment. To sustain angle accumulation, all negative orientations of the robot are negated.

Available in appendix E

Relating to the direction nominated for sideward traversal, a number is randomised between the range of '1' and '100', using the modulo operator, if the number selected is even the robot will rotate right for the given angle, vice versa; this range supports increased variation in the direction chosen.

Edge Following Strategy

Edge following is a significant behaviour of mobile robots when considering environment mapping; following the edges of objects allows mapping to be conducted more efficiently, as a robot attempts to maintain a detectable range to objects, rather than avoiding them entirely. For implementing this behaviour, the Pioneer 3-DX equips a proportional-integral-derivative (PID) controller for maintaining a set distance to objects (set-point), that the robot considers followable. The application of a PID controller is well adapted for mobile robot motion control, given its providance for "smoothness, performance and accuracy" [9] in navigation; when considering the Pioneer 3-DX robot, the availability of ultrasonic sensors and it being a "wheeled robot" [9], allows its "motor drives" [9] to be controlled for said navigation to be achieved.

Relating to the configuration set, the Pioneer 3-DX is capable of following the edges of objects detected to the left or right of its body, the robot employs its left-most, right-most, front-left and front-right sensors to achieve edge following behaviours; all of the sensors specified only reside in the front face of the robot, due to the robot only traversing forwards when edge following.

Available in appendix F

Logically, when the robot detects an object from one of its side-most sensors and its corresponding front sensor has not detected an object, the robot will enter the ‘edge following’ state; this enables the robot to transition between the ‘avoiding’ and ‘edge following’ states seamlessly, as when a front-facing sensor other than the side-most sensor detects an object, the controller will invoke the avoidance strategy.

Available in appendix G

In focus of the accuracy of edge following, with the PID controller integrated the robot is able to approximately maintain the set-point (‘0.25’ metres) from object edges overtime; the controllers set-point and maximum distance values have been configured similarly to the Braitenberg avoidance non-detection and maximum detection values, in addition, the controller samples the most recent ‘10’ errors to achieve such.

Available in appendix H

Importantly, the robot cannot follow the edges of objects detected on both of its sides simultaneously. However, as previously mentioned for the robot’s avoidance strategy, the robot would traverse forwards in the scenario presented, thereby resulting in similar mapping output to edge following.

Environment Mapping

Environment mapping is well purposed for “acquiring a global overview map that integrates all of the data collected by the robot” [10], to determine the layout of “an unknown terrain” and whether all of the “regions have been searched” [10] for the area of the terrain. The result, being a visual representation of the “robots’ environment from a top-view perspective”, can be achieved offline or in “real-time” [10]; relating to the controllers configuration, a series of offline and online maps are computed autonomously from the robots ultrasonic sensor readings, this aims to illustrate the arrangement of the robot’s environment disparately.

For map building and calculations, see ***appendix I***.

Behavioural Control

Architecture Choice

Preliminarily, the robot controller’s architecture was decided in advance of implementing the robot’s behavioural strategies, this enabled a “basic control system to be established” [11], which proved to be essential for navigating the implementation of the robot’s behaviours. In result of implementing the behaviours detailed in this document, the robot now demonstrates “levels of competence” [11] in the forms of obstacle avoidance, edge following and mapping, as well, the robot can explore an unknown environment arbitrarily. When relating to the design of the system, it was intended for the system to resemble Rodney Brooks’ “subsumption architecture” [11], for the purpose of incremental development and achieving system robustness; being a “control system with increasing levels of competence” [11], achieves this.

Implementing the behaviours architecturally was achieved via a series of globally defined Boolean variables, acting as behavioural states or “modules” [11] of the Pioneer 3-DX robot; the application of Boolean variables was particularly useful for “inhibiting” [11] behaviours, which forms the basis of state transition and actuation invocation for the robot. Inhibiting behaviours was favoured over “suppressing behaviours” [11] for logic comprehension and increased computational performance aims. Relating to priority, avoidance displays the highest precedence over any other behaviour, proceeded by edge following and then wandering; this configuration considers evasive behaviours as the most significant.

Available in appendix J

For determining the active state of the robot, all of the state variables are compared to decide which actuation and corresponding behaviours are invoked. The structure used to control this flow of execution, can be considered a series of finite-state machines (FSM’s), which benefit system robustness and seamless behavioural transitioning, due to “low processor overhead” [12].

Available in appendix K

Testing

In the determination of the controller’s final configuration, each of the robot’s behaviours were calibrated for accomplishing compatibility in a range of environments; this process required the robot’s actuation variables to be adjusted and then observed from CoppeliaSim’s console and simulation windows. This was necessary for evaluating the robots resulting behaviours and preventing unexpected or inappropriate actuation where applicable; it is inevitable that the test cases created in this instance, have bettered the robot controller for all of its behaviours present.

For the entire testing regime, see ***appendix L***.

Conclusions

Summarising the controller configured, the Pioneer 3-DX robot exhibits behavioural competence for avoiding, edge following and mapping obstacles in a chosen environment, as well as within the exploration of it. In result of testing the final configuration, the robot is proven to behave as expected from an observational and programmatical standpoint and can therefore be regarded as successful, in the context of this domain.

Acknowledgements

Without the supervision and support of Dr. Pamela Hardaker and Ms. Elizabeth Felton, this project’s development may not have climaxed to the standard presented, thank you. Also, without the physical and virtual facilities, alongside the academic support provided by De Montfort University (Leicester) at the time of this pandemic, this project may never have been completed. I thank those who are responsible, equally.

Bibliography

[1] THONDIYAH, A. (2016) Autonomy for Robots: Design and Developmental Challenges. In: *3rd International Conference on Innovations in Automation and Mechatronics Engineering ICIAME 2016, Chennai, February 2016.* Amsterdam: Elsevier LTD, pp. 4-6

[2] MURPHY, R.R. (2000) Introduction to AI Robotics. [Online] Available from: https://www.researchgate.net/publication/238699045_Introduction_to_AI_Robotics

[3] MURPHY, R.R. (2000) Introduction to AI Robotics. [Diagram] In: MURPHY, R.R. (2000) *Introduction to AI Robotics.* Massachussets: MIT Press, p. 260

[4] KELASIDI, E. and MOE, S. and PETTERSEN, K.Y. and KOHL, A.M. and LILJEBACK, P. and GRAVDAHL, J.T. (2019) Path Following, Obstacle Detection and Obstacle Avoidance for Thrusted Underwater Snake Robots. [Online] Available from: https://www.frontiersin.org/articles/10.3389/frobt.2019.00057/full

[5] GOCHEV, I. and NADZINSKI, G. and STANKOVSKI, M. (2017) Path Planning and Collision Avoidance Regime for a Multi-Agent System in Industrial Robotics. *Machines. Technologies. Materials.* [Online] Available from: https://www.semanticscholar.org/paper/PATH-PLANNING-AND-COLLISION-AVOIDANCE-REGIME-FOR-A-Gochev-Nadzinski/69faf19aaf406796377141ed6ba76a4dd6641f23

[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.* Chausseestraße: ResearchGate, pp. 43-49.

[9] LEKKALA, K.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

[10] LAKAEMPER, R. and LATECKI, L..J. and SUN, X. and WOLTER, D. (2005) Geomtric Robot Mapping. In: *Discrete Geometry for Computer Imagery, 12th International Conference, DGCI 2005, Poitiers*. Chausseestraße: ResearchGate, pp. 11-22.

[11] 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.* Chausseestraße: ResearchGate, pp. 225-236.

[12] BANYASAD, O. and COX, P.T. (2008) Visual Programming of Subsumption - Based Reactive Behaviour. *International Journal of Advanced Robotic Systems.* [Online] 5(4). Available from: https://www.researchgate.net/publication/221787373_Visual_Programming_of_Subsumption_-_Based_Reactive_Behaviour

Appendices

Appendix A:

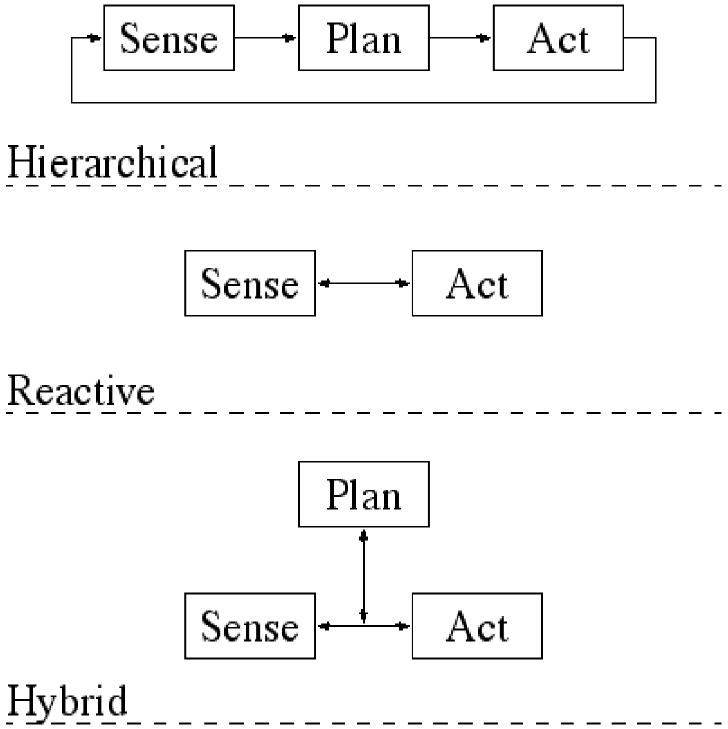

Figure 1: Robin Murphy's behaviour architecture paradigms [4]

Appendix B:

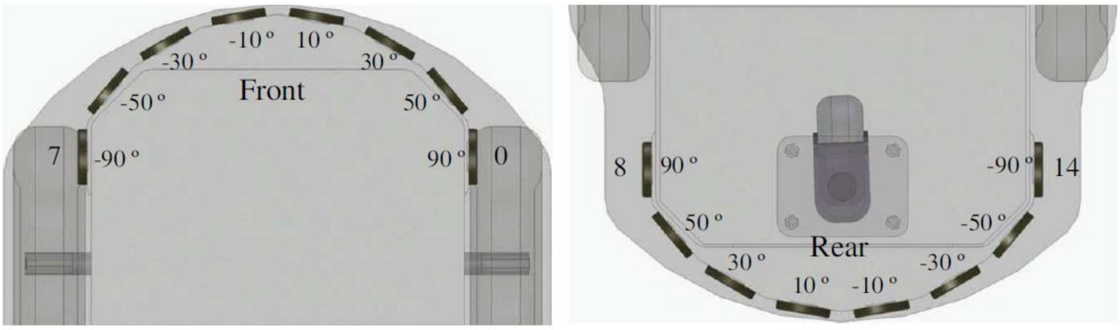

Figure 2: Pioneer 3-DX ultrasonic sensor alignment [6]

Appendix C:

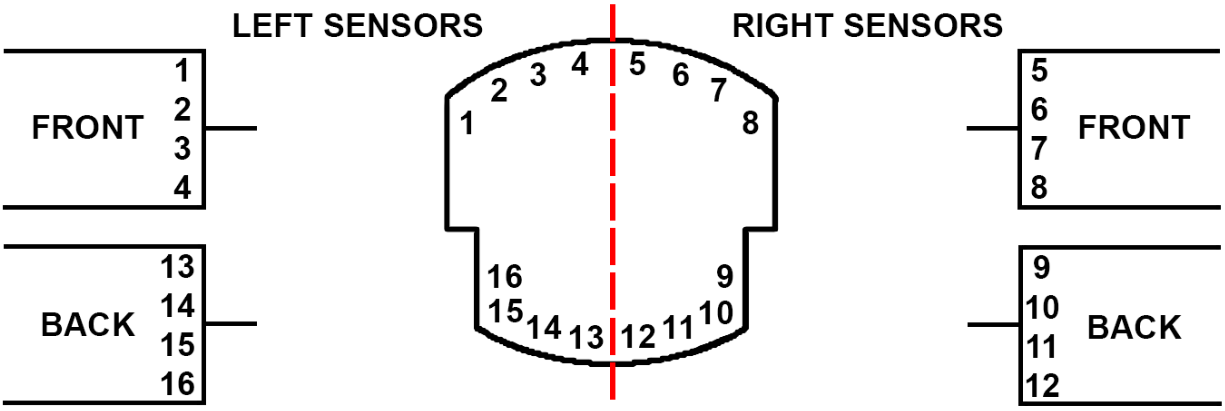

Figure 3: Pioneer 3-DX ultrasonic sensors sectioned

Appendix D:

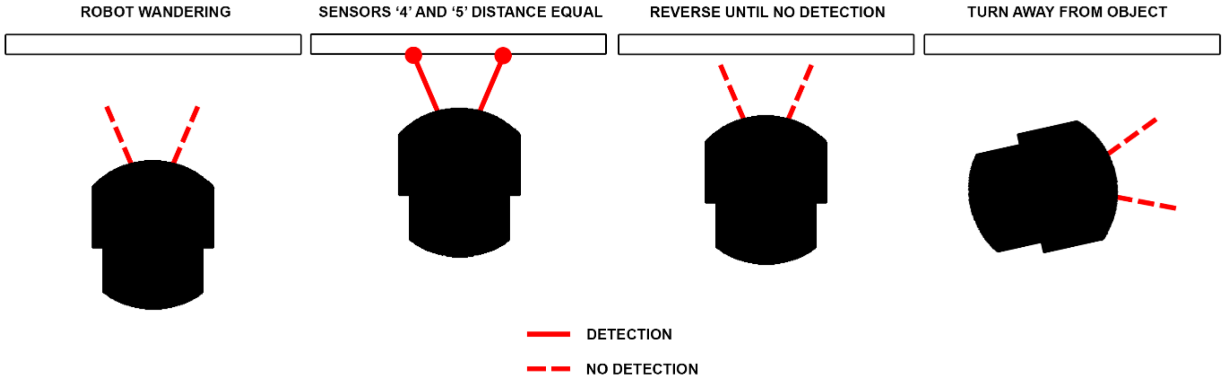

Figure 4: Avoiding state, subsidiary state transition, Pioneer 3-DX robot 'reversing' to robot 'turning'

Appendix E:

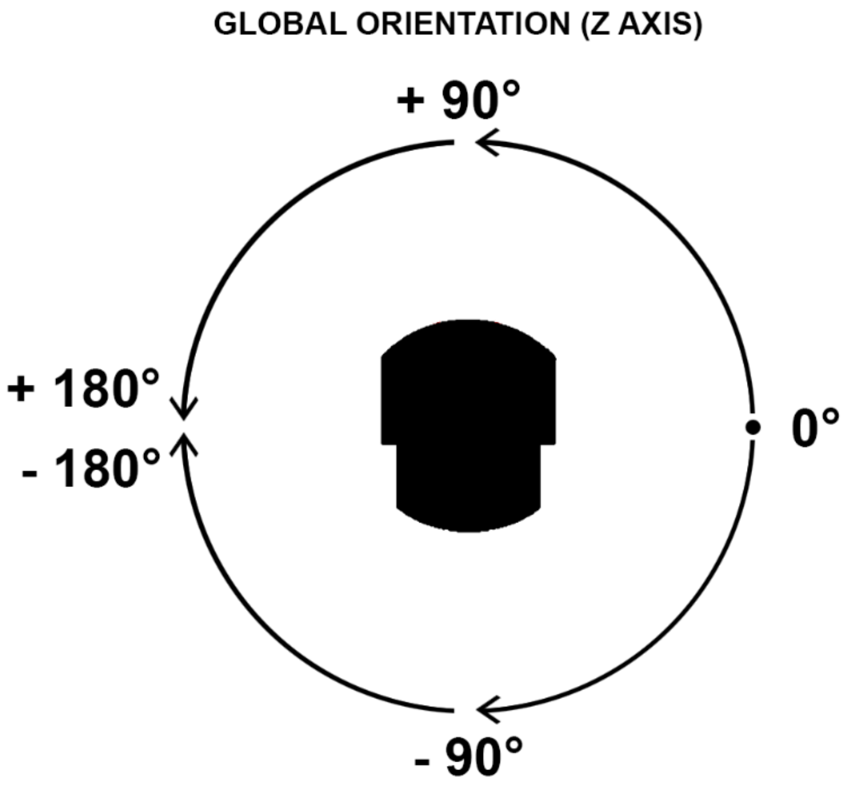

Figure 5: CoppeliaSim, global orientation in the Z axis, also known as the heading

Appendix F:

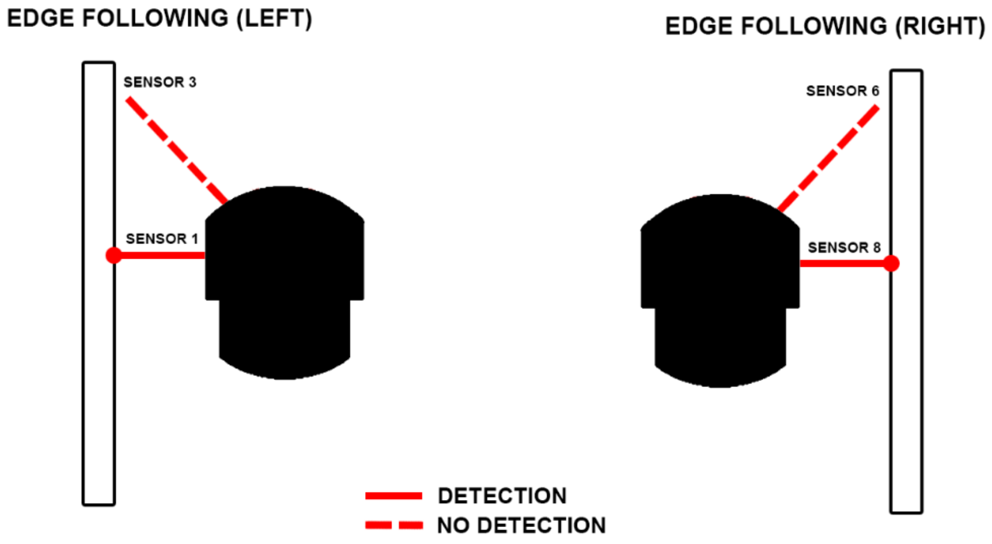

Figure 6: Pioneer 3-DX ultrasonic sensors used to edge follow objects on both sides of the robot's body

Appendix G:

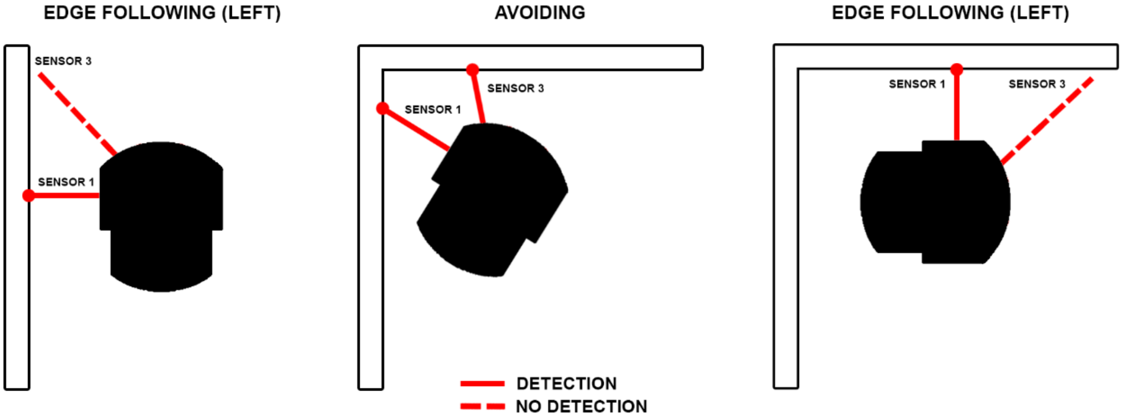

Figure 7: Pioneer 3-DX, state transition, robot 'edge following' to 'avoiding' and back to 'edge following'

Appendix H:

Avoiding Variables		*Edge Following Variables*	
noDetectionDistance (metres)	0.34	setPoint (metres)	0.25
maxDetectionDistance (metres)	0.2	maxDistance (metres)	0.275

Table 1: Pioneer 3-DX 'avoiding' and 'edge following' state variables, showing similarity

Appendix I:

For preserving the accuracy of data collected about the robot's environment, it was necessary for the controllers mapping strategy to be executed throughout the entirety of the robots tasking, to ensure that all of the objects detected by the robot, were localized and recorded concurrently for building the series of maps intended. Unlike the robots behavioural-based strategies, environment mapping was neither inhibited nor suppressed.

Relating to the online map configuration, map construction is achieved by using plots that represent the positions of where objects have been detected and the path the robot has taken whilst being tasked in the environment. Plots are represented within a graph object, appearing as an undocked window within CoppeliaSim's interface; the plots are appended to the map in real-time and are colour coordinated to differentiate between the robot's path and the positions of detected objects. For the calculations involved in the map's construction, when objects are detected, their position is determined by translating the robot's sensor readings into the simulators global coordinate space. Amongst these calculations, the difference in position between the robot and the sensor that has detected an object, is passed into the rotation matrix in attempt to populate plots with a linear alignment; this aims to prevent the straight edges of objects from being misrepresented as curves.

In continuation of online mapping calculations, upon a plots position being translated to the global coordinate space, it is then rounded to the nearest decimal place so that it can be aligned to a plane, that represents the entire area of the environment. Rounding was necessary as a data validation strategy, whereby a point detected by one sensor can only be populated in the map when another sensor has detected it, in the same frame; this is used to verify whether an objects position has or has not been miscalculated, given the lack of reliability for sonar reception. Without rounding the position detected, the same point would not likely be detected by another sensor in the same frame and therefore it would not be populated in the map, this is due to the fidelity of CoppeliaSim's coordinate system. Proceeding on from the maps calculations, the plots are then populated in the map if the conditions were met and are not otherwise; the positions of each plot populated in the map, are also stored into a two-dimensional array for the use of an offline map.

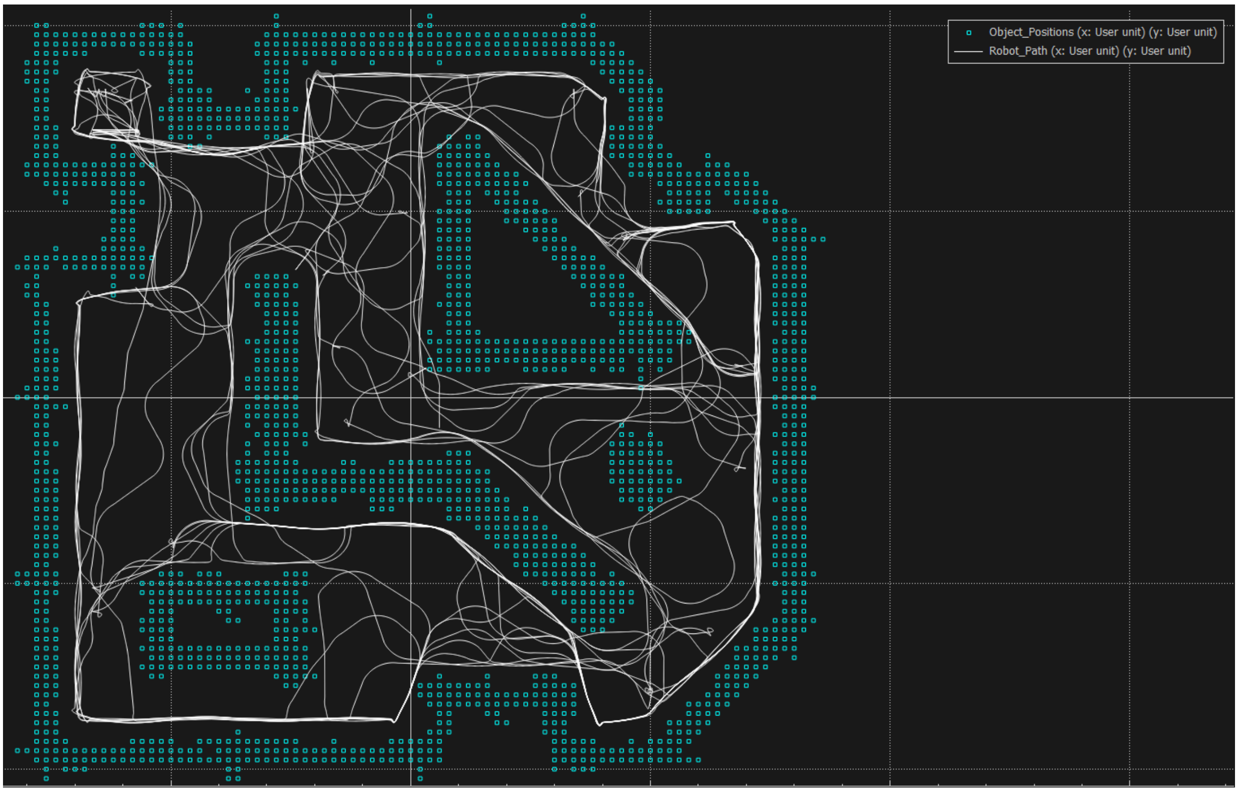

Figure 8: CoppeliaSim graph object, online (real-time) map, detected object positions and robot pathing

Regarding offline map construction, similarly to the online mapping method, object positions are detected, translated, and validated. However, rather than handling the positions of detected objects, counters are used alternatively, for indicating the number of times that a position has been detected. This serves as another data validation technique to provide certainty for an object's existence, the counters are stored in a two-dimensional array also, where the indexes of the array determine the coordinate for the position detected.

In use of the array populated for the offline map, a Microsoft Excel Comma Separated Values File (CSV) is handled by CoppeliaSim in the 'sysCall_cleanup()' method, for writing all of the counters in the array, to file. When the writing process finishes, a Microsoft Excel workbook establishes a connection with the file handled by CoppeliaSim and imports all of the data into a section of cells, that have been conditionally format by colour. This attempts to display the borders of objects that the robot has encountered, in a combination of colours; a key is provided for acknowledging the conditions that each colour represents. As the connection between the files is re-established upon opening the file and via periodic updates, the offline map can be considered automated for renewing mapping data and representing it.

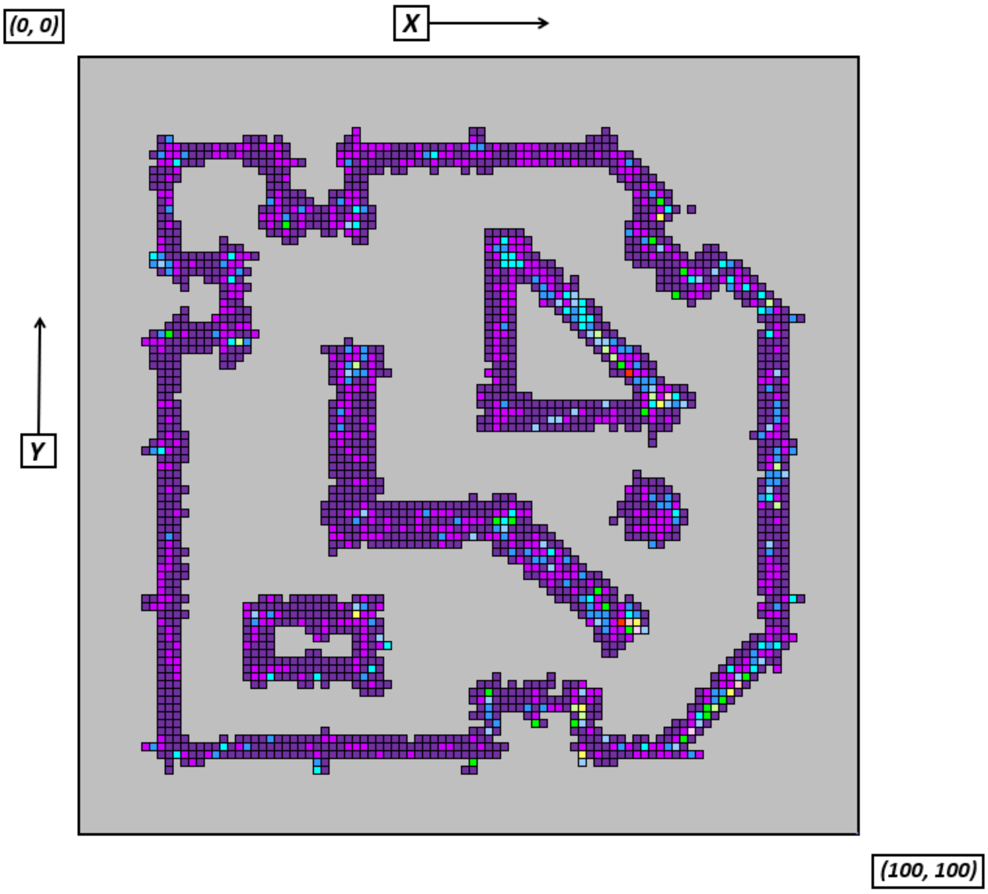

Figure 9: Microsoft Excel workbook, offline map, detected object positions colour coordinated for number of detections

In addition to point-based maps, another offline map has been configured for its methods supposed accuracy and determination of inlying and outlining data, from a given data model. The algorithm used for the provided reasons is Random Sample Consensus (RANSAC), which has proven to be worthy for determining the positions and principal geometry of objects in an environment; this is realised from the use of line-based illustration. For populating the map, as the algorithm handles vectors, it adopts the object position array that is generated by the online mapping method. When the robot's assignment is complete, the arrays size is determined and used to section the array of coordinates equally. This ensures that a line of best fit will be determined and drawn for each section, as the minimum number of points per section allowed is '3', whereas the maximum is '12'. For this range, the algorithm only considers an even number of positions detected, if not, a neutral position is appended to the end of the array; this is necessary for assuring that an equal number of coordinates is distributed to each of the sections.

As an algorithm, RANSAC is used to recursively compare the number of agreeing points in a section, with a line that is formed by randomly selecting two points from the equivalent section. For a point to be considered agreeing with a line, its Euclidean distance must measure below '1'; this was set to improve the selection of points, that make up the line of best fit for the current section. In relation to the algorithms end condition, it is expected that the algorithm iterates over all of the points within each section '1000' times, before submitting the lines of best fit for each section that exists. Such magnitude enables the lines of best fit to be thoroughly considered and well purposed for representing the robot's environment.

Upon the algorithms completion, a CSV file is handled by CoppeliaSim in the ‘sysCall_cleanup()’ method, where all of the lines of best fit are written, to file; each line is stored as a set of points for the simplicity of representing the lines in an external application, as opposed to a complete line equation. For which, the points stored within the CSV file are then read-in to a Microsoft Visual Studio solution (SLN) that is SFML enabled and are then drawn as a series of lines in a graphically-dependant window. Multiple windows have been configured to present the results of RANSAC and are colour coordinated for differentiating between the lines that do and do not represent objects in the robot’s environment; this is not entirely accurate, however.

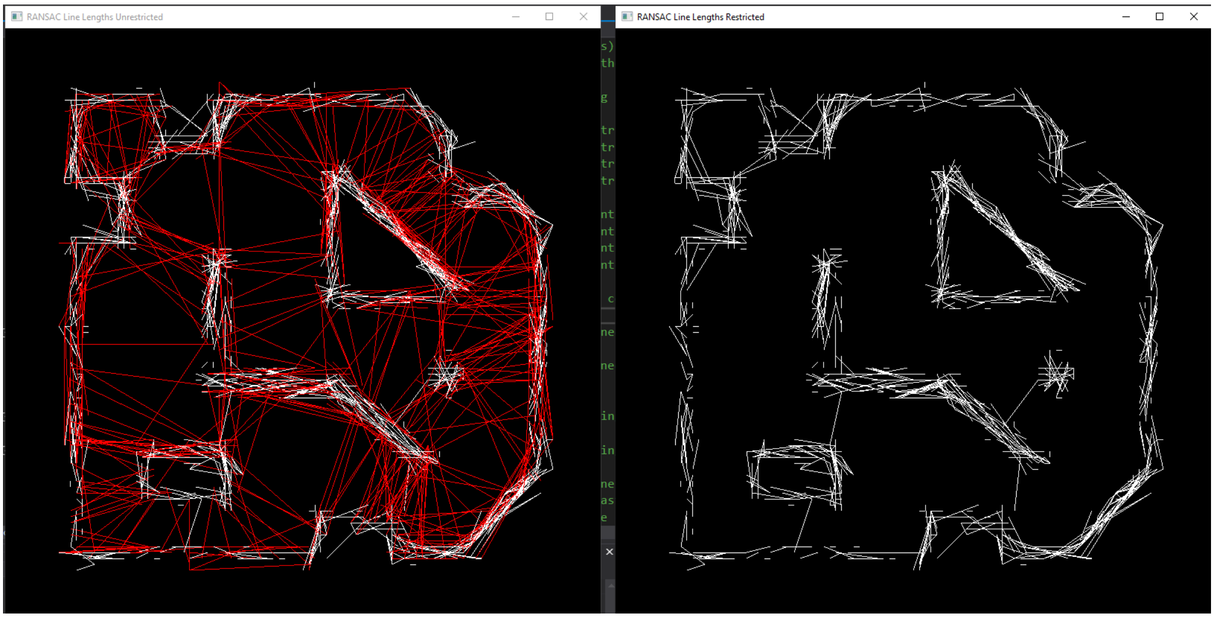

Figure 10: Microsoft Visual Studio solution, offline map, lines of best fit RANSAC output, all lines vs validated lines in separate SFML windows

Appendix J:

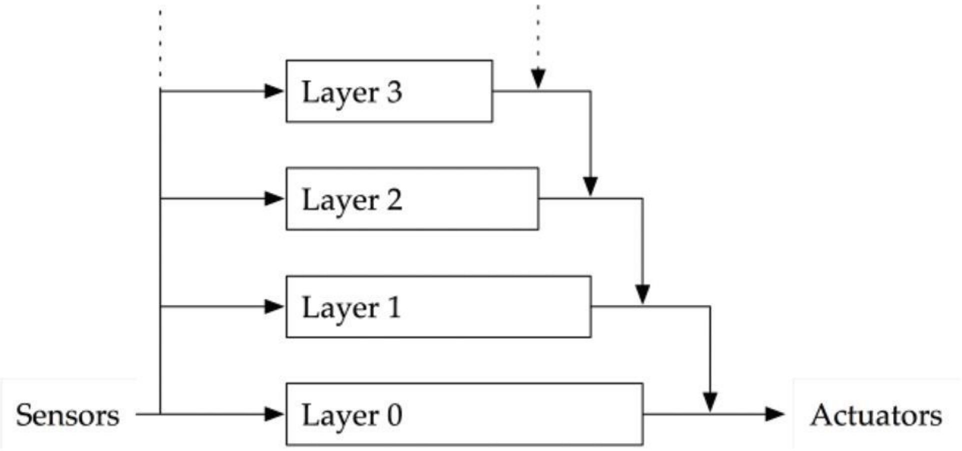

Figure 11: Rodney Brooks’, subsumption architecture control model [11]

Appendix K:

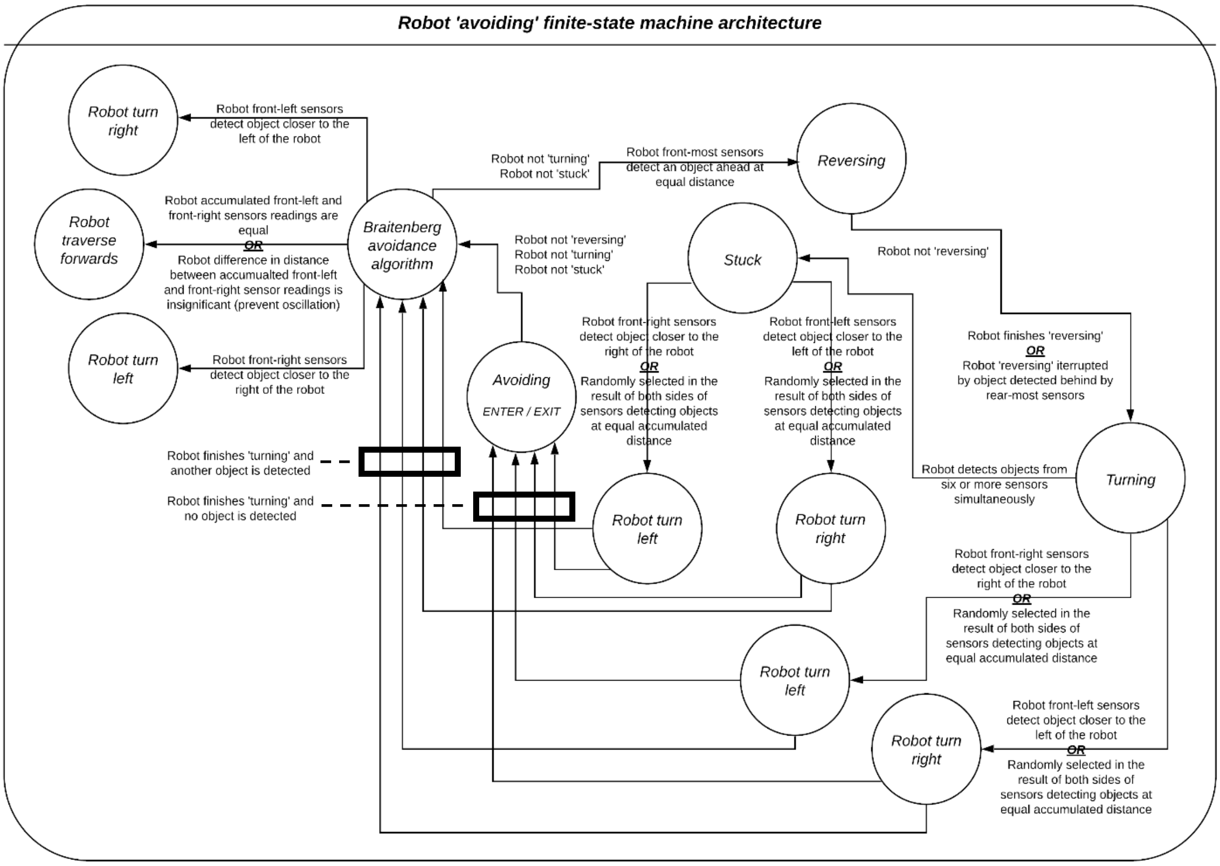

Figure 12: Robot controller, robot 'avoiding' finite-state machine architecture

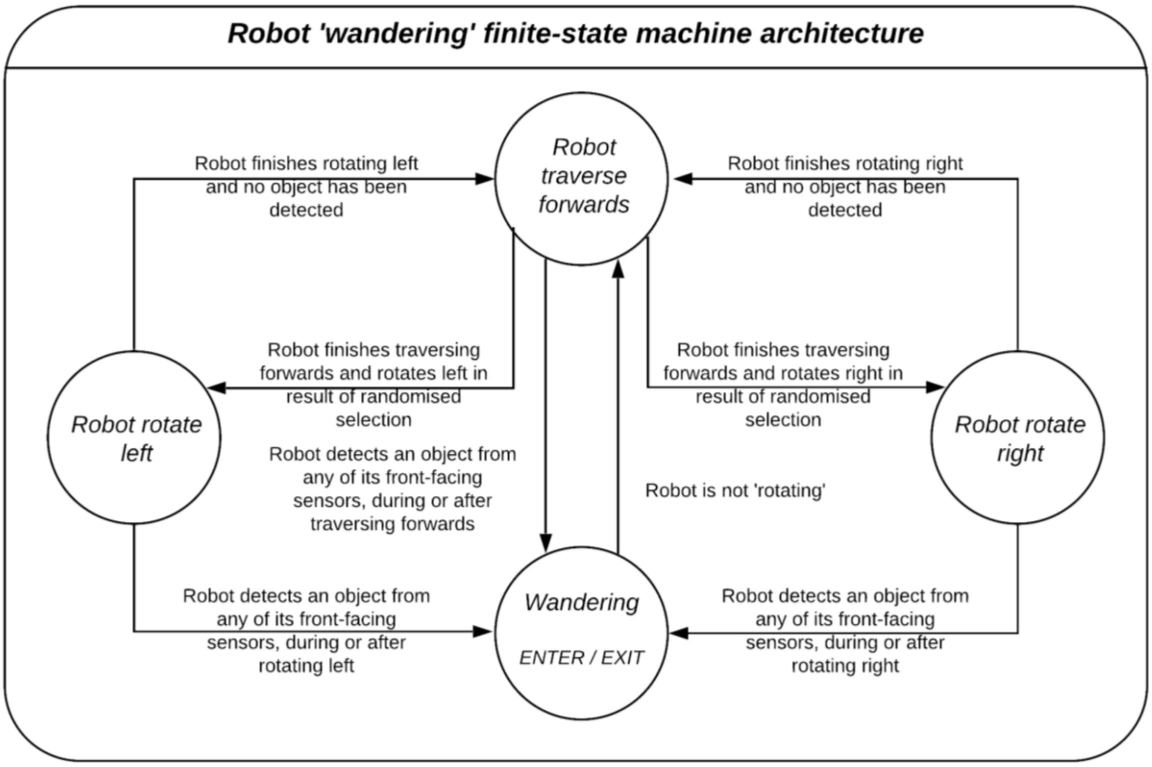

Figure 13: Robot controller, robot ‘wandering’ finite-state machine architecture

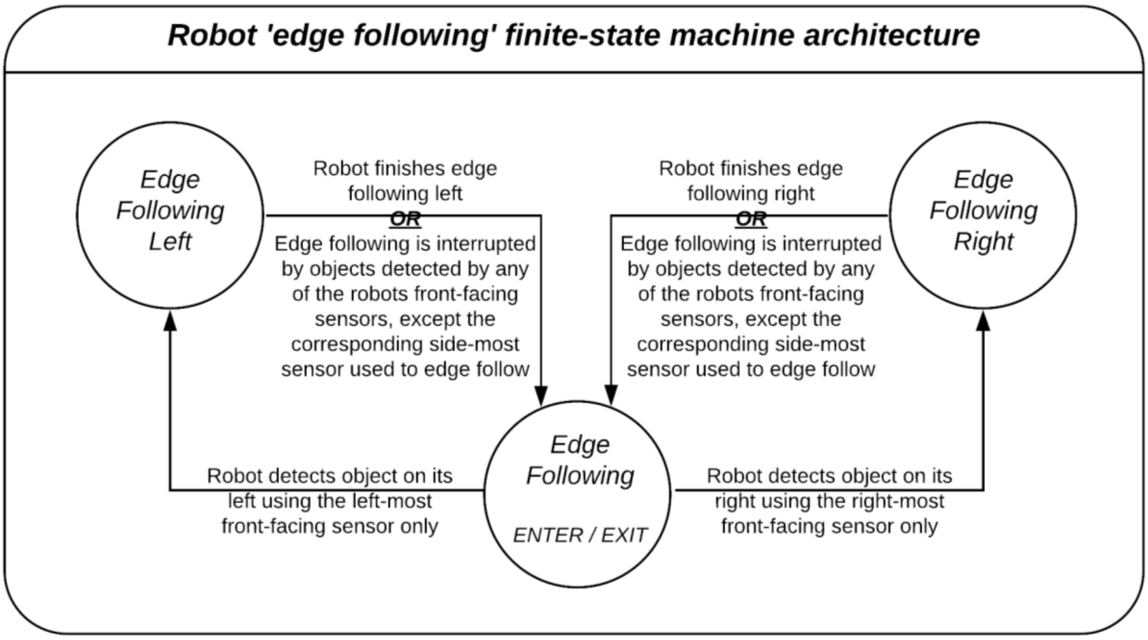

Figure 14: Robot controller, robot 'edge following' finite-state machine architecture

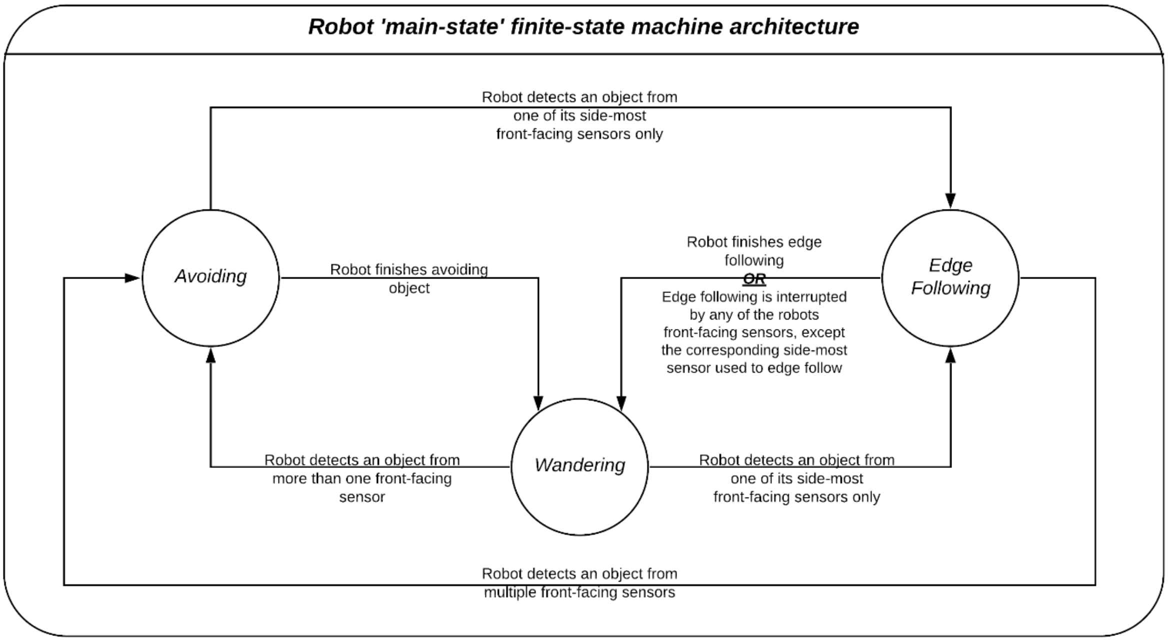

Figure 15: Robot controller, robot 'main-state' finite-state machine architecture

Appendix L:

Testing Regime

Featured below, exists the test cases that were undertaken for the robots ‘avoiding’ state, in attempt to establish the best configuration for each variable specified; the test cases relate closely to the Braitenberg avoidance adaptation for the avoidance strategy, being the most significant technique for object evasion. Relating to the cases summarily, the desired outcome was for the robot to not exhibit oscillatory motions when detecting objects from multiple sensors simultaneously, to not collide with objects and to be able to roam areas or spaces of its environment that are considered confined.

Robot Avoiding, Valentino Braitenberg Avoidance Algorithm				
Case	**Variables**	**Values**	**Observations**	**Implemented**
1	noDetectionDistance maxDetectionDistance	0.5 0.2	Robot oscillates severely when objects are detected on either side of its body, robot does not collide with objects, robot cannot exist in confined spaces	No
2	noDetectionDistance maxDetectionDistance	0.4 0.3	Robot oscillates severely when objects are detected on either side of its body, robot does not collide with objects, robot can exist in relatively confined spaces	No
3	noDetectionDistance maxDetectionDistance	0.4 0.2	Robot oscillates noticeably when objects are detected on either side of its body, robot does not collide with objects, robot can exist in confined spaces	No
4	noDetectionDistance maxDetectionDistance	0.3 0.2	Robot does not oscillate when objects are detected on either side of its body, robot does not collide with objects, robot can exist in confined spaces	No
5	noDetectionDistance maxDetectionDistance	0.3 0.1	Robot does not oscillate when objects are detected on either side of its body, robot collides with objects often, robot can exist in confined spaces	No
6	noDetectionDistance maxDetectionDistance	0.25 0.2	Robot does not oscillate when objects are detected on either side of its body, robot collides with objects rarely, robot can exist in confined spaces	No
7	noDetectionDistance maxDetectionDistance	0.375 0.2	Robot does not oscillate when objects are detected on either side of its body, robot does not collide with objects, robot can exist in confined spaces	No
8	noDetectionDistance maxDetectionDistance	0.35 0.2	Robot does not oscillate when objects are detected on either side of its body, robot does not collide with objects, robot can exist in confined spaces	No
9	noDetectionDistance maxDetectionDistance	0.325 0.2	Robot does not oscillate when objects are detected on either side of its body, robot does not collide with objects, robot can exist in confined spaces (too close to objects)	No
10	noDetectionDistance maxDetectionDistance	0.34 0.2	Robot does not oscillate when objects are detected on either side of its body, robot does not collide with objects, robot can exist in confined spaces (desired distance from objects)	Yes

Table 2: Robot 'avoidance', Braitenberg avoidance algorithm variable test cases

In conclusion of this investigation, it is obvious that the behaviours exhibited from the testing, succeed the behavioural requirements and intentions of the Braitenberg avoidance adaptation. Relating to the final variable values nominated, many of the values configured in the test cases prior were sufficient for exhibiting the desired behaviours, however, for minimising the distance between the robot and objects it detects, they evidently were not. This differentiation has enabled the robot to achieve exploration in confined spaces and improve the quality of environment mapping.

In continuation of testing avoidance behaviours, the test cases advancing this passage investigate the configuration of the robots 'turning' subsidiary state, which is invoked upon the robot exiting the 'reversing' subsidiary state and the robot being in dispute of its turning direction. Testing the ranges used to determine the turning actuation of the Pioneer-3DX robot, was essential for the robot preventing itself from re-entering the 'reversing' subsidiary state, given the scenario that the robot merely turns away from an object. When applying the scenario to the robot, it would be expected of the robot to iteratively reverse and traverse forwards (wandering); such behaviour would increase the probability of collision, which is an undesired behaviour of the robot that this research aims to prevent.

Robot Avoiding, Turning After Reversing Direction and Time Ranges for Randomisation

Case	**Variables**	**MIN**	**MAX**	**Observations**	**Implemented**
1	reverseTurnTimer	1	3	Robot turns after reversing, robot turns in a random direction, robot turns further than required to prevent the robot from re-transitioning to the 'reversing' state	No
	rotationDirection	1	2		
2	reverseTurnTimer	1	2	Robot turns after reversing, robot turns in a random direction, robot turns sufficiently to prevent itself re-transitioning to the 'reversing' state, robot does not display variation in turning duration however	No
	rotationDirection	1	2		
3	reverseTurnTimer	0.1	2	Robot turns after reversing, robot turns in a random direction, robot often turns inadequately and re-transitions to the 'reversing' state multiple times, robot displays variation in turning duration however	No
	rotationDirection	1	2		
4	reverseTurnTimer	0.5	2	Robot turns after reversing, robot turns in a random direction, robot turns sufficiently to prevent itself from re-transitioning to the 'reversing' state and demonstrates variation in turning duration also	Yes
	rotationDirection	1	2		

Table 3: Robot 'avoidance', robot turning after reversing direction and time ranges for randomisation, test cases

In accordance to the results presented, the investigation has enabled the robot to prevent itself from re-transitioning to the 'reversing' subsidiary state, iteratively; the probability of collision has been significantly reduced in result of this configuration, as well, the efficiency of environment exploration and mapping has bettered also.

In the proceeding table, belongs the test cases conducted for the robots 'edge following' state, more specifically the configuration for the PID controller gain variables. In mention of the behaviours that were intended for edge following, these test cases aimed to determine the better situated gain variable values, for the robot to follow the entirety of object edges that were not considered small and spherical, or arched; oscillatory motions were undesired also, as similarly discussed for the 'avoidance' strategy.

Robot Edge Following, PID Controller (gain variables)

Case	**Variables**	**Values**	**Observations**	**RMSE**	**Implemented**
1	proportionalGain	15	Robot maintains the set-point to the edges of objects, robot follows small spherical objects, robot does not oscillate on enter or exit	0.00001	No
	integralGain	5			
	derivativeGain	0.1			

2	proportionalGain	10	Robot maintains the set-point to the edges of objects, robot follows small spherical objects, robot does not oscillate on enter or exit	0.00002	No
	integralGain	5			
	derivativeGain	0.1			
3	proportionalGain	5	Robot maintains the set-point to the edges of objects, robot does not edge follow small spherical objects, robot does not oscillate on enter or exit	0.00052	No
	integralGain	5			
	derivativeGain	0.1			
4	proportionalGain	7	Robot maintains the set-point to the edges of objects, robot follows small spherical objects, robot does not oscillate on enter or exit	0.00011	No
	integralGain	5			
	derivativeGain	0.1			
5	proportionalGain	8	Robot maintains the set-point to the edges of objects, robot follows small spherical objects rarely, robot does not oscillate on enter or exit	0.00007	No
	integralGain	5			
	derivativeGain	0.1			
6	proportionalGain	7	Robot maintains the set-point to the edges of objects, robot follows small spherical objects, robot does not oscillate on enter or exit	0.00001	No
	integralGain	10			
	derivativeGain	0.1			
7	proportionalGain	7	Robot maintains the set-point to the edges of objects, robot follows small spherical objects, robot does not oscillate on enter or exit	0.00002	No
	integralGain	7			
	derivativeGain	0.1			
8	proportionalGain	7	Robot maintains the set-point to the edges of objects, robot follows small spherical objects, robot does not oscillate on enter or exit	0.00002	No
	integralGain	6			
	derivativeGain	0.1			
9	proportionalGain	7	Robot maintains the set-point to the edges of objects, robot does not edge follow small spherical objects, robot does not oscillate on enter or exit	0.00016	No
	integralGain	4			
	derivativeGain	0.1			
10	proportionalGain	7	Robot maintains the set-point to the edges of objects, robot follows small spherical objects, robot does not oscillate on enter or exit	0.0006	No
	integralGain	4.5			
	derivativeGain	0.1			
11	proportionalGain	7	Robot maintains the set-point to the edges of objects, robot follows small spherical objects, robot does not oscillate on enter or exit	0.00013	No
	integralGain	4.25			
	derivativeGain	0.1			
12	proportionalGain	7	Robot maintains the set-point to the edges of objects, robot follows small spherical objects, robot does not oscillate on enter or exit	0.00007	No
	integralGain	4			
	derivativeGain	1			
13	proportionalGain	7	Robot maintains the set-point to the edges of objects, robot follows small spherical objects, robot does not oscillate on enter or exit	0.0009	No
	integralGain	4			
	derivativeGain	0.5			
14	proportionalGain	7	Robot maintains the set-point to the edges of objects, robot follows small spherical objects, robot does not oscillate on enter or exit	0.00014	No
	integralGain	4			
	derivativeGain	0.25			
15	proportionalGain	7	Robot maintains the set-point to the edges of objects, robot does not edge follow small spherical objects, robot does not oscillate on enter or exit	0.00011	Yes
	integralGain	4			
	derivativeGain	0.2			

Table 4: Robot 'edge following', PID controller gain variable test cases

Conclusively, the results from the investigation prove that the intended behaviours for edge following have been achieved, whereby, the robot is now able to follow the edges of objects that are not considered small and spherical, and without displaying oscillatory motions. In result of this configuration, the robot can explore environments more efficiently and prevent occurrences of collision when transitioning to ‘edge following’.

Furthermore, in the table below, underlines the configuration for the PID controllers distance variables, which are implemented within the robots ‘edge following’ strategy also. The distance variable values were primarily investigated for enabling the robot to edge follow effectively and smoothly in result of state transitioning; also, it was desired for the robot to not attach itself to the edges of small and spherical, or arched objects and to be situated closer to the edges of objects, whilst not exhibiting oscillation. These expectations purposed to better the robot's environment exploration and mapping efficiency. For the existence of these test cases, the final configuration for the robot's 'avoidance' strategy was implemented, due to the 'noDetectionDistance' variable being a conditional barrier to the invocation of edge following behaviours.

Robot Edge Following, PID Controller (distance variables) – restrained by robot 'avoidance' 'noDetectionDistance' variable, final configuration for 'avoidance' considered for these test cases

Case	Variables	Values	Observations	Implemented
1	setPoint	0.34	Robot momentarily edge follows an object before exiting, robot transitions to edge following smoothly as it oscillates insignificantly, robot does not attach itself to spherical or arched object edges, does not allow oscillation also	No
	maxDistance	0.34		
2	setPoint	0.3	Robot edge follows an object for a short duration before exiting, robot transitions to edge following smoothly as it oscillates insignificantly, robot does not attach itself to spherical or arched object edges, allows for small oscillation also	No
	maxDistance	0.34		
3	setPoint	0.25	Robot edge follows an object entirely for the length of the objects edge before exiting, robot transitions to edge following smoothly, allows for large oscillations however	No
	maxDistance	0.34		
4	setPoint	0.25	Robot edge follows an object entirely for the length of the objects edge before exiting, robot transitions to edge following smoothly, allows for large oscillation however	No
	maxDistance	0.3		
5	setPoint	0.25	Robot edge follows an object entirely for the length of the objects edge before exiting, robot transitions to edge following smoothly as it oscillates insignificantly, robot does not attach itself to spherical or arched object edges, allows for small oscillation also (desired to be closer to objects)	Yes
	maxDistance	0.275		
6	setPoint	0.2	Robot edge follows an object entirely for the length of the objects edge, robot transitions to edge following roughly as it oscillates frequently, robot attaches itself to every spherical and arched object edge, robot rarely detaches from arched object edge, allows for relatively large oscillation also	No
	maxDistance	0.275		
7	setPoint	0.225	Robot edge follows an object entirely for the length of the objects edge, robot transitions to edge following roughly as it oscillates frequently, robot attaches itself to every spherical and arched object edge, robot rarely detaches from arched object edge, allows for relatively large oscillation also	No
	maxDistance	0.275		
8	setPoint	0.23	Robot edge follows an object entirely for the length of the objects edge, robot transitions to edge following roughly as it oscillates frequently, robot attaches itself to every spherical and arched object edge, robot rarely detaches from arched	No
	maxDistance	0.275		

			object edge, allows for relatively large oscillation also	
9	setPoint	0.24	Robot edge follows an object entirely for the length of the objects edge, robot transitions to edge following roughly as it oscillates frequently, robot attaches itself to every spherical and arched object edge, robot rarely detaches from arched object edge, allows for relatively large oscillation also	No
	maxDistance	0.275		
10	setPoint	0.245	Robot edge follows an object entirely for the length of the objects edge before exiting, robot transitions to edge following smoothly as it oscillates unnoticeably, robot attaches itself to every spherical and arched object edges minimally, allows for small oscillation also	No
	maxDistance	0.275		

Table 5:Robot 'edge following', PID controller distance variable test cases

Regarding the outcomes of this investigation, the robot's ability to edge follow has improved from the findings of the preceding investigation alone. The values nominated for this configuration allows the robot to transition to the 'edge following' state seamlessly, this has been achieved by eliminating oscillatory actuation of the robot, during its transition to the state and there on adjustment to the set-point. Moreover, the robot can also follow the entirety of objects that are not considered small and spherical, or arched and within close proximity; configuring the behaviours in such way has enhanced the efficiency of environment exploration and mapping once more.

In regard to wandering, it was intended for the Pionner-3DX robot to traverse forwards, leftwards and rightwards in the form of random "continuous movements" [7]; this purposed for the robot to display efficient environment exploration and mapping behaviours. Thereby a series of test cases has been created, to exercise the sensibility of the values for the relevant variables, in attempt to enable the robot to explore its environment unguided and arbitrarily.

Robot Wandering, Forward and Sideward Traversal Ranges for Randomisation

Case	Variables	MIN	MAX	Observations	Implemented
1	wanderingForwardDistance	1	5	Robot does not enter 'rotation' states as forward distance is never achieved in small environment, robot traverses' forwards however	No
	rotationDirection	1	2		
	wanderingTurnAngle	1	360		
2	wanderingForwardDistance	1	3	Robot traverses forward, robot rotates left and right randomly and rarely, robot rotates right mostly, robot rotates far and traverses into area explored prior	No
	rotationDirection	1	2		
	wanderingTurnAngle	1	360		
3	wanderingForwardDistance	1	2	Robot traverses forward, robot rotates left and right randomly and often, robot rotates right mostly, robot rotates far and traverses into area explored prior	No
	rotationDirection	1	2		
	wanderingTurnAngle	1	360		
4	wanderingForwardDistance	0.1	1	Robot traverses forward, robot rotates left and right randomly and regularly, robot rotates right often, robot rotates far and traverses into area explored prior	No
	rotationDirection	1	2		
	wanderingTurnAngle	1	360		
5	wanderingForwardDistance	0.1	0.5	Robot traverses forward, robot rotates left and right randomly and frequently, robot rotates right often, robot rotates	No
	rotationDirection	1	2		
	wanderingTurnAngle	1	360		

				far and traverses into area explored prior	
6	wanderingForwardDistance	0.1	0.5	Robot traverses forward, robot rotates left and right randomly and frequently, robot rotates right often, robot merely rotates	No
	rotationDirection	1	2		
	wanderingTurnAngle	1	90		
7	wanderingForwardDistance	0.1	0.5	Robot traverses forward, robot rotates left and right randomly and frequently, robot rotates right often, robot rotates similarly for each iteration	No
	rotationDirection	1	2		
	wanderingTurnAngle	45	90		
8	wanderingForwardDistance	0.1	0.5	Robot traverses forward, robot rotates left and right randomly and frequently, robot rotates right often, robot rotates with increased variation in angle, robot explores more unvisited areas in environment	No
	rotationDirection	1	2		
	wanderingTurnAngle	30	90		
9	wanderingForwardDistance	0.1	0.5	Robot traverses forward, robot rotates left and right randomly and frequently, robot rotates with increased variation in direction and angle, robot explores more unvisited areas in environment	No
	rotationDirection	1	10		
	wanderingTurnAngle	30	90		
10	wanderingForwardDistance	0.1	0.5	Robot traverses forward, robot rotates left and right randomly and frequently, robot rotates with plentiful variation in direction and angle, robot explores more unvisited areas in environment more frequently	Yes
	rotationDirection	1	100		
	wanderingTurnAngle	30	90		

Table 6: Robot 'wandering', forward and sideward traversal ranges for randomisation, test cases

Relating to the desired behaviours for the robot when 'wandering', the results from the test cases support that the robot has been configured to traverse forwards, left and right, for a randomly selected distance and angle, within sensible ranges. In which, the robot demonstrates competence in turning in a randomly selected direction, for a randomly selected angle. Undoubtedly the robot explores unvisited areas or spaces in an environment more frequently, which suffices for the purpose of environment exploration and mapping efficiency, yet again.

In further regard to the robots wandering behaviours, it was primarily desired for the robot to explore all of the unknown spaces in its environment arbitrarily, before transitioning to its random traversal sequence. This particular strategy for behavioural invocation aimed for the robot to explore and map its environment as quickly and efficiently as possible. For assuring the structure in the robot's exploration pattern and transition to random traversal, a series of scenarios was created and tested against the robot's actuation observed.

Robot Wandering, Movement Adjustments for Exploring Unknown Areas

Case	Scenario	Expectations	Successful
1	Robot is assigned a target area of the environment to explore, the area that the robot is assigned to explore is the top-left region of the robot's environment	Robot rotates towards the position of the target area when within the 'wandering' state, robot turns towards the position in the direction that accumulates the least angle	Yes
2	Robot is assigned a target area of the environment to explore, the area that the robot is assigned to explore is the top-left region of the robot's	Robot rotates towards the position of the target area when within the 'wandering' state, robot turns towards the position in the direction that accumulates the least angle, robot situates within	Yes

	environment, the robot enters the area it is assigned to explore	the desired proximity of the position assigned overtime as it maintains its facing direction to the target, robot then marks the area as explored	
3	Robot is assigned a target area of the environment to explore, the area that the robot is assigned to explore is the bottom-right region of the robot’s environment, the robot encounters many objects upon its target area being set, robot moves further away from the target position as evasive behaviours are invoked	Robot avoids and follows the edges of objects until reaching space where no objects are detected by the robot, robot rotates towards the position of the target area when transitioned to the ‘wandering’ state, robot turns towards the position in the direction that accumulates the least angle, robot maintains the assignment of the current target area until marked as explored, robot situates within the desired proximity of the position assigned overtime as it maintains its facing direction to the target, robot then marks the area as explored	Yes
4	Robot is assigned a target area of the environment to explore, the area that the robot is initially assigned to explore is the top-right region of the robot’s environment, following the areas exploration the robot is then assigned to explore the bottom-left region of its environment	Robot rotates towards the position of the target area when within the ‘wandering’ state, robot turns towards the position in the direction that accumulates the least angle, robot situates within the desired proximity of the position assigned overtime as it maintains its facing direction to the target, robot then marks the area as explored, robot does not exhibit random traversal behaviours, robot repeats process of exploration for the bottom-left region until being within the desired proximity of the position assigned, robot marks the area as explored also	Yes
5	Robot is assigned a target area of the environment to explore, the area that the robot is assigned to explore is the top-right region of the robot’s environment, upon entering and nearing the target position assigned, the robot invokes other behaviours due to objects detected nearby, the robot exists the target area from the resulting actuation	Robot rotates towards the position of the target area when within the ‘wandering’ state, robot turns towards the position in the direction that accumulates the least angle, robot repeats process iteratively when transitioning to the ‘wandering’ state and maintains its facing direction to the target position, robot renters the target area overtime, robot resides in the desired proximity of the position assigned, robot marks the area as explored	Yes
6	Robot is assigned all of the possible target areas that are unknown overtime, the robot encounters many objects overtime that adjusts the robot’s orientation, the angular difference between the robots facing direction and target position changes relatively	Robot rotates towards the position of the current target area when within the ‘wandering’ state, robot turns towards the position in the direction that accumulates the least angle, always	Yes
7	Robot is assigned all off of the possible target areas that are unknown, overtime the robot explores all of the areas assigned	Robot traverses’ forwards which is later followed by sideward traversal when within the ‘wandering’ state, robot traverses randomly and does not adjust its orientation or maintain its facing direction towards a known target area	Yes
8	Robot is assigned all off of the possible target areas that are unknown overtime, robot is assigned a target area randomly for each time the currently assigned area is explored	Robot rotates towards the position of the current target area when within the ‘wandering’ state, robot turns towards the position in the direction that accumulates the least angle, robot situates within the desired proximity of the position	Yes

	[Tested many times]	assigned overtime as it maintains its facing direction to the target, robot then marks the area as explored, robot does not exhibit random traversal behaviours, robot repeats process for all other target areas selected, robot does not demonstrate a recognisable pattern in its actuation for the assignment of different target areas	

Table 7: Robot 'wandering', adjustments made within movement for exploring unknown areas of an environment, test cases

From the results of the scenario-driven test cases, the configuration of the robot when exploring unknown areas of an environment, appears to be well established and integrated, when also considering the random traversal behaviours of the robots ‘wandering’ strategy. It is more so evident that the controller enables the robot to explore unknown areas of an environment efficiently and arbitrarily, which supports its purpose for being autonomous. Without doubt, the configuration presented for the robots wandering strategy allows the robot to be increasingly independent of human intervention, for the basis of exploration and mapping.

For the array of behaviours tested independently of each other, the behavioural expectations of the robot when subjected to an unknown environment was in need of testing also, to resolve any discrepancies within state transitions and related actuations; this was considered possible by invoking behaviours incorrectly in the result of main-state (avoiding, wandering and edge following) conditions being setup differently, to how they were intended to be. Thereby in the following study, the robots “levels of competence” [10] are exercised for the purpose of determining the controller’s suitability, in the domain of autonomy. For such, the final configuration for each behaviour was implemented when undergoing the following tests.

Robot Avoiding, Wandering and Edge Following (robot autonomy within an unknown environment)			
Case	**Scenario**	**Expectations**	**Successful**
1	Robot wanders into a wall that is positioned directly in front of its front-most facing sensors and detects the object using both sensors at relatively similar distances	Robot reverses from the wall until it no longer detects the object, the robot then turns away from the wall to prevent itself from reversing again, before it re-transitions to ‘wandering’ again	Yes
2	Robot wanders into a wall that is positioned in front of its body, but is angled when made relative to the robots facing direction	Robot gradually avoids the wall until the robots corresponding side-most sensor on the robot’s front face is the only sensor detecting the wall, the robot then transitions to follow the entire length of the current wall, before re-transitioning to ‘avoiding’	Yes
3	Robot wanders into a wall that is positioned in front of its body, but is angled when made relative to the robots facing direction, two additional walls are connected to the wall the robot initially wanders into	Robot gradually avoids the initial wall until the robots corresponding side-most sensor on the robot’s front face is the only sensor detecting the wall, the robot then transitions to follow the entire length of the current wall, before re-transitioning to ‘avoiding’, the robot repeats this process for the other two walls ahead as they are connected	Yes
4	Robot wanders into a wall that is positioned in front of its body, but is angled when made relative to the robots facing direction, an additional wall is connected to the wall the robot initially wanders into, there is a relatively large object positioned along the wall	Robot gradually avoids the initial wall until the robots corresponding side-most sensor on the robots front face is the only sensor detecting the wall, the robot then transitions to follow the entire length of the current wall, before re-transitioning to ‘avoiding’, the robot repeats this process for the following wall ahead as it is connected, up until the object is detected, for which the robot re-transitions	Yes

		to ‘avoiding’ once more and lastly ‘wanders’ into space following the evasion of the object	
5	Robot edge follows a triangular layout of walls with small arched corners leading into another followable wall	Robot follows the current wall in the triangular layout, robot transitions to the ‘wandering’ state upon finishing the following of the entire length of the wall, robot does not attach itself to the small arched edge and therefore does not follow the wall connected to it	Yes
6	Robot detects a small spherical object using its side-most sensor, for the corresponding side it was detected on	Robot follows the edge of the object momentarily upon initially detecting it, robot transitions to ‘avoiding’ upon exiting ‘edge following’, robot evades the object and then transitions to ‘wandering’ when in open space	Yes
7	Robot wanders around its subjective environment arbitrarily when no objects are nearby or in the detectable range of the robot’s sensors	Robot traverses’ forwards for a randomly selected distance until reached, then follows the robot turning in a randomly selected direction for a randomly selected angle, this process is reiterated as the robot does not detect an object, the robot explores different areas of the environment in a relatively short period of time	Yes
8	Robot ‘wanders’ or follows the edges of objects into a corner or relatively small space that has no front-facing exit	Robot avoids the objects that border and reside in the space and where applicable, follows the edges of the objects that exist there, robot iteratively transitions between ‘avoiding’ and ‘edge following’ where applicable, robot does not collide or enter the ‘wandering’ state	Yes
9	Robot ‘wanders’ or follows the edges of objects into a corner or confined space that has no front-facing exit	Robot avoids the objects that border and reside in the space and where applicable, follows the edges of objects that exist there, robot iteratively transitions between ‘avoiding’ and ‘edge following’ unless robot transitions to ‘stuck’ state when in a confined space, whereby the robot pivots around its own axis until the front-most facing sensors find an exit, robot transitions to ‘avoiding’ or ‘edge following’ afterwards to emerge into space before transitioning to ‘wandering’, robot does not collide or enter the ‘wandering’ state whilst in confined space(s)	Yes
10	Robot ‘wanders’, ‘edge follows’ or ‘avoids’ objects that leads itself into a space where multiple objects reside on either side of the robot, the robot’s sensors detect the objects at equal distances from it	Robot exits the current state and transitions to ‘avoiding’ (if not already), robot traverses forwards whilst no objects are detected ahead of it, after evading the objects the robot transitions to ‘wandering’ when in open space	Yes

Table 8: Robot controller, robot 'avoiding', 'wandering' and 'edge following' in an unknown environment, behavioural expectations, and autonomy test cases

To summarise the robot controller’s capabilities, it is evident that the Pioneer-3DX robot is able to combat a considerable number of scenarios in a range of environments; in consideration of the results obtained from the testing regime, it is inevitable that the robot outputs “levels of competence” [10] across its avoiding, wandering and edge following behavioural strategies. Significantly, each behaviour and related actuation can be invoked interchangeably and autonomously, this is proven by the robot not requiring “human intervention to complete tasks” [1].

Controller Code Base

```
function sysCall_init() -- System initialisation functionality

    openFilesAutomatically = true -- Determine whether the executable files are opened on simulation end

    relativePath = sim.getStringParameter(sim.stringparam_scene_path) -- Store the path to the current scene (used to make relative paths to other files)

    mapCoordinates = "" -- Create a string to store the file path to the map coordinates CSV file
    mapCoordinates = mapCoordinates .. relativePath .. "/MapCoordinates.csv" -- Concatenate the strings to form a full file path

    ransacCoordinates = "" -- Create a string to store the file path to the RANSAC coordinates CSV file
    ransacCoordinates = ransacCoordinates .. relativePath .. "/RANSAC/RansacCoordinates.csv" -- Concatenate the strings to form a full file path

    buildRANSAC = "" -- Create a string to store the file path to the RANSAC build solution batch file
    buildRANSAC = buildRANSAC .. relativePath .. "/RANSAC/RANSACBuild.bat" -- Concatenate the strings to form a full file path

    offlineMap = "" -- Create a string to store the file path to the Excel offline map file
    offlineMap = offlineMap .. relativePath .. "/Map.xlsx"

    solutionRANSAC = "" -- Create a string to store the file path to the RANSAC visual studio solution file
    solutionRANSAC = solutionRANSAC .. relativePath .. "/RANSAC/RANSAC.sln" -- Concatenate the strings to form a full file path

    mainMap = true -- Determine whether the robot is currently in the primary environment

    do -----[ AVOIDANCE VARIABLES ]-----
        sonarSensors = { -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1} -- Create and initialise an array for the robots sonar sensor objects

        for i = 1, 16, 1 do -- For all of the robots sonar sensors, do the following
            sonarSensors[i] = sim.getObjectHandle("Pioneer_p3dx_ultrasonicSensor".. i) -- Store all of the robots sonar sensor components into an array
        end -- End of the iterative statement

        leftWheelMotor = sim.getObjectHandle("Pioneer_p3dx_leftMotor") -- Store the robots left wheel motor object
        rightWheelMotor = sim.getObjectHandle("Pioneer_p3dx_rightMotor") -- Store the robots right wheel motor object

        noDetectionDistance = 0.34 -- Robots non-detection distance
        maxDetectionDistance = 0.2 -- Robots maximum detection distance

        objectDetected = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0} -- Create and initialise an array for the robots sonar sensor detections
```

```
        detectedDistance = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0} -- Store the distance that an object has been detected at for each of the robots sonar sensors

        braitenbergLeft = {-0.2, -0.4, -0.6, -0.8, -1, -1.2, -1.4, -1.6, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0} -- Create and initialise an array for the robots front-left sonar sensor angular offsets
        braitenbergRight = {-1.6, -1.4, -1.2, -1, -0.8, -0.6, -0.4, -0.2, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0} -- Create and initialise an array for the robots front-right sonar sensor angular offsets

        defaultVelocity = 2 -- Robot wheels default velocity

        turnTimer = 0 -- Robots turning timer
    end -----[ AVOIDANCE VARIABLES ]-----



    do -----[ MAPPING VARIABLES ]-----
        sceneDrawingPoints = sim.addDrawingObject(sim.drawing_points, 2, 0.005, -1, 100000) -- Setup scene drawing points

        pioneerObject = sim.getObjectHandle("Pioneer_p3dx") -- Store the robots object

        sonarAngles = { 90, 50, 30, 10, -10, -30, -50, -90} -- Sonar angles of the robots front eight sensors (heading assumed to be '0' degrees)

        sonarReadings = { -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1 } -- Create and initialise an array for sonar readings

        sonarSensorPositions = { -1, -1, -1, -1, -1, -1, -1, -1 } -- Create and initialise an array for sonar sensor positions

        do -----[ OFFLINE MAP VARIABLES ]------
        graphPointRoundDecimalPlaces = 1 -- The number of decimal places the graph plot positions are rounded to

        offlineMapCoordinates = { } -- Create an array for storing the counts for objects detected at each map space coordinate
        offlineMapCounters = { } -- Create an array for storing the counts of detected objects

        mapWidth = 0 -- The width of the scene space to be mapped (resizeable floor size)
        mapHeight = 0 -- The height of the scene space to be mapped (resizable floor size)

        mapWidth = 10 / (graphPointRoundDecimalPlaces / 10) -- Width of the map (resizeable floor width), considers round amount
        mapHeight = 10 / (graphPointRoundDecimalPlaces / 10) -- Height of the map (resizable floor height), considers round amount

        for i = 1, mapWidth, 1 do -- For the width of the scene to be mapped
            offlineMapCoordinates[i] = { } -- Create an array for storing the 'Y' axis values of map positions

            offlineMapCounters[i] = { } -- Create an array for storing the count of objects detected in the 'Y' axis of map positions
```

```
      for j = 1, mapHeight, 1 do -- For the height of the scene to be mapped
        offlineMapCoordinates[i][j] = 0 -- Initialise the positions used for mapping graph plots to the
scene space

        offlineMapCounters[i][j] = 0 -- Initialise the counts for detected objects in the map space
      end -- End of the iterative statement
    end -- End of the iterative statement

    previousGraphPositionX = { 0, 0, 0, 0, 0, 0, 0, 0 } -- Store the previous graph plot position value
for the 'X' axis, translated into the maps coordinate space
    previousGraphPositionY = { 0, 0, 0, 0, 0, 0, 0, 0 } -- Store the previous graph plot position value
for the 'Y' axis, translated into the maps coordinate space
    end -----[ OFFLINE MAP VARIABLES ]------

    do -----[ RANSAC VARIABLES ]------
    sections = 2 -- The number of sections detected coordinates are divided into
    coordinateArraySize = 0 -- The number of coordinates detected throughout the simulation
    coordinatesPerSection = 0 -- The number of coordinates allocated to each section of coordinates

    allCoordinatesCounter = 1 -- The number of object positions detected throughout the simulation
(used to index array)
    allDetectedCoordinates = 0 -- The number of object positions detected throughout the
simulation

    allCoordinatesDetected = { } -- Create an array for storing all of the object positions detected
throughout the simulation

    ransacSections = { } -- Create an array for storing the sectioned coordinates

    pointIndexOne = { } -- Index of randomly selected, sectioned 'X' and 'Y' coordinates
    pointIndexTwo = { } -- Index of randomly selected, sectioned 'X' and 'Y' coordinates
    ransacPoints = { } -- Create an array for storing the randomly selected, sectioned coordinates

    ymc = { } -- Create an array for storing the gradient, y intersect and y values of a calcualted line
(y = mx + c)

    pointEuclideanDistanceFromLine = 0 -- The distance the currently iterated point is from the
currently calculated line
    desiredPointEuclideanDistanceFromLine = 1 -- The distance a point is condisered agreeing with
the currently calculated line

    currentPointsAgreeWithLine = { } -- Create an array for storing the amount of point that agree
with the currently calculated line
    highestPointsAgreeWithLine = { } -- Create an array for storing the highest amount of points that
agree with a line

    pointsForBestLines = { } -- Create an array for storing the points for lines calculated, that have
the most amount of points agreeing with it (lines of best fit)

    end -----[ RANSAC VARIABLES ]------
```

```
    do -----[ DRAWING VARIABLES ]-----
    sensorReadingToDrawGraph = {} -- Create and initialise an array for storing the graph points to plot

    for i = 1, 8, 1 do -- For all of the robots front facing sensors, do the following
      sensorReadingToDrawGraph[i] = {} -- Create and initialise an array for storing the 'X' and 'Y' coordinates of each graphs point, for the currently iterated sensor

      sensorReadingToDrawGraph[i][1] = 0 -- Store the 'X' coordinate of the graph point detected and calculated for the currently iterated sensor
      sensorReadingToDrawGraph[i][2] = 0 -- Store the 'Y' coordinate of the graph point detected and calculated for the currently iterated sensor
    end -- End of the iterative statement

    sensorReadingToDrawPoint = {} -- Create and initialise an array for storing the scene points to draw

    for i = 1, 8, 1 do -- For all of the robots front facing sensors, do the following
      sensorReadingToDrawPoint[i] = {} -- Create and initialise an array for storing the 'X' and 'Y' coordinates of each scenes point, for the currently iterated sensor

      sensorReadingToDrawPoint[i][1] = 0 -- Store the 'X' coordinate of the scene point detected and calculated for the currently iterated sensor
      sensorReadingToDrawPoint[i][2] = 0 -- Store the 'Y' coordinate of the scene point detected and calculated for the currently iterated sensor
    end -- End of the iterative statement

    end -----[ DRAWING VARIABLES]-----

  end -----[ MAPPING VARIABLES ]-----

  do -----[ EDGE FOLLOWING VARIABLES ]-----
    setPoint = 0.25 -- PID controller set point (desired distance from walls)
    maxDistance = 0.275 -- PID controller maximum distance (maximum distance from walls)

    proportionalGain = 7 -- PID controller proportional gain
    integralGain = 4 -- PID controller integral gain
    derivativeGain = 0.2 -- PID controller derivative gain

    leftErrorSum = { } -- PID controller left edge following error array
    rightErrorSum = { } -- PID controller right edge following error array

    leftErrorCounter = 1 -- PID controller left edge following error counter
    rightErrorCounter = 1 -- PID controller right edge following error counter

    integralThreshold = 10 -- PID controller integral threshold

    leftCurrentError = 0 -- PID controller current left edge following error
    leftLastError = 0 -- PID controller last left edge following error

    rightCurrentError = 0 -- PID controller current right edge following error
```

```
        rightLastError = 0 -- PID controller last right edge following error

        edgeFollowingLeftDetected = false -- Determine whether the robot will follow the edge of a detected object using its left-most front facing sensor
        edgeFollowingRightDetected = false -- Determine whether the robot will follow the edge of a detected object using its right-most front facing sensor

        edgeEndReached = false -- Determine whether the end of a followed edge has been reached
        edgeFollowingTimer = 0 -- Robots minimum time edge following to be considered in the 'edge following' phase

        RMSE = 0 -- Robot edge following RMSE value
    end -----[ EDGE FOLLOWING VARIABLES ]-----

    do -----[ WANDERING VARIABLES ]-----
        wanderingTurnAngle = 0 -- Robots wandering turn angle

        wanderingForwardDistance = 0 -- Robots wandering forward distance

        robotPosition = { 0, 0, 0 } -- Robots position incrementer

        currentRobotPosition = { 0, 0, 0 } -- Robots current position table/ array
        previousRobotPosition = { 0, 0, 0 } -- Robots previous position table/ array
        accumulatedForwardDistance = 0 -- Accumulated distance the robot has moved forwards

        wanderingForwardDistanceSet = false -- Determine whether the robots wandering forward distance has been set

        robotRotation = 0 -- Robots rotation incrementer

        currentRobotRotation = { 0, 0, 0 } -- Robots current rotation table/ array
        accumulatedRotationAngle = 0 -- Accumulated angle the robot has rotated towards

        currentRobotHeading = 0 -- Robots current heading (facing direction)
        previousRobotHeading = 0 -- Robots previous heading (facing direction)

        robotWanderingReset = false -- Determine whether the robots 'wandering' phase configuration requires to be reset (was interrupted)

        targetPositions = { } -- Create an array for storing the positions of each area in the environment

        for i = 1, 9, 1 do -- For all of the target areas specified, do the following
            targetPositions[i] = { } -- Create another array for store the positions values for each area (two-dimensional)
        end -- End of the iterative statement

        targetPositions[1][1] = -3.85 -- Top-left area target position for the 'X' axis
        targetPositions[1][2] = 3.85 -- Top-left area target position for the 'Y' axis

        targetPositions[2][1] = 0.00 -- Top-middle area target position for the 'X' axis
        targetPositions[2][2] = 3.85 -- Top-middle area target position for the 'X' axis
```

```
    targetPositions[3][1] = 3.00 -- Top-right area target position for the 'X' axis
    targetPositions[3][2] = 3.00 -- Top-right area target position for the 'X' axis

    targetPositions[4][1] = -3.85 -- Middle-left area target position for the 'X' axis
    targetPositions[4][2] = 0.00 -- Middle-left area target position for the 'X' axis

    targetPositions[5][1] = 0.00 -- Central area target position for the 'X' axis
    targetPositions[5][2] = 0.00 -- Central area target position for the 'X' axis

    targetPositions[6][1] = 4.00 -- Middle-right area target position for the 'X' axis
    targetPositions[6][2] = 0.00 -- Middle-right area target position for the 'X' axis

    targetPositions[7][1] = -3.85 -- Bottom-left area target position for the 'X' axis
    targetPositions[7][2] = -3.85 -- Bottom-left area target position for the 'X' axis

    targetPositions[8][1] = 0.00 -- Bottom-middle area target position for the 'X' axis
    targetPositions[8][2] = -3.85 -- Bottom-middle area target position for the 'X' axis

    targetPositions[9][1] = 4.00 -- Bottom-right area target position for the 'X' axis
    targetPositions[9][2] = -3.85 -- Bottom-right area target position for the 'X' axis

    topLeftTarget = { -3.85, 3.85 } -- Store the top-left area target for robot exploration
    bottomLeftTarget = { -3.85, -3.85 } -- Store the top-right area target for robot exploration
    topRightTarget = { 4.00, 3.85 } -- Store the bottom-left area target for robot exploration
    bottomRightTarget = { 4.00, -3.85 } -- Store the bottom-right area target for robot exploration
    centreMiddleTarget = { 0.00, 0.00 } -- Store the centre area target for robot exploration
    topMiddleTarget = { 0.00, 3.85 } -- Store the top-middle area target for robot exploration
    bottomMiddleTarget = { 0.00, -3.85 } -- Store the bottom-middle area target for robot exploration
    leftMiddleTarget = { -3.85, 0.00 } -- Store the left-middle area target for robot exploration
    rightMiddleTarget = { 4.00, 0.00 } -- Store the right-middle area target for robot exploration

    targetClosestReached = { } -- Create an array for storing the closest position achieved to each area in the environment

    for i = 1, 9, 1 do -- For all of the target areas specified, do the following
      targetClosestReached[i] = { } -- Create another array for storing the position values for each area (two-dimensional)

      targetClosestReached[i][1] = 0 -- Intialise the 'X' position value for the currently iterated area
      targetClosestReached[i][2] = 0 -- Intialise the 'Y' position value for the currently iterated area
    end -- End of the iterative statement

    robotCurrentArea = { } -- Create an array for determining the area in the environment that the robot is currently situated in

    for i = 1, 9, 1 do -- For all of the target areas specified, do the following
      robotCurrentArea[i] = false -- Intialise the boolean variables
    end -- End of the iterative statement
```

```
        robotExploredArea = { } -- Create an array for determining the areas in the environment that the robot has explored

        for i = 1, 9, 1 do -- For all of the target areas specified, do the following
           robotExploredArea[i] = false -- Intialise the boolean variables
        end -- End of the iterative statement

        robotAreaExploring = { } -- Create an array for determining the area in the environment that the robot is currently exploring

        for i = 1, 9, 1 do -- For all of the target areas specified, do the following
           robotAreaExploring[i] = false -- Intialise the boolean variables
        end -- End of the iterative statement

        robotExploringArea = 0 -- Determine the area that the robot has been assigned to explore, numerically
        robotUnexploredAreas = 9 -- Store the number of areas unexplored by the robot

        robotAreaToExploreSelected = false -- Determine whether the robot has been assigned an area to explore
        robotExploredAreaSelected = true -- Determine whether the robot has explored the curretly assigned area
        allAreasExplored = false -- Detemine whether all of the areas have been explored by the robot

        exploringAreaOutput = "UNASSIGNED" -- Store the area being explored by the robot currently

        targetDifference = { } -- Create an array for storing the difference between the robots and target areas positions for each area in the environment

        for i = 1, 9, 1 do -- For all of the target areas specified, do the following
           targetDifference[i] = 0 -- Intialise all of the distances for each area
        end -- End of the iterative statement

        previousTargetDifference = { } -- Create an array for storing the previous difference between the robots ad target areas positions for each area in the environment

        for i = 1, 9, 1  do -- For all of the target areas specified, do the following
           previousTargetDifference[i] = 0 -- Intialise all of the distance variables for each area
        end -- End of the iterative statement

        robotDistanceToTargets = { } -- Create an array for storing the distance between the robot and all of the target areas in the environment

        for i = 1, 9, 1 do -- For all of the target areas specified, do the following
           robotDistanceToTargets[i] = 0 -- Intialise all of the distance variables for each area
        end -- End of the iterative statement

        robotClosestToTarget = { } -- Create an array for storing the closest distance achieved to each area in the environment

        for i = 1, 9, 1 do -- For all of the target areas specified, do the following
```

```
      robotClosestToTarget[i] = -1 -- Initialise all of the distance variables for each area
    end -- End of the iterative statement

    areaExploredOutput = { } -- Create an array for determining whether the area the robot is currently situated in, has been explroed already

    for i = 1, 9, 1 do -- For all of the target areas specified, do the following
      areaExploredOutput[i] = "No" -- Intialise the string variables
    end -- End of the iterative statement

    robotRotationTranslated = { 0, 0 } -- Store the robots orientation for the 'X' and 'Y' axes (translated)
    robotTargetExploreAngle = 0 -- Store the angle the robot considers when rotating to the current area target assigned

    robotCurrentHeading0to360 = 0 -- Store the robots current heading (translated)
    robotDifferenceBetweenAngles = 0 -- Store the angular difference between the robots position and the assigned target position

    robotRotatedToTarget = 0 -- Determine how far the robot has rotated towards the current area target since the last frame was made
    rotateAccumulatedRotatedToTarget = 0 -- Determine how far the robot has rotated towards the current area target cummulatively
  end -----[ WANDERING VARIABLES ]-----


  do -----[ ROBOT STATE VARIABLES ]------
    robotIsAvoiding = false -- Determine whether the robot is within the 'avoiding' phase
    robotIsEdgeFollowing = false -- Determine whether the robot is within the 'edge-following' phase

    robotIsReversing = false -- Determine whether the robot is traversing backwards in relation to its facing direction
    robotIsTurning = false -- Determine whether the robot is rotating around its axis for a given time
    robotIsStuck = false -- Determine whether the robot is stuck (objects surrounding it)

    robotIsMovingForward = false -- Determine whether the robot is traversing forwards in relation to its facing direction
    robotIsRotating = false -- Determine whether the robot is rotating around its axis to a given angle
  end -----[ROBOT STATE VARIABLES ]------


  do -----[ CONSOLE OUTPUT ]------
    debugMode = true -- Determine whether comments are printed to the console from outside of the actuation function

    robotPosition = { 0, 0 } -- Robots position (X, Y)
    robotHeading = 0 -- Robots heading (degrees)

    robotSpeed = 0 -- Robots movement speed (meteres per second)
```

```
    robotDistanceTravelled = 0 -- Robots distance travelled (metres)

    previousPosition = { 0, 0, 0 } -- Robots previous position

    leftMostDetectedObject = { 0, 0 } -- Left most detected objects corresponding sensor and its distance from the object
    rightMostDetectedObject = { 0, 0 } -- Right most detected objects corresponding sensor and its distance from the object

    leftString = "" -- Store the robots object detecting sensor and distance to the closet object, relative to the robots left side
    rightString = "" -- Store the robots object detecting sensor and distance to the closet object, relative to the robots right side

    ransacTarget = math.random(8000, 12000) -- Generate a number of coordinates for RANSAC lines to be calculated from

    targetSet = false -- Determine whether the RANSAC target has been set
    validTarget = false -- Determine whether a suitable RANSAC target has been selected

    while (targetSet == false) do -- While the the target number of coordinates is not even (odd), do the following
      if (ransacTarget % 2 == 0) then -- If the current target is divisible by '2' (without a remainder - even), do the following
        if (validTarget == false) then -- If a valid target for the number of coordinates used by RANSAC (coordinates per section) has not been met, do the following
          for i = 1, ransacTarget, 1 do -- For the size of the current RANSAC target, do the following
            if ((ransacTarget / i) > 2 and (ransacTarget / i) <= 12 and (ransacTarget / i) % 2 == 0) then -- If the current RANSAC target is divisble by '2' (without a remainder - even)
                                                                      -- and creates a multiple of more than '2' but less than '12' (more than two coordinates per section is achieveable), do the following
              validTarget = true -- A valid target has been met
            end -- End of the conditional statement
          end -- End of iterative statement

          if (validTarget == false) then -- If the current RANSAC target is not valid for the purpose of calculation, do the following
            ransacTarget = math.random(8000, 12000) -- Regenerate a number of coordinates for RANSAC lines to be calculated from

          end -- End of conditional statement
        else -- If a valid target for the number of coordinates used by RANSAC (coordinates per section) has been met, do the following
          targetSet = true -- The target has been set
        end -- End of the conditional statement
      else -- If the current target is not divisible by '2' (with a remainder - odd), do the following
        ransacTarget = math.random(8000, 12000) -- Regenerate a number of coordinates for RANSAC lines to be calculated from

      end -- End of conditional statement
    end -- End of the conditional statement
```

```
    printf("RANSAC Target [" .. ransacTarget .. "]") -- Output the target number of coordinates for RANSAC lines to be calculated from

    ransacTargetCompletion = 0 -- The percentage of completion, relative to the number of positions detected vs the target number of coordinates to be detected
  end -----[ CONSOLE OUTPUT ]------

end -- End of the function declaration




function sysCall_cleanup() -- System cleanup functionality

  if(mainMap == true) then -- If the robot is currently in the primary environment, do the following

    do -----[ MAP OUTPUT ]------
    mapFile = io.open(mapCoordinates, "w") -- Open the mapping file, set to write data to the file
    io.output(mapFile) -- Write the output to the opened file

    for i = 1, mapWidth, 1 do -- For the width of the scene to be mapped, do the following
      for j = 1, mapHeight, 1 do -- For the height of the scene to be mapped, do the following
        if (j == mapHeight) then -- If the current iteration is equal to the height dimension of the resizable floor, do the following
          io.write(offlineMapCounters[i][j], "\n") -- Write the number of times an object at the currently iterated position has been detected, followed by a new line for writing the next row of positions
        else -- If the current iteration is not equal to the height dimension of the resizable floor, do the following
          io.write(offlineMapCounters[i][j], ",") -- Write the number of times an object at the currently iterated position has been detected, to the corresponding file, followed by a comma for writing the next counter
        end -- End of the conditional statement
      end -- End of the iterative statement
    end -- End of the iterative statement

    io.close(mapFile) -- Close the mapping file

    end -----[ MAP OUTPUT ]------

    do -----[ RANSAC OUTPUT ]------
    ransacFile = io.open(ransacCoordinates, "w") -- Open the mapping file, set to write data to the file
    io.output(ransacFile) -- Write the output to the opened file

    do -----[ CALCULATE THE NUMBER OF SECTIONS TO USE ]------
    coordinateArraySize = table.getn(allCoordinatesDetected) -- Store the number of object poisitions detected throughout the simulation
```

```
    if (coordinateArraySize % 2 ~= 0) then -- If the number of coordinates detected is not divisble by '2' without a remainder (odd), do the following
        allCoordinatesDetected[allCoordinatesCounter + 1] = { } -- Create an array for storing detected object positions

        allCoordinatesDetected[allCoordinatesCounter + 1][1] = 0 -- Intialise the first element in the array
        allCoordinatesDetected[allCoordinatesCounter + 1][2] = 0 -- Intialise the second element in the array
    end -- End of the conditional statement

    for i = 2, coordinateArraySize, 1 do -- For the given range of possible number of sections, do the following
        if (i ~= coordinateArraySize) then -- If the current iteration is not equal to the number of coordinates stored, do the following
            if (coordinateArraySize % i == 0) then -- If the size of all of the detected object positions is divisble by the current iteration and has no remainder, do the following
                if ((coordinateArraySize / i) > 2 and (coordinateArraySize / i) <= 12) then -- If the current iteration provides more than '2' coordinates per section, do the following
                    sections = i -- Set the number of sections of coordinates to the current iteration
                    coordinatesPerSection = coordinateArraySize / sections -- Calculate the number of coordinates allocated to each section
                end -- End of the conditional statement
        end -- End of the conditional statement
        end -- End of the conditional statement
    end -- End of the iterative statement
    end -----[ CALCULATE THE NUMBER OF SECTIONS TO USE ]------

    do -----[ INITIALISE ARRAYS FOR STORING EACH SECTIONS POINTS/ POINTS FOR LINE EQUATIONS ]------
    for i = 1, sections, 1 do -- For all of the sections of coordinates, do the following
        ransacSections[i] = { } -- Create an array for storing the section number of coordinates

        ransacPoints[i] = { } -- Create an array for storing two randomly selected points, used by RANSAC to calculate a line of best fit for the currently iterated section of coordinates
        ransacPoints[i][1] = 0 -- Coordinate 'x1' initialisation
        ransacPoints[i][2] = 0 -- Coordinate 'y1' initialisation
        ransacPoints[i][3] = 0 -- Coordinate 'x2' initialisation
        ransacPoints[i][4] = 0 -- Coordinate 'y2' initialisation

        ymc[i] = { } -- Create an array for storing the gradient, y intersect and y value, values, used to calculate lines for every section of coordinates
        ymc[i][1] = 0 -- Y value initialisation
        ymc[i][2] = 0 -- Gradient initialisation
        ymc[i][3] = 0 -- Y intersect initialisation

        currentPointsAgreeWithLine[i] = { } -- Create an array for storing the number of points agreeing with the calcualted line, for the currently iterated section of coordinates
        currentPointsAgreeWithLine[i][1] = 0 -- Initialise the number of agreeing points for the currently iterated section of coordinates
```

```
        highestPointsAgreeWithLine[i] = { } -- Create an array for storing the highest amount of points agreeing with any line calculated, for the currently iterated section of coordinates
        highestPointsAgreeWithLine[i][1] = 0 -- Initialise the highest number of agreeing points for the currently iterated section of coordinates

        pointsForBestLines[i] = { } -- Create an array for storing the points that represent the best line of fit for the currently iterated section of coordinates
        pointsForBestLines[i][1] = 0 -- Coordinate 'x1' initialisation
        pointsForBestLines[i][2] = 0 -- Coordinate 'y1' initialisation
        pointsForBestLines[i][3] = 0 -- Coordinate 'x2' initialisation
        pointsForBestLines[i][4] = 0 -- Coordinate 'y2' initialisation

        for j = 1, coordinateArraySize, 1 do -- For the number of object positions detected throughout the simulation, do the following
            ransacSections[i][j] = { } -- Create an array for storing the coordinates of the currently iterated sections coordinates
            ransacSections[i][j][1] = 0 -- Coordinate 'X' initialisation
            ransacSections[i][j][2] = 0 -- Coordinate 'Y' initialisation
        end -- End of the iterative statement

    end -- End of the iterative statement
    end -----[ INITIALISE ARRAYS FOR STORING EACH SECTIONS POINTS/ POINTS FOR LINE EQUATIONS ]------

    do -----[ SECTION THE COORDINATES ]------
    for i = 1, sections, 1 do -- For all of the sections of coordinates, do the following
        if (i == 1) then -- If the current section iterated is the first section of coordinates, do the following
            for j = 1, coordinatesPerSection, 1 do -- For the number of coordinates allocated to every section, do the folloiwng
                ransacSections[i][j][1] = allCoordinatesDetected[j][1] -- Store the 'X' value of coordinates for the section
                ransacSections[i][j][2] = allCoordinatesDetected[j][2] -- Store the 'Y' value of coordinates for the section

            end -- End of the iterative statement
        else -- If the current section iterated is not the first section of coordinates, do the following
            for k = (coordinatesPerSection * (i - 1) + 1), coordinatesPerSection * i, 1 do -- For the sections coordinate boundaries (relative to all of the object positions detected throughout the simulation), do the following
                ransacSections[i][k][1] = allCoordinatesDetected[k][1] -- Store the 'X' value of coordinates for the section
                ransacSections[i][k][2] = allCoordinatesDetected[k][2] -- Store the 'Y' value of coordinates for the section

            end -- End of the iterative statement
        end -- End of the conditional statement
    end -- End of the iterative statement
    end -----[ SECTION THE COORDINATES ]------

    for x = 1, 1000, 1 do -----[ STOP CONDITION ]------
```

```
        do -----[ GET TWO POINTS FOR EACH SECTION OF COORDINATES ]------
        for i = 1, sections, 1 do -- For all of the sections of coordinates, do the following
          if (i == 1) then -- If the current section iterated is the first section of coordinates, do the
following
            pointIndexOne[i] = math.random(1, coordinatesPerSection * i) --  Generate a random
index for retrieving the first coordinate in the currently iterated section of coordinates
            pointIndexTwo[i] = math.random(1, coordinatesPerSection * i) --  Generate a random
index for retrieving the second coordinate in the currently iterated section of coordinates

            while (pointIndexTwo[i] == pointIndexOne[i]) do -- If the randomly generated indexes are
the same, do the following
              pointIndexOne[i] = math.random(1, coordinatesPerSection * i) -- Regenerate the first
coordinates index
              pointIndexTwo[i] = math.random(1, coordinatesPerSection * i) -- Regenerate the
second coordinates index
            end -- End of the conditional statement

          else -- If the current section iterated is not the first section of coordinates, do the following
            pointIndexOne[i] = math.random(coordinatesPerSection * (i - 1) + 1,
coordinatesPerSection * i) --  Generate a random index for retrieving the first coordinate in the
currently iterated section of coordinates--  Generate a random index for retrieving the first
coordinate in the currently iterated section of coordinates
            pointIndexTwo[i] = math.random(coordinatesPerSection * (i - 1) + 1,
coordinatesPerSection * i) --  Generate a random index for retrieving the second coordinate in the
currently iterated section of coordinates--  Generate a random index for retrieving the second
coordinate in the currently iterated section of coordinates

            while (pointIndexTwo[i] == pointIndexOne[i]) do -- If the randomly generated indexes are
the same, do the following
              pointIndexOne[i] = math.random(coordinatesPerSection * (i - 1) + 1,
coordinatesPerSection * i) -- Regenerate the first coordinates index
              pointIndexTwo[i] = math.random(coordinatesPerSection * (i - 1) + 1,
coordinatesPerSection * i) -- Regenerate the second coordinates index
            end -- End of the conditional statement--]]

          end -- End of the conditional statement

          ransacPoints[i][1] = ransacSections[i][pointIndexOne[i]][1] -- Store the 'x1' coordinate for
the curreny iterated section of coordinates (first point)
          ransacPoints[i][2] = ransacSections[i][pointIndexOne[i]][2] -- Store the 'y1' coordinate for
the curreny iterated section of coordinates (first point)

          ransacPoints[i][3] = ransacSections[i][pointIndexTwo[i]][1] -- Store the 'x2' coordinate for
the curreny iterated section of coordinates (second point)
          ransacPoints[i][4] = ransacSections[i][pointIndexTwo[i]][2] -- Store the 'y2' coordinate for
the curreny iterated section of coordinates (second point)

          --printf("Sections: " .. sections .. "    Per section: " .. coordinatesPerSection ..
              --"   X1: " .. ransacPoints[i][1] .. "   Y1: " .. ransacPoints[i][2] .. "   Index One: " ..
pointIndexOne[i] ..
```

```
                        --"   X2: " .. ransacPoints[i][3] .. "   Y2: " .. ransacPoints[i][4] .. "   Index Two: " .. pointIndexTwo[i])
                --io.write("Section: " .. i, ",", ransacPoints[i][1], ",", ransacPoints[i][2], ",", ransacPoints[i][3], ",", ransacPoints[i][4], "\n")
            end
            end -----[ GET TWO POINTS FOR EACH SECTION OF COORDINATES ]------

            do -----[ FIND BEST LINE OF FIT ]------
            for i = 1, sections, 1 do -- For all of the sections of coordinates, do the following
                ymc[i][2] = (ransacPoints[i][4] - ransacPoints[i][2]) / (ransacPoints[i][3] - ransacPoints[i][1]) -- Gradient (m) = (y2-y1)/(x2-x1)
                ymc[i][3] = ransacPoints[i][2] - (ymc[i][2] * (ransacPoints[i][1])) -- Y intersect (c) = y1 - (gradient * x1)
                ymc[i][1] = 1 -- Give Y a value (itself - 1Y)

                -- y = mx + c (original)
                -- c = y - mx (translated)

                if (i == 1) then -- If the current section iterated is the first section of coordinates, do the following
                    for j = 1, coordinatesPerSection, 1 do -- For the number of coordinates allocated to every section, do the folloiwng

                        -- If the currently iterated pair of coordinates are not the randomly selected coordinates for the currently iterated section, do the following
                        --if (ransacPoints[i][1] == ransacSections[i][j][1] and ransacPoints[i][2] == ransacSections[i][j][2]) then
                        --elseif (ransacPoints[i][3] == ransacSections[i][j][1] and ransacPoints[i][4] == ransacSections[i][j][2]) then
                        --else

                            pointEuclideanDistanceFromLine = ymc[i][2] * (ransacSections[i][j][1] - ransacSections[i][j][2]) -- Calculate the euclidean distance between the currenty iterated point in the currently iterated section, to the calcualted line

                            if (pointEuclideanDistanceFromLine < 0) then -- If the currently iterated points distance to the calculated line is smaller than '0' (negative), do the following
                                pointEuclideanDistanceFromLine = -(pointEuclideanDistanceFromLine) -- Negate the euclidean distance
                            end -- End of the conditional statement

                            if (pointEuclideanDistanceFromLine < desiredPointEuclideanDistanceFromLine) then -- If the currently iterated points distance to the calculated line is within the desired distance to the line, do the following
                                currentPointsAgreeWithLine[i][1] = currentPointsAgreeWithLine[i][1] + 1 -- Increment the number of agreeing points to the calculated line
                            end -- End of the conditional statement

                            if (highestPointsAgreeWithLine[i][1] == 0) then -- If no highest points agreeing with the current line has been set before for the section of coordinates
```

```
                        highestPointsAgreeWithLine[i][1] = currentPointsAgreeWithLine[i][1] -- Set the highest number of points agreeing with any line for the currently iterated section of coordinates, to the current number of points agreeing with the calculated line

                        for k = 1, 4, 1 do -- For each points 'X' and 'Y' value, do the following
                           pointsForBestLines[i][k] = ransacPoints[i][k] -- Store the best line of fit for the currently iterated section of coordinates
                        end -- End of iterative statement
                     else -- If a highest points agreeing has been set before for the section of coordinates
                        if (currentPointsAgreeWithLine[i][1] > highestPointsAgreeWithLine[i][1]) then -- If the number of points agreeing with the currently calculated line is larger than the highest number of points agreeing with any line for the currently iterated section of coordinates, do the following
                           highestPointsAgreeWithLine[i][1] = currentPointsAgreeWithLine[i][1] -- Set the highest number of points agreeing with any line for the currently iterated section of coordinates, to the current number of points agreeing with the calculated line

                           for l = 1, 4, 1 do -- For each points 'X' and 'Y' value, do the following
                              pointsForBestLines[i][l] = ransacPoints[i][l] -- Store the best line of fit for the currently iterated section of coordinates
                           end -- End of the iterative statement
                        end -- End of the conditional statement
                     end -- End of the conditional statement

                     --printf("Distance: " .. pointEuclideanDistanceFromLine .. "  Gradient: " .. ymc[i][2] ..
                        --"  X1: " .. ransacPoints[i][1] .. "  Y1: " .. ransacPoints[i][2] ..
                        --"  X2: " .. ransacPoints[i][3] .. "  Y2: " .. ransacPoints[i][4])

                  --end -- End of conditional statement
               end -- End of the iterative statement
            else
               for m = (coordinatesPerSection * (i - 1) + 1), coordinatesPerSection * i, 1 do -- For the sections coordinate boundaries (relative to all of the object positions detected throughout the simulation), do the following

                  -- If the currently iterated pair of coordinates are not the randomly selected coordinates for the currently iterated section, do the following
                  --if (ransacPoints[i][1] == ransacSections[i][m][1] and ransacPoints[i][2] == ransacSections[i][m][2]) then
                  --elseif (ransacPoints[i][3] == ransacSections[i][m][1] and ransacPoints[i][4] == ransacSections[i][m][2]) then
                  --else

                     pointEuclideanDistanceFromLine = ymc[i][2] * (ransacSections[i][m][1] - ransacSections[i][m][2]) -- Calculate the euclidean distance between the currenty iterated point in the currently iterated section, to the calcualted line

                     if (pointEuclideanDistanceFromLine < 0) then -- If the currently iterated points distance to the calculated line is smaller than '0' (negative), do the following
                        pointEuclideanDistanceFromLine = -(pointEuclideanDistanceFromLine) -- Negate the euclidean distance
                     end -- End of the conditional statement
```

```
              if (pointEuclideanDistanceFromLine < desiredPointEuclideanDistanceFromLine) then
-- If the currently iterated points distance to the calculated line is within the desired distance to the
line, do the following
                currentPointsAgreeWithLine[i][1] = currentPointsAgreeWithLine[i][1] + 1 --
Increment the number of agreeing points to the calculated line
              end -- End of the conditional statement

              if (highestPointsAgreeWithLine[i][1] == 0) then -- If no highest points agreeing with
the current line has been set before for the section of coordinates
                highestPointsAgreeWithLine[i][1] = currentPointsAgreeWithLine[i][1] -- Set the
highest number of points agreeing with any line for the currently iterated section of coordinates, to
the current number of points agreeing with the calculated line

                for n = 1, 4, 1 do -- For each points 'X' and 'Y' value, do the following
                  pointsForBestLines[i][n] = ransacPoints[i][n] -- Store the best line of fit for the
currently iterated section of coordinates
                end -- End of the iterative statement
              else -- If a highest points agreeing has been set before for the section of coordinates
                if (currentPointsAgreeWithLine[i][1] > highestPointsAgreeWithLine[i][1]) then -- If
the number of points agreeing with the currently calculated line is larger than the highest number of
points agreeing with any line for the currently iterated section of coordinates, do the following
                  highestPointsAgreeWithLine[i][1] = currentPointsAgreeWithLine[i][1] -- Set the
highest number of points agreeing with any line for the currently iterated section of coordinates, to
the current number of points agreeing with the calculated line

                  for o = 1, 4, 1 do -- For each points 'X' and 'Y' value, do the following
                    pointsForBestLines[i][o] = ransacPoints[i][o] -- Store the best line of fit for the
currently iterated section of coordinates
                  end -- End of the iterative statement
                end -- End of the conditional statement
              end -- End  of the conditional statement

              --printf("Distance: " .. pointEuclideanDistanceFromLine .. "  Gradient: " .. ymc[i][2] ..
                --"  X1: " .. ransacPoints[i][1] .. "  Y1: " .. ransacPoints[i][2] ..
                --"  X2: " .. ransacPoints[i][3] .. "  Y2: " .. ransacPoints[i][4])

            --end -- End of the conditional statement
          end -- End of the iterative statement
        end -- End of the conditional statement
      end -- End of the iterative statement
      end -----[ FIND BEST LINE OF FIT ]------

      printf("RANSAC Calculation Completion [" .. string.format("%.2f", (x / 1000) * 100) .. "
PERCENT]" ..
        " Coordinates: " .. coordinateArraySize .. "  Sections: " .. sections .. "  Coordinates Per
Section: " .. coordinatesPerSection) -- Indicate the RANSAC calculation completion

    end -----[ STOP CONDITION ]------

    for i = 1, sections, 1 do -- For all of the sections of coordinates, do the following
```

```
      io.write(pointsForBestLines[i][1], ",", pointsForBestLines[i][2], ",", pointsForBestLines[i][3], ",", pointsForBestLines[i][4], "\n") -- Write the points representing the lines of best fit, for the currently iterated section of coordinates
    end -- End of the iterative statement

    io.close(ransacFile) -- Close the ransac file

    printf("RANSAC Calculations Complete!") -- Notify the RANSAC calculations are complete

    if (openFilesAutomatically == true) then -- If the executable files are set to execute on simulation end, do the following
      sim.launchExecutable(offlineMap) -- Launch the EXCEL offline map
      sim.launchExecutable(solutionRANSAC) -- Launch the RANSAC SFML solution
      --sim.launchExecutable(buildRANSAC) -- Build and run the RANSAC SFML solution

    end -- End of the conditional statement

    end -----[ RANSAC OUTPUT ]------

    end -- end of the conditional statement

end -- End of the function declaration




function sysCall_sensing() -- Robot sensing functionality

  currentRobotPosition = sim.getObjectPosition(pioneerObject, -1) -- Store the current position of the robot (for all axes)

  currentRobotRotation = sim.getObjectOrientation(pioneerObject, -1) -- Store the current orientation of the robot (for all axes)
  currentRobotHeading = math.deg(currentRobotRotation[3]) -- Store the robots current heading (Z axis)

  robotHeading = math.deg(currentRobotRotation[3]) -- Store the robots current heading for console output (Z axis)

  for i = 1, 2, 1 do -- For the 'X' and 'Y' axes of the robots position, do the following
    robotPosition[i] = currentRobotPosition[i] -- Store the current position of the robot for console output (for all axes)
  end -- End of the iterative statement

  if (currentRobotHeading < 0) then -- If the robots current heading is smaller than zero (negative), do the following
    currentRobotHeading = -(currentRobotHeading) -- Set the robots current heading to the robots current heading negated (positive)
  end -- End of the conditional statement

  do -----[ WANDERING ]-----
```

```
    robotRotationTranslated[1] = math.deg(currentRobotRotation[1]) -- Store the robots current orientation in the 'X' axis
    robotRotationTranslated[2] = math.deg(currentRobotRotation[2]) -- Store the robots current orientation in the 'Y' axis

    if (robotRotationTranslated[1] < 0) then -- If the robots current orientation in the 'X' axis is smaller than '0' degrees (negative), do the following
        robotRotationTranslated[1] = 359.9 -- Set the robots current orientation in the 'X' axis to '259.9' degrees
    elseif (robotRotationTranslated[1] > 359.9) then -- If the robots current orientation in the 'X' axis is larger than '359.9' degrees, do the following
        robotRotationTranslated[1] = 0 -- Set the robots current orientation in the 'X' axis to '0' degrees
    end -- End of the conditional statement

    if (robotRotationTranslated[2] < 0) then -- If the robots current orientation in the 'Y' axis is smaller than '0' degrees (negative), do the following
        robotRotationTranslated[2] = 359.9 -- Set the robots current orientation in the 'Y' axis to '259.9' degrees
    elseif (robotRotationTranslated[2] > 359.9) then -- If the robots current orientation in the 'Y' axis is larger than '359.9' degrees, do the following
        robotRotationTranslated[2] = 0 -- Set the robots current orientation in the 'Y' axis to '0' degrees
    end -- End of the conditional statement

    robotCurrentHeading0to360 = math.deg(currentRobotRotation[3]) -- Store the robots current heading for exploring unknown areas (Z axis)

    if (robotCurrentHeading0to360 < 0) then -- If the robots current heading is smaller than zero (negative), do the following
        robotCurrentHeading0to360 = 360 + robotCurrentHeading0to360 -- Set the robots current heading to the robots current heading translated
    end -- End of the conditional statement

    if (robotCurrentHeading0to360 < 0) then -- If the robots current heading translated is smaller than '0' degrees, do the following
        robotCurrentHeading0to360 = 359.9 -- Set the robots current heading translated to '359.9' degrees
    elseif (robotCurrentHeading0to360 > 359.9) then -- If the robots current heading translated is larger than '359.9' degrees, do the following
        robotCurrentHeading0to360 = 0 -- Set the robots current heading translated to '0' degrees
    end -- End of the conditional statement

    if (robotUnexploredAreas == 0) then -- If robot has explored all areas of the map, do the following
        allAreasExplored = true -- All areas of the map have been explored
    end -- End of the conditional statement

    if (allAreasExplored == false) then -- If all of the areas of the environment have not been explored, do the following
        for i = 1, 9, 1 do -- For all of the target areas specified, do the following
            robotDistanceToTargets[i] = math.sqrt((robotPosition[1] - targetPositions[i][1])^2 + (robotPosition[2] - targetPositions[i][2])^2)
        end -- End of the iterative statement
```

```
    areaClosestTo = 0 -- Store the area that the robot is closest to (numerically)
    closestDistance = 0 -- Store the distance that the robot is from each area of the environment

    for i = 1, 9, 1 do -- For all of the target areas specified, do the following
      if (closestDistance == 0) then -- If the closest distance has not been set, do the following
        areaClosestTo = i -- Store the current iteration (used for indexing)
        closestDistance = robotDistanceToTargets[i] -- Store the distance to the currently iterated
area as the closest area to the robot
      else -- If the closest distance has been set previously, do the following
        if (robotDistanceToTargets[i] < closestDistance) then -- If the robots current distance to the
currently iterated area is the closest distance to any area, do the following
          areaClosestTo = i -- Store the current iteration (used for indexing)
          closestDistance = robotDistanceToTargets[i] -- Store the distance to the currently iterated
area as the closest area to the robot
        end -- End of the conditional statement
      end -- End of the conditional statement
    end -- End of the iterative statement

    if (areaClosestTo == 1) then -- If the robot is situated closer to the 'top-left' region of the
environment, do the following
      for i = 1, 9, 1 do -- For all of the target areas specified, do the following
        if (i ~= areaClosestTo) then -- If the currently iterated area is not the area that the robot is
closest to, do the following
          robotCurrentArea[i] = false -- The robot is not in the currently iterated area
        else -- If the currently iterated area is the area that the robot is closest to, do the following
          robotCurrentArea[i] = true -- The robot is in the currently iterated area
        end -- End of the conditional statement
      end -- End of the iterative statement

      targetDifference[areaClosestTo] = math.sqrt(((targetPositions[areaClosestTo][1] -
targetClosestReached[areaClosestTo][1])^2) +
                            ((targetPositions[areaClosestTo][2] -
targetClosestReached[areaClosestTo][2])^2)) -- Store the difference between the robot and the
target for its exploration

      if (robotClosestToTarget[areaClosestTo] ~= -1) then -- If the robot has entered the area
before, do the following
        if (targetDifference[areaClosestTo] < robotClosestToTarget[areaClosestTo]) then -- If the
robots current distance to the area target is the closest it has been, do the following
          robotClosestToTarget[areaClosestTo] = targetDifference[areaClosestTo] -- Set the closest
distance to the current difference
        end -- End of the conditional statement
      else -- If the robot has not entered the area before, do the following
        robotClosestToTarget[areaClosestTo] = targetDifference[areaClosestTo] -- Store the current
distance of the robot to the target, as the closest distance
      end -- End of the conditional statement

      targetClosestReached[areaClosestTo][1] = robotPosition[1] -- Store the robots current 'X'
position as the closest achieved position to the area target position
```

```
      targetClosestReached[areaClosestTo][2] = robotPosition[2] -- Store the robots current 'Y' position as the closest achieved position to the area target position

      if (robotExploredArea[areaClosestTo] == false) then -- If the area the robot is current positioned in has not been explored, do the following
        if (targetDifference[areaClosestTo] < 1) then -- If the distance between the robots position and area targets position is smaller than '1' metre, do the following
          robotExploredArea[areaClosestTo] = true -- Mark the area as explored

          robotUnexploredAreas = robotUnexploredAreas - 1 -- Decrement the number of areas that are unexplored

          areaExploredOutput[areaClosestTo] = "Yes" -- Output the area as explored

          if (robotExploringArea == areaClosestTo) then -- If the area is currently the target area for exploration, do the following
            robotExploredAreaSelected = true -- The robot searches for another unexplored area
          end -- End of the conditional statement
        else -- If the distance between the robots position and area targets position is larger than desired distance, do the following
          areaExploredOutput[areaClosestTo] = "No" -- Output the robot has not explored the area
        end -- End of the conditional statement
      end -- End of the conditional statement

      previousTargetDifference[areaClosestTo] = targetDifference[areaClosestTo] -- Store the current distance to the target as the previous distance to the target (end of frame)

      if (debugMode == false) then -- If debug mode is not active, do the following
        printf("Robot Location [Top Left]    Robot Position in Area ["
          .. string.format("%.2f", targetClosestReached[areaClosestTo][1]) .. ", "
          .. string.format("%.2f", targetClosestReached[areaClosestTo][2]) .. "]    "
          .. "Closest [" .. string.format("%.2f", robotClosestToTarget[areaClosestTo]) .. "]    "
          .. "Explored [" .. areaExploredOutput[areaClosestTo] .. "]    "
          .. "Target Angle [" .. robotTargetExploreAngle .."]    "
          .. "Heading [" .. robotCurrentHeading0to360 .. "]    "
          .. "Difference [" .. robotDifferenceBetweenAngles .. "]    "
          .. "Angle Rotated [" .. string.format("%.2f", rotateAccumulatedRotatedToTarget) .. "]    "
          .. "Area Exploring [" .. exploringAreaOutput .. "]    "
          .. "Areas Unexplored [" .. robotUnexploredAreas .. "]") -- Output robot exploration information
      end -- End of the conditional statement
    elseif (areaClosestTo == 2) then -- If the robot is situated closer to the 'top-middle' region of the environment, do the following
      for i = 1, 9, 1 do -- For all of the target areas specified, do the following
        if (i ~= areaClosestTo) then -- If the currently iterated area is not the area that the robot is closest to, do the following
          robotCurrentArea[i] = false -- The robot is not in the currently iterated area
        else -- If the currently iterated area is the area that the robot is closest to, do the following
          robotCurrentArea[i] = true -- The robot is in the currently iterated area
        end -- End of the conditional statement
      end -- End of the iterative statement
```

```
    targetDifference[areaClosestTo] = math.sqrt(((targetPositions[areaClosestTo][1] -
targetClosestReached[areaClosestTo][1])^2) +
                              ((targetPositions[areaClosestTo][2] -
targetClosestReached[areaClosestTo][2])^2)) -- Store the difference between the robot and the
target for its exploration

    if (robotClosestToTarget[areaClosestTo] ~= -1) then -- If the robot has entered the area
before, do the following
      if (targetDifference[areaClosestTo] < robotClosestToTarget[areaClosestTo]) then -- If the
robots current distance to the area target is the closest it has been, do the following
        robotClosestToTarget[areaClosestTo] = targetDifference[areaClosestTo] -- Set the closest
distance to the current difference
      end -- End of the conditional statement
    else -- If the robot has not entered the area before, do the following
      robotClosestToTarget[areaClosestTo] = targetDifference[areaClosestTo] -- Store the current
distance of the robot to the target, as the closest distance
    end -- End of the conditional statement

    targetClosestReached[areaClosestTo][1] = robotPosition[1] -- Store the robots current 'X'
position as the closest achieved position to the area target position
    targetClosestReached[areaClosestTo][2] = robotPosition[2] -- Store the robots current 'Y'
position as the closest achieved position to the area target position

    if (robotExploredArea[areaClosestTo] == false) then -- If the area the robot is current
positioned in has not been explored, do the following
      if (targetDifference[areaClosestTo] < 0.6) then -- If the distance between the robots position
and area targets position is smaller than '0.6' metres, do the following
        robotExploredArea[areaClosestTo] = true -- Mark the area as explored

        robotUnexploredAreas = robotUnexploredAreas - 1 -- Decrement the number of areas
that are unexplored

        areaExploredOutput[areaClosestTo] = "Yes" -- Output the area as explored

        if (robotExploringArea == areaClosestTo) then -- If the area is currently the target area for
exploration, do the following
          robotExploredAreaSelected = true -- The robot searches for another unexplored area
        end -- End of the conditional statement
      else -- If the distance between the robots position and area targets position is larger than
desired distance, do the following
        areaExploredOutput[areaClosestTo] = "No" -- Output the robot has not explored the area
      end -- End of the conditional statement
    end -- End of the conditional statement

    previousTargetDifference[areaClosestTo] = targetDifference[areaClosestTo] -- Store the
current distance to the target as the previous distance to the target (end of frame)

    if (debugMode == false) then -- If debug mode is not active, do the following
      printf("Robot Location [Top Middle]    Robot Position in Area ["
        .. string.format("%.2f", targetClosestReached[areaClosestTo][1]) .. ", "
```

```
          .. string.format("%.2f", targetClosestReached[areaClosestTo][2]) .. "]    "
          .. "Closest [" .. string.format("%.2f", robotClosestToTarget[areaClosestTo]) .. "]    "
          .. "Explored [" .. areaExploredOutput[areaClosestTo] .. "]    "
          .. "Target Angle [" .. robotTargetExploreAngle .."]    "
          .. "Heading [" .. robotCurrentHeading0to360 .. "]    "
          .. "Difference [" .. robotDifferenceBetweenAngles .. "]    "
          .. "Angle Rotated [" .. string.format("%.2f", rotateAccumulatedRotatedToTarget) .. "]    "
          .. "Area Exploring [" .. exploringAreaOutput .. "]    "
          .. "Areas Unexplored [" .. robotUnexploredAreas .. "]") -- Output robot exploration
information
      end -- End of the conditional statement
    elseif (areaClosestTo == 3) then -- If the robot is situated closer to the 'top-right' region of the
environment, do the following
      for i = 1, 9, 1 do -- For all of the target areas specified, do the following
        if (i ~= areaClosestTo) then -- If the currently iterated area is not the area that the robot is
closest to, do the following
          robotCurrentArea[i] = false -- The robot is not in the currently iterated area
        else -- If the currently iterated area is the area that the robot is closest to, do the following
          robotCurrentArea[i] = true -- The robot is in the currently iterated area
        end -- End of the conditional statement
      end -- End of the iterative statement

      targetDifference[areaClosestTo] = math.sqrt(((targetPositions[areaClosestTo][1] -
targetClosestReached[areaClosestTo][1])^2) +
                            ((targetPositions[areaClosestTo][2] -
targetClosestReached[areaClosestTo][2])^2)) -- Store the difference between the robot and the
target for its exploration

      if (robotClosestToTarget[areaClosestTo] ~= -1) then -- If the robot has entered the area
before, do the following
        if (targetDifference[areaClosestTo] < robotClosestToTarget[areaClosestTo]) then -- If the
robots current distance to the area target is the closest it has been, do the following
          robotClosestToTarget[areaClosestTo] = targetDifference[areaClosestTo] -- Set the closest
distance to the current difference
        end -- End of the conditional statement
      else -- If the robot has not entered the area before, do the following
        robotClosestToTarget[areaClosestTo] = targetDifference[areaClosestTo] -- Store the current
distance of the robot to the target, as the closest distance
      end -- End of the conditional statement

      targetClosestReached[areaClosestTo][1] = robotPosition[1] -- Store the robots current 'X'
position as the closest achieved position to the area target position
      targetClosestReached[areaClosestTo][2] = robotPosition[2] -- Store the robots current 'Y'
position as the closest achieved position to the area target position

      if (robotExploredArea[areaClosestTo] == false) then -- If the area the robot is current
positioned in has not been explored, do the following
        if (targetDifference[areaClosestTo] < 1) then -- If the distance between the robots position
and area targets position is smaller than '1' metre, do the following
          robotExploredArea[areaClosestTo] = true -- Mark the area as explored
```

```
                    robotUnexploredAreas = robotUnexploredAreas - 1 -- Decrement the number of areas
that are unexplored

                    areaExploredOutput[areaClosestTo] = "Yes" -- Output the area as explored

                    if (robotExploringArea == areaClosestTo) then -- If the area is currently the target area for
exploration, do the following
                        robotExploredAreaSelected = true -- The robot searches for another unexplored area
                    end -- End of the conditional statement
                else -- If the distance between the robots position and area targets position is larger than
desired distance, do the following
                    areaExploredOutput[areaClosestTo] = "No" -- Output the robot has not explored the area
                end -- End of the conditional statement
            end -- End of the conditional statement

            previousTargetDifference[areaClosestTo] = targetDifference[areaClosestTo] -- Store the
current distance to the target as the previous distance to the target (end of frame)

            if (debugMode == false) then -- If debug mode is not active, do the following
                printf("Robot Location [Top Right]    Robot Position in Area ["
                    .. string.format("%.2f", targetClosestReached[areaClosestTo][1]) .. ", "
                    .. string.format("%.2f", targetClosestReached[areaClosestTo][2]) .. "]    "
                    .. "Closest [" .. string.format("%.2f", robotClosestToTarget[areaClosestTo]) .. "]    "
                    .. "Explored [" .. areaExploredOutput[areaClosestTo] .. "]    "
                    .. "Target Angle [" .. robotTargetExploreAngle .."]    "
                    .. "Heading [" .. robotCurrentHeading0to360 .. "]    "
                    .. "Difference [" .. robotDifferenceBetweenAngles .. "]    "
                    .. "Angle Rotated [" .. string.format("%.2f", rotateAccumulatedRotatedToTarget) .. "]    "
                    .. "Area Exploring [" .. exploringAreaOutput .. "]    "
                    .. "Areas Unexplored [" .. robotUnexploredAreas .. "]") -- Output robot exploration
information
            end -- End of the conditional statement
        elseif (areaClosestTo == 4) then -- If the robot is situated closer to the 'middle-left' region of the
environment, do the following
            for i = 1, 9, 1 do -- For all of the target areas specified, do the following
                if (i ~= areaClosestTo) then -- If the currently iterated area is not the area that the robot is
closest to, do the following
                    robotCurrentArea[i] = false -- The robot is not in the currently iterated area
                else -- If the currently iterated area is the area that the robot is closest to, do the following
                    robotCurrentArea[i] = true -- The robot is in the currently iterated area
                end -- End of the conditional statement
            end -- End of the iterative statement

            targetDifference[areaClosestTo] = math.sqrt(((targetPositions[areaClosestTo][1] -
targetClosestReached[areaClosestTo][1])^2) +
                                                        ((targetPositions[areaClosestTo][2] -
targetClosestReached[areaClosestTo][2])^2)) -- Store the difference between the robot and the
target for its exploration

            if (robotClosestToTarget[areaClosestTo] ~= -1) then -- If the robot has entered the area
before, do the following
```

```
        if (targetDifference[areaClosestTo] < robotClosestToTarget[areaClosestTo]) then -- If the
robots current distance to the area target is the closest it has been, do the following
          robotClosestToTarget[areaClosestTo] = targetDifference[areaClosestTo] -- Set the closest
distance to the current difference
        end -- End of the conditional statement
      else -- If the robot has not entered the area before, do the following
        robotClosestToTarget[areaClosestTo] = targetDifference[areaClosestTo] -- Store the current
distance of the robot to the target, as the closest distance
      end -- End of the conditional statement

      targetClosestReached[areaClosestTo][1] = robotPosition[1] -- Store the robots current 'X'
position as the closest achieved position to the area target position
      targetClosestReached[areaClosestTo][2] = robotPosition[2] -- Store the robots current 'Y'
position as the closest achieved position to the area target position

      if (robotExploredArea[areaClosestTo] == false) then -- If the area the robot is current
positioned in has not been explored, do the following
        if (targetDifference[areaClosestTo] < 0.6) then -- If the distance between the robots position
and area targets position is smaller than '0.6' metres, do the following
          robotExploredArea[areaClosestTo] = true -- Mark the area as explored

          robotUnexploredAreas = robotUnexploredAreas - 1 -- Decrement the number of areas
that are unexplored

          areaExploredOutput[areaClosestTo] = "Yes" -- Output the area as explored

          if (robotExploringArea == areaClosestTo) then -- If the area is currently the target area for
exploration, do the following
            robotExploredAreaSelected = true -- The robot searches for another unexplored area
          end -- End of the conditional statement
        else -- If the distance between the robots position and area targets position is larger than
desired distance, do the following
          areaExploredOutput[areaClosestTo] = "No" -- Output the robot has not explored the area
        end -- End of the conditional statement
      end -- End of the conditional statement

      previousTargetDifference[areaClosestTo] = targetDifference[areaClosestTo] -- Store the
current distance to the target as the previous distance to the target (end of frame)

      if (debugMode == false) then -- If debug mode is not active, do the following
        printf("Robot Location [Middle Left]    Robot Position in Area ["
          .. string.format("%.2f", targetClosestReached[areaClosestTo][1]) .. ", "
          .. string.format("%.2f", targetClosestReached[areaClosestTo][2]) .. "]    "
          .. "Closest [" .. string.format("%.2f", robotClosestToTarget[areaClosestTo]) .. "]    "
          .. "Explored [" .. areaExploredOutput[areaClosestTo] .. "]    "
          .. "Target Angle [" .. robotTargetExploreAngle .."]    "
          .. "Heading [" .. robotCurrentHeading0to360 .. "]    "
          .. "Difference [" .. robotDifferenceBetweenAngles .. "]    "
          .. "Angle Rotated [" .. string.format("%.2f", rotateAccumulatedRotatedToTarget) .. "]    "
          .. "Area Exploring [" .. exploringAreaOutput .. "]    "
```

```
          .. "Areas Unexplored [" .. robotUnexploredAreas .. "]") -- Output robot exploration
information
      end -- End of the conditional statement
    elseif (areaClosestTo == 5) then -- If the robot is situated closer to the 'centre' region of the
environment, do the following
      for i = 1, 9, 1 do -- For all of the target areas specified, do the following
        if (i ~= areaClosestTo) then -- If the currently iterated area is not the area that the robot is
closest to, do the following
          robotCurrentArea[i] = false -- The robot is not in the currently iterated area
        else -- If the currently iterated area is the area that the robot is closest to, do the following
          robotCurrentArea[i] = true -- The robot is in the currently iterated area
        end -- End of the conditional statement
      end -- End of the iterative statement

      targetDifference[areaClosestTo] = math.sqrt(((targetPositions[areaClosestTo][1] -
targetClosestReached[areaClosestTo][1])^2) +
                            ((targetPositions[areaClosestTo][2] -
targetClosestReached[areaClosestTo][2])^2)) -- Store the difference between the robot and the
target for its exploration

      if (robotClosestToTarget[areaClosestTo] ~= -1) then -- If the robot has entered the area
before, do the following
        if (targetDifference[areaClosestTo] < robotClosestToTarget[areaClosestTo]) then -- If the
robots current distance to the area target is the closest it has been, do the following
          robotClosestToTarget[areaClosestTo] = targetDifference[areaClosestTo] -- Set the closest
distance to the current difference
        end -- End of the conditional statement
      else -- If the robot has not entered the area before, do the following
        robotClosestToTarget[areaClosestTo] = targetDifference[areaClosestTo] -- Store the current
distance of the robot to the target, as the closest distance
      end -- End of the conditional statement

      targetClosestReached[areaClosestTo][1] = robotPosition[1] -- Store the robots current 'X'
position as the closest achieved position to the area target position
      targetClosestReached[areaClosestTo][2] = robotPosition[2] -- Store the robots current 'Y'
position as the closest achieved position to the area target position

      if (robotExploredArea[areaClosestTo] == false) then -- If the area the robot is current
positioned in has not been explored, do the following
        if (targetDifference[areaClosestTo] < 0.6) then -- If the distance between the robots position
and area targets position is smaller than '0.6' metres, do the following
          robotExploredArea[areaClosestTo] = true -- Mark the area as explored

          robotUnexploredAreas = robotUnexploredAreas - 1 -- Decrement the number of areas
that are unexplored

          areaExploredOutput[areaClosestTo] = "Yes" -- Output the area as explored

          if (robotExploringArea == areaClosestTo) then -- If the area is currently the target area for
exploration, do the following
            robotExploredAreaSelected = true -- The robot searches for another unexplored area
```

```
          end -- End of the conditional statement
        else -- If the distance between the robots position and area targets position is larger than
desired distance, do the following
          areaExploredOutput[areaClosestTo] = "No" -- Output the robot has not explored the area
        end -- End of the conditional statement
      end -- End of the conditional statement

      previousTargetDifference[areaClosestTo] = targetDifference[areaClosestTo] -- Store the
current distance to the target as the previous distance to the target (end of frame)

      if (debugMode == false) then -- If debug mode is not active, do the following
        printf("Robot Location [Centre]    Robot Position in Area ["
          .. string.format("%.2f", targetClosestReached[areaClosestTo][1]) .. ", "
          .. string.format("%.2f", targetClosestReached[areaClosestTo][2]) .. "]    "
          .. "Closest [" .. string.format("%.2f", robotClosestToTarget[areaClosestTo]) .. "]    "
          .. "Explored [" .. areaExploredOutput[areaClosestTo] .. "]    "
          .. "Target Angle [" .. robotTargetExploreAngle .."]    "
          .. "Heading [" .. robotCurrentHeading0to360 .. "]    "
          .. "Difference [" .. robotDifferenceBetweenAngles .. "]    "
          .. "Angle Rotated [" .. string.format("%.2f", rotateAccumulatedRotatedToTarget) .. "]    "
          .. "Area Exploring [" .. exploringAreaOutput .. "]    "
          .. "Areas Unexplored [" .. robotUnexploredAreas .. "]") -- Output robot exploration
information
      end -- End of the conditional statement
    elseif (areaClosestTo == 6) then -- If the robot is situated closer to the 'middle-right' region of
the environment, do the following
      for i = 1, 9, 1 do -- For all of the target areas specified, do the following
        if (i ~= areaClosestTo) then -- If the currently iterated area is not the area that the robot is
closest to, do the following
          robotCurrentArea[i] = false -- The robot is not in the currently iterated area
        else -- If the currently iterated area is the area that the robot is closest to, do the following
          robotCurrentArea[i] = true -- The robot is in the currently iterated area
        end -- End of the conditional statement
      end -- End of the iterative statement

      targetDifference[areaClosestTo] = math.sqrt(((targetPositions[areaClosestTo][1] -
targetClosestReached[areaClosestTo][1])^2) +
                            ((targetPositions[areaClosestTo][2] -
targetClosestReached[areaClosestTo][2])^2)) -- Store the difference between the robot and the
target for its exploration

      if (robotClosestToTarget[areaClosestTo] ~= -1) then -- If the robot has entered the area
before, do the following
        if (targetDifference[areaClosestTo] < robotClosestToTarget[areaClosestTo]) then -- If the
robots current distance to the area target is the closest it has been, do the following
          robotClosestToTarget[areaClosestTo] = targetDifference[areaClosestTo] -- Set the closest
distance to the current difference
        end -- End of the conditional statement
      else -- If the robot has not entered the area before, do the following
        robotClosestToTarget[areaClosestTo] = targetDifference[areaClosestTo] -- Store the current
distance of the robot to the target, as the closest distance
```

```
        end -- End of the conditional statement

        targetClosestReached[areaClosestTo][1] = robotPosition[1] -- Store the robots current 'X' position as the closest achieved position to the area target position
        targetClosestReached[areaClosestTo][2] = robotPosition[2] -- Store the robots current 'Y' position as the closest achieved position to the area target position

        if (robotExploredArea[areaClosestTo] == false) then -- If the area the robot is current positioned in has not been explored, do the following
            if (targetDifference[areaClosestTo] < 0.6) then -- If the distance between the robots position and area targets position is smaller than '0.6' metres, do the following
                robotExploredArea[areaClosestTo] = true -- Mark the area as explored

                robotUnexploredAreas = robotUnexploredAreas - 1 -- Decrement the number of areas that are unexplored

                areaExploredOutput[areaClosestTo] = "Yes" -- Output the area as explored

                if (robotExploringArea == areaClosestTo) then -- If the area is currently the target area for exploration, do the following
                    robotExploredAreaSelected = true -- The robot searches for another unexplored area
                end -- End of the conditional statement
            else -- If the distance between the robots position and area targets position is larger than desired distance, do the following
                areaExploredOutput[areaClosestTo] = "No" -- Output the robot has not explored the area
            end -- End of the conditional statement
        end -- End of the conditional statement

        previousTargetDifference[areaClosestTo] = targetDifference[areaClosestTo] -- Store the current distance to the target as the previous distance to the target (end of frame)

        if (debugMode == false) then -- If debug mode is not active, do the following
            printf("Robot Location [Middle Right]    Robot Position in Area ["
                .. string.format("%.2f", targetClosestReached[areaClosestTo][1]) .. ", "
                .. string.format("%.2f", targetClosestReached[areaClosestTo][2]) .. "]    "
                .. "Closest [" .. string.format("%.2f", robotClosestToTarget[areaClosestTo]) .. "]    "
                .. "Explored [" .. areaExploredOutput[areaClosestTo] .. "]    "
                .. "Target Angle [" .. robotTargetExploreAngle .."]    "
                .. "Heading [" .. robotCurrentHeading0to360 .. "]    "
                .. "Difference [" .. robotDifferenceBetweenAngles .. "]    "
                .. "Angle Rotated [" .. string.format("%.2f", rotateAccumulatedRotatedToTarget) .. "]    "
                .. "Area Exploring [" .. exploringAreaOutput .. "]    "
                .. "Areas Unexplored [" .. robotUnexploredAreas .. "]") -- Output robot exploration information
        end -- End of the conditional statement
    elseif (areaClosestTo == 7) then -- If the robot is situated closer to the 'bottom-left' region of the environment, do the following
        for i = 1, 9, 1 do -- For all of the target areas specified, do the following
            if (i ~= areaClosestTo) then -- If the currently iterated area is not the area that the robot is closest to, do the following
                robotCurrentArea[i] = false -- The robot is not in the currently iterated area
```

```
      else -- If the currently iterated area is the area that the robot is closest to, do the following
        robotCurrentArea[i] = true -- The robot is in the currently iterated area
      end -- End of the conditional statement
    end -- End of the iterative statement

    targetDifference[areaClosestTo] = math.sqrt(((targetPositions[areaClosestTo][1] -
targetClosestReached[areaClosestTo][1])^2) +
                              ((targetPositions[areaClosestTo][2] -
targetClosestReached[areaClosestTo][2])^2)) -- Store the difference between the robot and the
target for its exploration

    if (robotClosestToTarget[areaClosestTo] ~= -1) then -- If the robot has entered the area
before, do the following
      if (targetDifference[areaClosestTo] < robotClosestToTarget[areaClosestTo]) then -- If the
robots current distance to the area target is the closest it has been, do the following
        robotClosestToTarget[areaClosestTo] = targetDifference[areaClosestTo] -- Set the closest
distance to the current difference
      end -- End of the conditional statement
    else -- If the robot has not entered the area before, do the following
      robotClosestToTarget[areaClosestTo] = targetDifference[areaClosestTo] -- Store the current
distance of the robot to the target, as the closest distance
    end -- End of the conditional statement

    targetClosestReached[areaClosestTo][1] = robotPosition[1] -- Store the robots current 'X'
position as the closest achieved position to the area target position
    targetClosestReached[areaClosestTo][2] = robotPosition[2] -- Store the robots current 'Y'
position as the closest achieved position to the area target position

    if (robotExploredArea[areaClosestTo] == false) then -- If the area the robot is current
positioned in has not been explored, do the following
      if (targetDifference[areaClosestTo] < 1) then -- If the distance between the robots position
and area targets position is smaller than '1' metre, do the following
        robotExploredArea[areaClosestTo] = true -- Mark the area as explored

        robotUnexploredAreas = robotUnexploredAreas - 1 -- Decrement the number of areas
that are unexplored

        areaExploredOutput[areaClosestTo] = "Yes" -- Output the area as explored

        if (robotExploringArea == areaClosestTo) then -- If the area is currently the target area for
exploration, do the following
          robotExploredAreaSelected = true -- The robot searches for another unexplored area
        end -- End of the conditional statement
      else -- If the distance between the robots position and area targets position is larger than
desired distance, do the following
        areaExploredOutput[areaClosestTo] = "No" -- Output the robot has not explored the area
      end -- End of the conditional statement
    end -- End of the conditional statement

    previousTargetDifference[areaClosestTo] = targetDifference[areaClosestTo] -- Store the
current distance to the target as the previous distance to the target (end of frame)
```

```
      if (debugMode == false) then -- If debug mode is not active, do the following
        printf("Robot Location [Bottom Left]    Robot Position in Area ["
          .. string.format("%.2f", targetClosestReached[areaClosestTo][1]) .. ", "
          .. string.format("%.2f", targetClosestReached[areaClosestTo][2]) .. "]    "
          .. "Closest [" .. string.format("%.2f", robotClosestToTarget[areaClosestTo]) .. "]    "
          .. "Explored [" .. areaExploredOutput[areaClosestTo] .. "]    "
          .. "Target Angle [" .. robotTargetExploreAngle .."]    "
          .. "Heading [" .. robotCurrentHeading0to360 .. "]    "
          .. "Difference [" .. robotDifferenceBetweenAngles .. "]    "
          .. "Angle Rotated [" .. string.format("%.2f", rotateAccumulatedRotatedToTarget) .. "]    "
          .. "Area Exploring [" .. exploringAreaOutput .. "]    "
          .. "Areas Unexplored [" .. robotUnexploredAreas .. "]") -- Output robot exploration
information
      end -- End of the conditional statement
    elseif (areaClosestTo == 8) then -- If the robot is situated closer to the 'bottom-middle' region of
the environment, do the following
      for i = 1, 9, 1 do -- For all of the target areas specified, do the following
        if (i ~= areaClosestTo) then -- If the currently iterated area is not the area that the robot is
closest to, do the following
          robotCurrentArea[i] = false -- The robot is not in the currently iterated area
        else -- If the currently iterated area is the area that the robot is closest to, do the following
          robotCurrentArea[i] = true -- The robot is in the currently iterated area
        end -- End of the conditional statement
      end -- End of the iterative statement

      targetDifference[areaClosestTo] = math.sqrt(((targetPositions[areaClosestTo][1] -
targetClosestReached[areaClosestTo][1])^2) +
                           ((targetPositions[areaClosestTo][2] -
targetClosestReached[areaClosestTo][2])^2)) -- Store the difference between the robot and the
target for its exploration

      if (robotClosestToTarget[areaClosestTo] ~= -1) then -- If the robot has entered the area
before, do the following
        if (targetDifference[areaClosestTo] < robotClosestToTarget[areaClosestTo]) then -- If the
robots current distance to the area target is the closest it has been, do the following
          robotClosestToTarget[areaClosestTo] = targetDifference[areaClosestTo] -- Set the closest
distance to the current difference
        end -- End of the conditional statement
      else -- If the robot has not entered the area before, do the following
        robotClosestToTarget[areaClosestTo] = targetDifference[areaClosestTo] -- Store the current
distance of the robot to the target, as the closest distance
      end -- End of the conditional statement

      targetClosestReached[areaClosestTo][1] = robotPosition[1] -- Store the robots current 'X'
position as the closest achieved position to the area target position
      targetClosestReached[areaClosestTo][2] = robotPosition[2] -- Store the robots current 'Y'
position as the closest achieved position to the area target position

      if (robotExploredArea[areaClosestTo] == false) then -- If the area the robot is current
positioned in has not been explored, do the following
```

```
        if (targetDifference[areaClosestTo] < 0.6) then -- If the distance between the robots position
and area targets position is smaller than '0.6' metres, do the following
          robotExploredArea[areaClosestTo] = true -- Mark the area as explored

          robotUnexploredAreas = robotUnexploredAreas - 1 -- Decrement the number of areas
that are unexplored

          areaExploredOutput[areaClosestTo] = "Yes" -- Output the area as explored

          if (robotExploringArea == areaClosestTo) then -- If the area is currently the target area for
exploration, do the following
            robotExploredAreaSelected = true -- The robot searches for another unexplored area
          end -- End of the conditional statement
        else -- If the distance between the robots position and area targets position is larger than
desired distance, do the following
          areaExploredOutput[areaClosestTo] = "No" -- Output the robot has not explored the area
        end -- End of the conditional statement
      end -- End of the conditional statement

      previousTargetDifference[areaClosestTo] = targetDifference[areaClosestTo] -- Store the
current distance to the target as the previous distance to the target (end of frame)

      if (debugMode == false) then -- If debug mode is not active, do the following
        printf("Robot Location [Bottom Middle]    Robot Position in Area ["
          .. string.format("%.2f", targetClosestReached[areaClosestTo][1]) .. ", "
          .. string.format("%.2f", targetClosestReached[areaClosestTo][2]) .. "]    "
          .. "Closest [" .. string.format("%.2f", robotClosestToTarget[areaClosestTo]) .. "]    "
          .. "Explored [" .. areaExploredOutput[areaClosestTo] .. "]    "
          .. "Target Angle [" .. robotTargetExploreAngle .."]    "
          .. "Heading [" .. robotCurrentHeading0to360 .. "]    "
          .. "Difference [" .. robotDifferenceBetweenAngles .. "]    "
          .. "Angle Rotated [" .. string.format("%.2f", rotateAccumulatedRotatedToTarget) .. "]    "
          .. "Area Exploring [" .. exploringAreaOutput .. "]    "
          .. "Areas Unexplored [" .. robotUnexploredAreas .. "]") -- Output robot exploration
information
      end -- End of the conditional statement
    elseif (areaClosestTo == 9) then -- If the robot is situated closer to the 'bottom-right' region of
the environment, do the following
      for i = 1, 9, 1 do -- For all of the target areas specified, do the following
        if (i ~= areaClosestTo) then -- If the currently iterated area is not the area that the robot is
closest to, do the following
          robotCurrentArea[i] = false -- The robot is not in the currently iterated area
        else -- If the currently iterated area is the area that the robot is closest to, do the following
          robotCurrentArea[i] = true -- The robot is in the currently iterated area
        end -- End of the conditional statement
      end -- End of the iterative statement

      targetDifference[areaClosestTo] = math.sqrt(((targetPositions[areaClosestTo][1] -
targetClosestReached[areaClosestTo][1])^2) +
```

```
                                        ((targetPositions[areaClosestTo][2] -
targetClosestReached[areaClosestTo][2])^2)) -- Store the difference between the robot and the
target for its exploration

        if (robotClosestToTarget[areaClosestTo] ~= -1) then -- If the robot has entered the area
before, do the following
          if (targetDifference[areaClosestTo] < robotClosestToTarget[areaClosestTo]) then -- If the
robots current distance to the area target is the closest it has been, do the following
            robotClosestToTarget[areaClosestTo] = targetDifference[areaClosestTo] -- Set the closest
distance to the current difference
          end -- End of the conditional statement
        else -- If the robot has not entered the area before, do the following
          robotClosestToTarget[areaClosestTo] = targetDifference[areaClosestTo] -- Store the current
distance of the robot to the target, as the closest distance
        end -- End of the conditional statement

        targetClosestReached[areaClosestTo][1] = robotPosition[1] -- Store the robots current 'X'
position as the closest achieved position to the area target position
        targetClosestReached[areaClosestTo][2] = robotPosition[2] -- Store the robots current 'Y'
position as the closest achieved position to the area target position

        if (robotExploredArea[areaClosestTo] == false) then -- If the area the robot is current
positioned in has not been explored, do the following
          if (targetDifference[areaClosestTo] < 1.5) then -- If the distance between the robots position
and area targets position is smaller than '1.5' metres, do the following
            robotExploredArea[areaClosestTo] = true -- Mark the area as explored

            robotUnexploredAreas = robotUnexploredAreas - 1 -- Decrement the number of areas
that are unexplored

            areaExploredOutput[areaClosestTo] = "Yes" -- Output the area as explored

            if (robotExploringArea == areaClosestTo) then -- If the area is currently the target area for
exploration, do the following
              robotExploredAreaSelected = true -- The robot searches for another unexplored area
            end -- End of the conditional statement
          else -- If the distance between the robots position and area targets position is larger than
desired distance, do the following
            areaExploredOutput[areaClosestTo] = "No" -- Output the robot has not explored the area
          end -- End of the conditional statement
        end -- End of the conditional statement

        previousTargetDifference[areaClosestTo] = targetDifference[areaClosestTo] -- Store the
current distance to the target as the previous distance to the target (end of frame)

        if (debugMode == false) then -- If debug mode is not active, do the following
          printf("Robot Location [Bottom Right]    Robot Position in Area ["
            .. string.format("%.2f", targetClosestReached[areaClosestTo][1]) .. ", "
            .. string.format("%.2f", targetClosestReached[areaClosestTo][2]) .. "]    "
            .. "Closest [" .. string.format("%.2f", robotClosestToTarget[areaClosestTo]) .. "]    "
            .. "Explored [" .. areaExploredOutput[areaClosestTo] .. "]    "
```

```
               .. "Target Angle [" .. robotTargetExploreAngle .."]    "
               .. "Heading [" .. robotCurrentHeading0to360 .. "]    "
               .. "Difference [" .. robotDifferenceBetweenAngles .. "]    "
               .. "Angle Rotated [" .. string.format("%.2f", rotateAccumulatedRotatedToTarget) .. "]    "
               .. "Area Exploring [" .. exploringAreaOutput .. "]    "
               .. "Areas Unexplored [" .. robotUnexploredAreas .. "]") -- Output robot exploration information
         end -- End of the conditional statement
      end -- End of the conditional statement
   end -- End of the conditional statement
   end -----[ WANDERING ]-----

   do -----[ MAPPING ]-----
   sonarReadings = {-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1} -- Set sonar readings (sensors stay at '-1' if an object was not detected)

   for i = 1, 16, 1 do -- For all of the robots sensors, do the following
      result, distance = sim.readProximitySensor(sonarSensors[i]) -- Determine whether an object was detected and at what distance

      if (result > 0) then -- If an object was detected, do the following
         sonarReadings[i] = distance -- Set the currently iterated sonar sensor reading to the detected distance
      end -- End of the conditional statement
   end -- End of the iterative statement
   end -----[ MAPPING ]-----

   do -----[ CONSOLE ROBOT DETECTED OBJECT POSITION CALCULATIONS ]-----

      smallestLeftValue = { 0, 0 } -- Create and initialise an array for storing the sensor and distance to the closest object detected, relative to the robots left side
      smallestRightValue = { 0, 0 } -- Create and initialise an array for storing the sensor and distance to the closest object detected, relative to the robots right side

      smallestLeftValueSet = false -- Determine whether the initial value of the robots left detection has been set for comparing in future iterations
      smallestRightValueSet = false -- Determine whether the initial value of the robots right detection has been set for comparing in future iterations

      leftDetections = 0 -- Store the number of sensors that have detected an object, relative to the left side of the robot
      rightDetections = 0 -- Store the number of sensors that have detected an object, relative to the right side of the robot

      for i = 1, 4, 1 do -- For the robots front-left facing sensors, do the following
         if (sonarReadings[i] > 0) then -- If the distance detected by the currently iterated sensor is larger than '0', do the following
            leftDetections = leftDetections + 1 -- Increment the left detection counter
         end -- End of the conditional statement
      end -- End of the iterative statement
```

```
    for i = 13, 16, 1 do -- For the robots back-left facing sensors, do the following
      if (sonarReadings[i] > 0) then -- If the distance detected by the currently iterated sensor is
larger than '0', do the following
        leftDetections = leftDetections + 1 -- Increment the left detection counter
      end -- End of the conditional statement
    end -- End of the iterative statement

    for i = 5, 12, 1 do -- For the robots right facing sensors, do the following
      if (sonarReadings[i] > 0) then -- If the distance detected by the currently iterated sensor is
larger than '0', do the following
        rightDetections = rightDetections + 1 -- Increment the right detection counter
      end -- End of the conditional statement
    end -- End of the iterative statement

    if (leftDetections + rightDetections > 0) then -- If a robots sensor has detected an object, do the
following
      for i = 1, 16, 1 do -- For all of the robots sonar sensors, do the following
        if (sonarReadings[i] > 0) then -- If the distance detected by the currently iterated sensor is
larger than '0', do the following
          if (i >= 1 and i <= 4) then -- If the currently iterated sensor is a front-left facing sensor, do
the following
            if (smallestLeftValueSet == false) then -- If the initial left detection value has been set,
do the following
              smallestLeftValue[1] = i -- Store the sensor that has detected the closest object to the
robot
              smallestLeftValue[2] = sonarReadings[i] -- Store the distance detected to the closest
object to the robot

              smallestLeftValueSet = true -- The initial left detection value has been set
            else -- If the initial left detection value has not been set, do the following
              if (sonarReadings[i] < smallestLeftValue[2]) then -- If current distance detected by the
currently iterated sensor is smaller than the current smallest distance detected to an object, do the
following
                smallestLeftValue[1] = i -- Store the sensor that has detected the closest object to
the robot
                smallestLeftValue[2] = sonarReadings[i] -- Store the distance detected from the
sensor to the closest object
              end -- End of the conditional statement
            end -- End of the conditional statement
          elseif (i >= 13 and i <= 16) then -- If the currently iterated sensor is a back-left facing
sensor, do the following
            if (sonarReadings[i] < smallestLeftValue[2]) then -- If current distance detected by the
currently iterated sensor is smaller than the current smallest distance detected to an object, do the
following
              smallestLeftValue[1] = i -- Store the sensor that has detected the closest object to the
robot
              smallestLeftValue[2] = sonarReadings[i] -- Store the distance detected from the
sensor to the closest object
            end -- End of the conditional statement
          elseif (i >= 5 and i <= 8) then -- If the currently iterated sensor is a front-right facing
sensor, do the following
```

```
            if (smallestRightValueSet == false) then -- If the initial right detection value has been set, do the following
              smallestRightValue[1] = i -- Store the sensor that has detected the closest object to the robot
              smallestRightValue[2] = sonarReadings[i] -- Store the distance detected from the sensor to the closest object

              smallestRightValueSet = true -- The initial right detection value has been set
            else -- If the initial right detection value has been set, do the following
              if (sonarReadings[i] < smallestRightValue[2]) then -- If current distance detected by the currently iterated sensor is smaller than the current smallest distance detected to an object, do the following
                smallestRightValue[1] = i -- Store the sensor that has detected the closest object to the robot
                smallestRightValue[2] = sonarReadings[i] -- Store the distance detected from the sensor to the closest object
              end -- End of the conditional statement
            end -- End of the conditional statement
          elseif (i >= 9 and i <= 12) then -- If the currently iterated sensor is a back-right facing sensor, do the following
            if (sonarReadings[i] < smallestRightValue[2]) then -- If current distance detected by the currently iterated sensor is smaller than the current smallest distance detected to an object, do the following
              smallestRightValue[1] = i -- Store the sensor that has detected the closest object to the robot
              smallestRightValue[2] = sonarReadings[i] -- Store the distance detected from the sensor to the closest object
            end -- End of the conditional statement
          end -- End of the conditional statement

          leftMostDetectedObject[1] = smallestLeftValue[1] -- Set the left facing sensor that has detected an object as the closest distance
          leftMostDetectedObject[2] = smallestLeftValue[2] -- Set the closest object detected on the left side of the robot to the smallest distance detected by a left facing sensor

          rightMostDetectedObject[1] = smallestRightValue[1] -- Set the right facing sensor that has detected an object as the closest distance
          rightMostDetectedObject[2] = smallestRightValue[2] -- Set the closest object detected on the right side of the robot to the smallest distance detected by a right facing sensor
        end -- End of the conditional statement
      end -- End of the iterative statement
    end -- End of the conditional statement

    if (leftDetections == 0 or leftMostDetectedObject[2] == 0) then
      leftString = "NO OBJECT" -- Set the robots left detection string to 'no object' (no object detected)
    else
      leftString = "Sensor [" .. leftMostDetectedObject[1] .. "] at " .. string.format("%.2f", leftMostDetectedObject[2]) .. " m" -- Set the robots left detection string to the sensor that has detected an object at the closest distance and the corresponding distance detected
    end -- End of the conditional statement
```

```
    if (rightDetections == 0 or rightMostDetectedObject[2] == 0) then
      rightString = "NO OBJECT" -- Set the robots right detection string to 'no object' (no object detected)
    else
      rightString = "Sensor [" .. rightMostDetectedObject[1] .. "] at " .. string.format("%.2f", rightMostDetectedObject[2]) .. " m" -- Set the robots right detection string to the sensor that has detected an object at the closest distance and the corresponding distance detected
    end -- End of the conditional statement

  end -----[ CONSOLE ROBOT DETECTED OBJECT POSITION CALCULATIONS ]-----

  do -----[ SENSING ]-----
    accumulatedDistance = 0 -- Reset the accumulated distance

    for i = 1, 16, 1 do -- For all of the robots sonar sensors, do the following
      result, distance = sim.readProximitySensor(sonarSensors[i]) -- Store whether the currently iterated sonar sensor detected an object and its distance from the robots position if so
      if (result > 0) and (distance < noDetectionDistance) then -- If the currently iterated sonar sensor detected an object and its distance from the robots current position is within the robots detectable range, do the following
        if (distance < maxDetectionDistance) then -- If the objects detected distance from the robots current position is wihtin the robots maximum detection distance, do the following
          distance = maxDetectionDistance -- Set the objects detected distance to the robots maximum detection distance
        end -- End of the conditional statement
        objectDetected[i] = 1 - ((distance - maxDetectionDistance) / (noDetectionDistance - maxDetectionDistance)) -- Set the currently iterated object detection array element (parallel to the sonar sensor) to the calculated distance detected
        detectedDistance[i] = distance -- Store the curently iterated sensor distance reading into the detected distance array
      else -- Else if the currently iterated sonar sensor has not detected an object, do the following
        objectDetected[i] = 0 -- Set the currently iterated object detection array element (parallel to the sonar sensor) to '0' (no object detected)
        detectedDistance[i] = 0 -- Set the currently iterated detected distance array element (parallel to the sonar sensor) to '0' (no object detected)
      end -- End of the conditional statement
    end -- End of the iterative statement

  if (edgeEndReached == false) then -- If the robot has not reached the edge of a followed object or has not entered the 'edge following' phase, do the following

    edgeEndReachedAvoidTimer = math.random(3, 5) -- Generate a random duration of time for the robot to avoid for, when the robot has finished following an edge of an object (prevent edge following loop)

    if (robotIsAvoiding == false) then -- If the robot is not in the 'avoiding' phase, do the following
      for i = 1, 8, 1 do -- For all of the robots sonar sensors, do the following
        if (detectedDistance[i] > 0) then -- If an object has been detected from the currently iterated sonar sensor, do the following
```

```
                    accumulatedDistance = accumulatedDistance + detectedDistance[i] -- Add and equal the current accumulated distance with the currently iterated sonar sensor reading
                end -- End of the conditional statement
            end -- End of the iterative statement

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

                if (detectedDistance[1] > 0 and detectedDistance[3] == 0) then -- If the robots left-most front facing sensor has detected an object and one of the robots front-left sensors has not detected an object (straighten), do the following

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

                    sensorDetectedIncrementer = 0 -- Reset the number of sensors with a detected distance

                    for i = 4, 8, 1 do -- For the robots other front facing sensors, do the following
                        if (detectedDistance[i] > 0) then -- If the currently iterated sensor has detected an object, do the following
                            sensorDetectedIncrementer = sensorDetectedIncrementer + 1 -- Increment the number of sensors that have detected an object
                        end -- End of the conditional statement
                    end -- End of the iterative statement

                    if (sensorDetectedIncrementer == 0) then -- If the robots other front facing sensors have not detected an object, do the following
                        edgeFollowingLeftDetected = true -- Set the robot to enter edge following for its left side
                    else -- If the robots other front facing sensors have detected an object, do the following
                        edgeFollowingLeftDetected = false -- The robot will not enter (or will exit) edge following for its left side
                    end -- End of the conditional statement

                elseif (detectedDistance[8] > 0 and detectedDistance[6] == 0) then -- If the robots right-most front facing sensor has detected an object and one of the robots front-right sensors has not detected an object (straighten), do the following

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

                    sensorDetectedIncrementer = 0 -- Reset the number of sensors with a detected distance

                    for i = 1, 5, 1 do -- For the robots other front facing sensors, do the following
                        if (detectedDistance[i] > 0) then -- If the currently iterated sensor has detected an object, do the following
                            sensorDetectedIncrementer = sensorDetectedIncrementer + 1 -- Increment the number of sensors that have detected an object
                        end -- End of the conditional statement
                    end -- End of the iterative statement
```

```
                    if (sensorDetectedIncrementer == 0) then -- If the robots other front facing sensors have not detected an object, do the following
                        edgeFollowingRightDetected = true -- Set the robot to enter edge following for its right side
                    else -- If the robots other front facing sensors have detected an object, do the following
                        edgeFollowingRightDetected = false -- The robot will not enter (or will exit) edge following for its right side
                    end -- End of the conditional statement
                else -- If the robots left-most or right-most front-facing sensor does not detect an object when their other front facing sensor does not detect an object, do the following
                    edgeFollowingLeftDetected = false -- The robot will not enter (or will exit) edge following for its left side
                    edgeFollowingRightDetected = false -- The robot will not enter (or will exit) edge following for its right side
                end -- End of the conditional statement

                if (edgeFollowingLeftDetected == true or edgeFollowingRightDetected == true) then -- If the robot has been set to follow the edge of an object on its right or left side, do the following
                    robotIsEdgeFollowing = true -- Robot enters the 'edge following' state
                    edgeEndReached = false -- The end of an objects followed edge has not been reached

                    edgeFollowingTimer = edgeFollowingTimer + sim.getSimulationTimeStep() -- Subtract and equal the edge following timer for the time passed since the last frame was made
                else -- If the robot has not been set to follow the edge of an object on either of its sides, do the following
                    robotIsEdgeFollowing = false -- Robot exits the 'edge following' phase
                    robotIsAvoiding = true -- Robot enters the 'avoiding' phase

                    if (edgeFollowingTimer > 3) then -- If the time the robot has been following an edge of an object is larger than '3' seconds, do the following
                        --edgeEndReached = true -- The end of an objects followed edge has been reached
                    else -- If the time the robot has been
                        edgeFollowingTimer = 0 -- Reset the edge following timer
                    end -- End of the conditional statement
                end -- End of the conditional statement

            else -- If an object has not been detected, do the following
                robotIsAvoiding = false -- Robot exits the 'avoiding' phase
                robotIsEdgeFollowing = false -- Robot exits the edge following' phase
                edgeEndReached = false -- The end of an objects followed edge has not been reached
            end -- End of the conditional statement
        end -- End of the conditional statement

    else -- If the end of an objects followed edge has been reached, do the following

        if (robotIsAvoiding == false) then -- If the robot has not entered the 'avoiding' phase, do the following
            robotIsAvoiding = true -- Robot enters the 'avoiding' phase
        end -- End of the conditional statement

    end -- End of the conditional statement
```

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


end -- End of the function declaration




function sysCall_actuation() -- Robot actuation functionality

    do -----[ CONSOLE ROBOT SPEED CALCULATIONS ]-----
    Distance = { 0, 0, 0 } -- Create and initialise an array for the difference between the robots

    currentPosition = sim.getObjectPosition(sim.getObjectHandle("Pioneer_p3dx"), -1) -- Store the robots current position

    for i = 1, 3, 1 do -- For the number of elements in the table/ array, do the following
        Distance[i] = currentPosition[i] - previousPosition[i] -- Calculate the difference between the positions
    end -- End of the iterative statement

    robotDistanceTravelled = math.sqrt((Distance[1] * Distance[1]) + (Distance[2] * Distance[2]) + (Distance[3] * Distance[3])) -- Set the robots distance travelled to the magnitude of the difference between vectors (previous and current positions)

    Speed = robotDistanceTravelled / sim.getSimulationTimeStep() -- Speed = Distance / Time

    robotDistanceTravelled = 0 -- Reset the robots distance travelled

    robotSpeed = Speed -- Set the robots speed to the calculate speed

    Speed = 0 -- Reset the robots calculated speed (local)

    previousPosition = currentPosition -- Set the robots previous position to the robots current position (end of frame)
    end -----[ CONSOLE ROBOT SPEED CALCULATIONS ]-----

    do -----[ MAPPING ]-----
        calculateMapping() -- Function call, calculate the robots sonar readings (unconditional)
    end -----[ MAPPING ]-----

    if (robotIsAvoiding == true) then -- If the robot 'avoiding' phase has been triggered, do the following
        robotAvoiding() -- Function call, set the robot to avoid obstacles
    elseif (robotIsEdgeFollowing == true) then -- Else if the robot 'edge-following' phase has been triggered, do the following
        robotEdgeFollowing() -- Function call, set the robot to follow an edge
    else -- Else if the robot is neither 'avoiding' or 'edge-following', do the following
        robotWandering() -- Function call, set the robot to wander the scene
    end -- End of the conditional statement
```

```
end -- End of the function declaration



function robotAvoiding() -- Robot avoidance strategy

    do -----[ FINISHED FOLLOWING EDGE TRANSITION ]-----
    if (edgeEndReached == true) then -- If the edge of an object has been reach after following it, do the following

        edgeEndReachedAvoidTimer = edgeEndReachedAvoidTimer - sim.getSimulationTimeStep() -- Subtract the time passed since the last frame was made away from the robot avoid timer

        if (edgeEndReachedAvoidTimer < 0) then -- If the robot avoid timer has depleted, do the following
            edgeEndReached = false -- The robot has reached the edge of an object it was following and has moved away from detected objects (prevent edge following loop)
            edgeFollowingTimer = 0 -- Reset the edge following timer
        end -- End of the conditional statement
    end -- End of the conditional statement
    end -----[ FINISHED FOLLOWING EDGE TRANSITION ]-----


    robotWanderingReset = true -- The robots 'wandering' phase configuration requires to be reset (was interrupted)

    do -----[ CALCULATIONS AND GENERAL AVOIDANCE ]-----
    if (robotIsReversing == false and robotIsTurning == false) then -- If the robot has not entered the 'reversing' and 'turning' phase, do the following

        distanceComparison = 0 -- Reset the comparative distance between sensors

        for i = 3, 4, 1 do -- For the robots front facing sensors, do the following
            if (detectedDistance[i] and detectedDistance[9 - i] > 0) then -- If the currently iterated sensors detected distance and its opposing sensors detected distance is larger than '0' (object was detected), do the following
                if (detectedDistance[i] - detectedDistance[9 - i] < 0) then -- If the difference between the currently iterated sensors detected distance and its opposing sensors detected distance is smaller than '0m' (negative), do the following
                    distanceComparison = -(detectedDistance[i] - detectedDistance[9 - i]) -- Set the comparative distance to the difference between the sensors detected distance (positively)
                elseif (detectedDistance[i] - detectedDistance[9 - i] > 0) then -- If the difference between the currently iterated sensors detected distance and its opposing sensors detected distance is larger than '0m' (positive), do the following
                    distanceComparison = detectedDistance[i] - detectedDistance[9 - i] -- Set the comparative distance to the difference between the sensors detected distance
                else -- If the detected distances are equal, do the following
                    distanceComparison = 0 -- Set the comparative distance to '0'
                end -- End of the conditional statement
```

```lua
                if (distanceComparison < 0.005) then -- If the difference between the sensors detected distances is smaller than '0.005m', do the following

                    if (debugMode == true) then -- If debug mode is active, do the following
                        --print("Equal distance to object detected from sensors [" .. i .. "] and [" .. (9 - i) .. "]") -- Output the sensors that have detected object(s) at an equal distance
                    end -- End of conditional statement

                    robotIsReversing = true -- Set the robot to enter its 'reversing' phase
                    reverseTurnTimer = math.random(5, 20) / 10 -- Generate a time amount for the robot to turn away from a detected object
                    rotationDirection = math.random(1, 2) -- Generate a rotation direction determining the direction the robot turns after reversing
                end -- End of the conditional statement
            end -- End of the conditional statement
        end -- End of the iterative statement



        do -----[ BRAITENBERG AVOIDANCE ]-----
        if (robotIsReversing == false) then -- If the robot has not entered the 'reversing' phase, do the following
            if (robotIsTurning == false) then -- If the robot has not entered the 'turning' phase, do the following
                if (robotIsStuck == false) then -- If the robot has not entered the 'stuck' phase, do the following

                    accumulatedFrontLeftSensorDistance = 0 -- Reset the robots accumulated front-left sensor distance
                    accumulatedFrontRightSensorDistance = 0 -- Reset the robots accumulated right-left sensor distance

                    for i = 1, 4, 1 do -- For all of the robots front-left sensors, do the following
                        accumulatedFrontLeftSensorDistance = accumulatedFrontLeftSensorDistance + detectedDistance[i] -- Add and equal the distance detected by the currently iterated front-left sensor
                    end -- End of the iterative statement

                    for i = 5, 8, 1 do -- For all of the robots front-right sensors, do the following
                        accumulatedFrontRightSensorDistance = accumulatedFrontRightSensorDistance + detectedDistance[i] -- Add and equal the distance detected by the currently iterated front-right sensor
                    end -- End of the iterative statement

                    leftWheelVelocity = defaultVelocity -- Set the robots left wheel motor velocity to the default velocity
                    rightWheelVelocity = defaultVelocity -- Set the robots right wheel motor velocity to the default velocity

                    frontSensorDistanceDifference = accumulatedFrontLeftSensorDistance - accumulatedFrontRightSensorDistance -- Store the difference in distance between either front facing side of sensors
```

```
          if (frontSensorDistanceDifference < 0) then -- If the difference in distance between either
front facing side of sensors is smaller than '0' (negative), do the following
            frontSensorDistanceDifference = -(frontSensorDistanceDifference) -- Negate the
difference in distance between either front facing side of sensors
          end -- End of conditional statement

          if (frontSensorDistanceDifference <= (noDetectionDistance - maxDetectionDistance) +
0.01) then -- If the difference in distance between the sensor readings is smaller than or equal to the
difference in distance between the robots no detection distance and the robots maximum detection
distance (prevent oscillation), do the following
            if (edgeEndReached == true) then -- If the robot has finished following an objects edge,
do the following
              if (debugMode == true) then -- If debug mode is active, do the following
                printf("Speed [" .. string.format("%.2f", robotSpeed) .. " m/s]    Heading [" ..
string.format("%.2f", robotHeading) .. " DEG]    "
                  .. "Position [" .. string.format("%.2f", robotPosition[1]) .. ", " ..
string.format("%.2f", robotPosition[2]) .. "]    "
                  .. "Detection Left [" .. leftString .. "]    " .. "Detection Right [" .. rightString .. "]    "
                  .. "State [Avoiding] -----> Moving Forwards [End of Followed Edge Reached]    "
                  .. "RANSAC [" .. string.format("%.2f", ransacTargetCompletion) .. " PCT] of [" ..
ransacTarget .. "]") -- Output the robot is moving forwards as the end of an objects followed edge
has been reached
              end -- End of the conditional statement
            else -- If the robot has not finished following an objects edge, do the following
              if (debugMode == true) then -- If debug mode is active, do the following
                printf("Speed [" .. string.format("%.2f", robotSpeed) .. " m/s]    Heading [" ..
string.format("%.2f", robotHeading) .. " DEG]    "
                  .. "Position [" .. string.format("%.2f", robotPosition[1]) .. ", " ..
string.format("%.2f", robotPosition[2]) .. "]    "
                  .. "Detection Left [" .. leftString .. "]    " .. "Detection Right [" .. rightString .. "]    "
                  .. "State [Avoiding] -----> Moving Forwards    "
                  .. "RANSAC [" .. string.format("%.2f", ransacTargetCompletion) .. " PCT] of [" ..
ransacTarget .. "]") -- Output the robot is moving forwards
              end -- End of the conditional statement
            end -- End of the conditional statement

            leftWheelVelocity = defaultVelocity -- Set the robots left wheels motor velocity to the
default velocity
            rightWheelVelocity = defaultVelocity -- Set the robots right wheels motor velocity to
the default velocity

          elseif (accumulatedFrontLeftSensorDistance > accumulatedFrontRightSensorDistance)
then -- If the front-left sensors detected distance is larger than the front-right sensors detected
distance, do the following
            if (edgeEndReached == true) then -- If the robot has finished following an objects edge,
do the following
              if (debugMode == true) then -- If debug mode is active, do the following
                printf("Speed [" .. string.format("%.2f", robotSpeed) .. " m/s]    Heading [" ..
string.format("%.2f", robotHeading) .. " DEG]    "
```

```
                  .. "Position [" .. string.format("%.2f", robotPosition[1]) .. ", " ..
string.format("%.2f", robotPosition[2]) .. "]    "
                  .. "Detection Left [" .. leftString .. "]    " .. "Detection Right [" .. rightString .. "]    "
                  .. "State [Avoiding] -----> Turning Right [End of Followed Edge Reached]    "
                  .. "RANSAC [" .. string.format("%.2f", ransacTargetCompletion) .. " PCT] of [" ..
ransacTarget .. "]") -- Output the robot is turning right as the end of an objects followed edge has
been reached
              end -- End of the condtional statement
            else -- If the robot has not finished following an objects edge, do the following
              if (debugMode == true) then -- If debug mode is active, do the following
                printf("Speed [" .. string.format("%.2f", robotSpeed) .. " m/s]    Heading [" ..
string.format("%.2f", robotHeading) .. " DEG]    "
                  .. "Position [" .. string.format("%.2f", robotPosition[1]) .. ", " ..
string.format("%.2f", robotPosition[2]) .. "]    "
                  .. "Detection Left [" .. leftString .. "]    " .. "Detection Right [" .. rightString .. "]    "
                  .. "State [Avoiding] -----> Turning Right    "
                  .. "RANSAC [" .. string.format("%.2f", ransacTargetCompletion) .. " PCT] of [" ..
ransacTarget .. "]") -- Output the robot is turning right
              end -- End of the conditional statement
            end -- End of the conditional statement

            for i = 1, 8, 1 do -- For all of the robots front facing sensors, do the following
              rightWheelVelocity = rightWheelVelocity + braitenbergRight[i] * objectDetected[i] --
Set the robots right wheels motor velocity to the calculated velocity (Braitenberg)
            end -- End of the iterative statement
          elseif (accumulatedFrontRightSensorDistance > accumulatedFrontLeftSensorDistance)
then -- If the front-right sensors detected distance is larger than the front-left sensors detected
distance, do the following
            if (edgeEndReached == true) then -- If the robot has finished following an objects edge,
do the following
              if (debugMode == true) then -- If debug mode is active, do the following
                printf("Speed [" .. string.format("%.2f", robotSpeed) .. " m/s]    Heading [" ..
string.format("%.2f", robotHeading) .. " DEG]    "
                  .. "Position [" .. string.format("%.2f", robotPosition[1]) .. ", " ..
string.format("%.2f", robotPosition[2]) .. "]    "
                  .. "Detection Left [" .. leftString .. "]    " .. "Detection Right [" .. rightString .. "]    "
                  .. "State [Avoiding] -----> Turning Left [End of Followed Edge Reached]    "
                  .. "RANSAC [" .. string.format("%.2f", ransacTargetCompletion) .. " PCT] of [" ..
ransacTarget .. "]") -- Output the robot is turning left as the end of an objects followed edge has
been reached
              end -- End of the conditional statement
            else -- If the robot has not finished following an objects edge, do the following
              if (debugMode == true) then -- If debug mode is active, do the following
                printf("Speed [" .. string.format("%.2f", robotSpeed) .. " m/s]    Heading [" ..
string.format("%.2f", robotHeading) .. " DEG]    "
                  .. "Position [" .. string.format("%.2f", robotPosition[1]) .. ", " ..
string.format("%.2f", robotPosition[2]) .. "]    "
                  .. "Detection Left [" .. leftString .. "]    " .. "Detection Right [" .. rightString .. "]    "
                  .. "State [Avoiding] -----> Turning Left    "
                  .. "RANSAC [" .. string.format("%.2f", ransacTargetCompletion) .. " PCT] of [" ..
ransacTarget .. "]") -- Output the robot is turning left
```

```
                     end -- End of the conditional statement
                  end -- End of the conditional statement

                  for i = 1, 8, 1 do -- For all of the robots front facing sensors, do the following
                     leftWheelVelocity = leftWheelVelocity + braitenbergLeft[i] * objectDetected[i] -- Set
the robots left wheels motor velocity to the calculated velocity (Braitenberg)
                  end -- End of the iterative statement
               else -- If the sensors detected distance is equal, do the following
                  if (edgeEndReached == true) then -- If the robot has finished following an objects edge,
do the following
                     if (debugMode == true) then -- If debug mode is active, do the following
                        printf("Speed [" .. string.format("%.2f", robotSpeed) .. " m/s]    Heading [" ..
string.format("%.2f", robotHeading) .. " DEG]    "
                           .. "Position [" .. string.format("%.2f", robotPosition[1]) .. ", " ..
string.format("%.2f", robotPosition[2]) .. "]    "
                           .. "Detection Left [" .. leftString .. "]    " .. "Detection Right [" .. rightString .. "]    "
                           .. "State [Avoiding] -----> Moving Forwards [End of Followed Edge Reached]    "
                           .. "RANSAC [" .. string.format("%.2f", ransacTargetCompletion) .. " PCT] of [" ..
ransacTarget .. "]") -- Output the robot is moving forwards as the end of an objects followed edge
has been reached
                     end -- End of the conditional statement
                  else -- If the robot has not finished following an objects edge, do the following
                     if (debugMode == true) then -- If debug mode is active, do the following
                        printf("Speed [" .. string.format("%.2f", robotSpeed) .. " m/s]    Heading [" ..
string.format("%.2f", robotHeading) .. " DEG]    "
                           .. "Position [" .. string.format("%.2f", robotPosition[1]) .. ", " ..
string.format("%.2f", robotPosition[2]) .. "]    "
                           .. "Detection Left [" .. leftString .. "]    " .. "Detection Right [" .. rightString .. "]    "
                           .. "State [Avoiding] -----> Moving Forwards    "
                           .. "RANSAC [" .. string.format("%.2f", ransacTargetCompletion) .. " PCT] of [" ..
ransacTarget .. "]") -- Output the robot is moving forwards
                     end -- End of the conditional statement
                  end -- End of the conditional statement

                  leftWheelVelocity = defaultVelocity -- Set the robots left wheels motor velocity to the
default velocity
                  rightWheelVelocity = defaultVelocity -- Set the robots right wheels motor velocity to
the default velocity
               end -- End of the conditional statement

               sim.setJointTargetVelocity(leftWheelMotor, leftWheelVelocity) -- Set the robots left
wheels motor velocity to the calculated velocity
               sim.setJointTargetVelocity(rightWheelMotor, rightWheelVelocity) -- Set the robots right
wheels motor velocity to the calculated velocity

            end -- End of conditional statement
         end -- End of conditional statement
      end -- End of conditional statement
      end -----[ BRAITENBERG AVOIDANCE ]-----
   end -- End of conditional statement
   end -----[ CALCULATIONS AND GENERAL AVOIDANCE ]-----
```

```
    do -----[ REVERSING ]-----
    if (robotIsStuck == false) then -- If the robot has not entered the 'stuck' phase, do the following
        if (robotIsTurning == false) then -- If the robot has not entered the 'turning' phase, do the following
            if (robotIsReversing == false) then -- If the robot has not entered the 'reversing' phase, do the following
                -- Reset variables (if required)
            else -- If the robot has entered the 'reversing' phase, do the following

                accumulatedFrontDistance = 0 -- Reset the accumulated distance of the robots front-facing sensors
                accumulatedBackDistance = 0 -- Reset the accumulated distance of the robots back-facing sensors

                for i = 1, 7, 1 do -- For all of the robots front-facing sensors (not 90 degree facing sensor), do the following
                    accumulatedFrontDistance = accumulatedFrontDistance + detectedDistance[i] -- Add and equal the distance detected by the currently iterated front-facing sensor
                end -- End of the iterative statement

                for i = 9, 15, 1 do -- For all of the robots back-facing sensors (not 90 degree facing sensor), do the following
                    accumulatedBackDistance = accumulatedBackDistance + detectedDistance[i] -- Add and equal the distance detected by the currently iterated back-facing sensor
                end -- End of the iterative statement

                if (accumulatedFrontDistance > 0) then -- If the robots front-facing sensors have detected an object, do the following
                    if (accumulatedBackDistance == 0) then -- If the robots back facing sensors have not detected an object, do the following
                        if (debugMode == true) then -- If debug mode is active, do the following
                            printf("Speed [" .. string.format("%.2f", robotSpeed) .. " m/s]    Heading [" .. string.format("%.2f", robotHeading) .. " DEG]    "
                                .. "Position [" .. string.format("%.2f", robotPosition[1]) .. ", " .. string.format("%.2f", robotPosition[2]) .. "]    "
                                .. "Detection Left [" .. leftString .. "]    " .. "Detection Right [" .. rightString .. "]    "
                                .. "State: [Avoiding] -----> Reversing    "
                                .. "RANSAC [" .. string.format("%.2f", ransacTargetCompletion) .. " PCT] of [" .. ransacTarget .. "]") -- Output the robot is reversing
                        end -- End of the conditional statement

                        leftWheelVelocity = -(defaultVelocity) -- Set the robots left wheels motor velocity to the default velocity negated
                        rightWheelVelocity = -(defaultVelocity) -- Set the robots right wheels motor velocity to the default velocity negated

                        sim.setJointTargetVelocity(leftWheelMotor, leftWheelVelocity) -- Set the robots left wheels motor velocity to the calculated velocity
```

```
                  sim.setJointTargetVelocity(rightWheelMotor, rightWheelVelocity) -- Set the robots
right wheels motor velocity to the calculated velocity
               else -- If the robots back-facing sensors have detected an object, do the following
                  if (debugMode == true) then -- If debug mode is active, do the following
                     printf("Speed [" .. string.format("%.2f", robotSpeed) .. " m/s]    Heading [" ..
string.format("%.2f", robotHeading) .. " DEG]    "
                        .. "Position [" .. string.format("%.2f", robotPosition[1]) .. ", " .. string.format("%.2f",
robotPosition[2]) .. "]    "
                        .. "Detection Left [" .. leftString .. "]    " .. "Detection Right [" .. rightString .. "]    "
                        .. "State [Avoiding] -----> Stopped Reversing [Object Detected Behind    "
                        .. "RANSAC [" .. string.format("%.2f", ransacTargetCompletion) .. " PCT] of [" ..
ransacTarget .. "]") -- Output the robot has stopped reversing due to an object behind
                  end -- End of the conditional statement

                  robotIsTurning = true -- Set the robot to enter the 'turning' phase
                  robotIsReversing = false -- Set the robot to exit the 'reversing' phase
               end -- End of the conditional statement
            else -- If the robots front-facing sensors no longer detect an object, do the following
               if (debugMode == true) then -- If debug mode is active, do the following
                  printf("Speed [" .. string.format("%.2f", robotSpeed) .. " m/s]    Heading [" ..
string.format("%.2f", robotHeading) .. " DEG]    "
                     .. "Position [" .. string.format("%.2f", robotPosition[1]) .. ", " .. string.format("%.2f",
robotPosition[2]) .. "]    "
                     .. "Detection Left [" .. leftString .. "]    " .. "Detection Right [" .. rightString .. "]    "
                     .. "State [Avoiding] -----> Finished Reversing    "
                     .. "RANSAC [" .. string.format("%.2f", ransacTargetCompletion) .. " PCT] of [" ..
ransacTarget .. "]") -- Output the robot has finished reversing
               end -- End of conditional statement

               robotIsTurning = true -- Set the robot to enter the 'turning' phase
               robotIsReversing = false -- Set the robot to exit the 'reversing' phase
            end -- End of the conditional statement
         end -- End of the conditional statement
      end -- End of the conditional statement
   end -- End of the conditional statement
   end -----[ REVERSING ]-----

   do -----[ TURNING ]-----
   if (robotIsReversing == false) then -- If the robot has not entered the 'reversing' phase, do the
following
      if (robotIsTurning == false) then -- If the robot has not entered the 'turning' phase, do the
following
         robotTurningRight = false -- Reset the robots 'turning left' phase
         robotTurningLeft = false -- Reset the robots 'turning right' phase
         robotRotationSet = false -- Unset the robots rotationl direction when turning
         rotationDirection = 0 -- Reset the robots rotation direction
         reverseTurnTimer = 0 -- Reset the robots turning timer
      else -- If the robot has entered the 'turning' phase, do the following

         sensorsDetected = 0 -- Reset the number of sensors that have detected an object
```

```
    for i = 1, 16, 1 do -- For all of the robots sensors, do the following
      if (detectedDistance[i] > 0) then -- If the currently iterated sensors detected distance is larger then '0m' (object has been detected), do the following
        sensorsDetected = sensorsDetected + 1 -- Increment the number of sensors that have detected an object
      end -- End of conditional statement
    end -- End of the iterative statement

    if (sensorsDetected >= 6) then -- If the number of sensors that has detected an object is equal to or larger than '8', do the following
      robotIsStuck = true -- Set the robot to enter the 'stuck' phase
      robotIsTurning = true -- Set the robot to enter the 'turning' phase
    else -- If the number of sensors that has detected an object is less than '8', do the following
      robotIsStuck = false -- The robot will
    end -- End of the conditional statement

    if (robotRotationSet == false) then -- If the robots rotation direction has not been set, do the following

      accumulatedFrontLeftSensorDistance = 0 -- Reset the accumulated distance of the robots front-left sensors
      accumulatedFrontRightSensorDistance = 0 -- Reset the accumulated distance of the robots front-right sensors

      accumulatedBackLeftSensorDistance = 0 -- Reset the accumulated distance of the robots back-left sensors
      accumulatedBackRightSensorDistance = 0 -- Reset the accumulated distance of the robots back-right sensors

      for i = 1, 4, 1 do -- For all of the robots front-left sensors, do the following
        accumulatedFrontLeftSensorDistance = accumulatedFrontLeftSensorDistance + detectedDistance[i] -- Add and equal the distance detected by the currently iterated front-left sensor
      end -- End of the iterative statement

      for i = 13, 16, 1 do -- For all of the robots back-left sensors, do the following
        accumulatedBackLeftSensorDistance = accumulatedBackLeftSensorDistance + detectedDistance[i] -- Add and equal the distance detected by the currently iterated back-left sensor
      end -- End of the iterative statement

      for i = 5, 8, 1 do -- For all of the robots front-right sensors, do the following
        accumulatedFrontRightSensorDistance = accumulatedFrontRightSensorDistance + detectedDistance[i] -- Add and equal the distance detected by the currently iterated front-right sensor
      end -- End of the iterative statement

      for i = 9, 12, 1 do -- For all of the robots back-right sensors, do the following
        accumulatedBackRightSensorDistance = accumulatedBackRightSensorDistance + detectedDistance[i] -- Add and equal the distance detected by the currently iterated back-right sensor
      end -- End of the iterative statement
```

```
        if (accumulatedFrontLeftSensorDistance > accumulatedFrontRightSensorDistance) then -- If the robots front-left sensors detect objects further away than the robots front-right sensors, do the following
          robotTurningRight = true -- Set the robot to enter the 'turning right' phase
        elseif (accumulatedFrontRightSensorDistance > accumulatedFrontLeftSensorDistance) then -- If the robots front-right sensors detect objects further away than the robots front-left sensors, do the following
          robotTurningLeft = true -- Set the robot to enter the 'turning left' phase
        elseif (accumulatedBackRightSensorDistance > accumulatedBackLeftSensorDistance) then -- If the robots back-right sensors detect objects further away than the robots back-left sensors, do the following
          robotTurningRight = true -- Set the robot to enter the 'turning right' phase
        elseif (accumulatedBackLeftSensorDistance > accumulatedBackRightSensorDistance) then -- If the robots back-left sensors detect objects further away than the robots back-right sensors, do the following
          robotTurningLeft = true -- Set the robot to enter the 'turning left' phase
        else -- If the sensor distances are equal (randomise), do the following
          if (rotationDirection == 1) then -- If the rotation direction is equal to '1', do the following
            robotTurningRight = true -- Set the robot to enter the 'turning right' phase
          else -- If the rotation direction is equal to '2', do the following
            robotTurningLeft = true -- Set the robot to enter the 'turning left' phase
          end -- End of the conditional statement
        end -- End of the conditional statement
        robotRotationSet = true -- The robots rotation direction has been set
      end -- End of the conditional statement

      do -----[ TURNING DIRECTION ]-----
      if (robotRotationSet == true) then -- If the robots rotation direction has been set (above), do the following
        if (robotIsStuck == true) then -- If the robot has entered the 'stuck' phase, do the following
          if (robotTurningLeft == true) then -- If the robot is turning left, do the following
            if (debugMode == true) then -- If debug mode is active, do the following
              printf("Speed [" .. string.format("%.2f", robotSpeed) .. " m/s]    Heading [" .. string.format("%.2f", robotHeading) .. " DEG]    "
                .. "Position [" .. string.format("%.2f", robotPosition[1]) .. ", " .. string.format("%.2f", robotPosition[2]) .. "]    "
                .. "Detection Left [" .. leftString .. "]    " .. "Detection Right [" .. rightString .. "]    "
                .. "State [Avoiding] -----> Turning Left [Robot Stuck]    "
                .. "RANSAC [" .. string.format("%.2f", ransacTargetCompletion) .. " PCT] of [" .. ransacTarget .. "]") -- Output the robot is turning left, robot is stuck
            end -- End of the conditional statement

            if (detectedDistance[4] + detectedDistance[5] ~= 0) then -- If an object is detected in front of robots facing direction, do the following
              leftWheelVelocity = -(defaultVelocity) -- Set the robots left wheel motors velocity to the default velocity negated
              rightWheelVelocity = defaultVelocity -- Set the robots right wheel motors velocity to the default velocity

              sim.setJointTargetVelocity(leftWheelMotor, leftWheelVelocity) -- Set the robots left wheels motor velocity to the calculated velocity
```

```
                            sim.setJointTargetVelocity(rightWheelMotor, rightWheelVelocity) -- Set the robots right wheels motor velocity to the calculated velocity
                        else -- If an object is no longer detected in front of robots facing direction, do the following
                            robotIsStuck = false -- Set the robot to exit the 'stuck' phase
                            robotIsTurning = false -- Set the robot to exit the 'turning' phase
                        end -- End of conditional statement
                    else -- If the robot is turning right, do the following
                        if (debugMode == true) then -- If debug mode is active, do the following
                            printf("Speed [" .. string.format("%.2f", robotSpeed) .. " m/s]    Heading [" .. string.format("%.2f", robotHeading) .. " DEG]    "
                                .. "Position [" .. string.format("%.2f", robotPosition[1]) .. ", " .. string.format("%.2f", robotPosition[2]) .. "]    "
                                .. "Detection Left [" .. leftString .. "]    " .. "Detection Right [" .. rightString .. "]    "
                                .. "State [Avoiding] -----> Turning Right [Robot Stuck]    "
                                .. "RANSAC [" .. string.format("%.2f", ransacTargetCompletion) .. " PCT] of [" .. ransacTarget .. "]") -- Output the robot is turning right, robot is stuck
                        end -- End of the conditional statement

                        if (detectedDistance[4] + detectedDistance[5] ~= 0) then -- If an object is detected in front of robots facing direction, do the following
                            leftWheelVelocity = defaultVelocity -- Set the robots left wheel motors velocity to the default velocity
                            rightWheelVelocity = -(defaultVelocity) -- Set the robots right wheel motors velocity to the default velocity negated

                            sim.setJointTargetVelocity(leftWheelMotor, leftWheelVelocity) -- Set the robots left wheels motor velocity to the calculated velocity
                            sim.setJointTargetVelocity(rightWheelMotor, rightWheelVelocity) -- Set the robots right wheels motor velocity to the calculated velocity
                        else -- If an object is no longer detected in front of robots facing direction, do the following
                            robotIsStuck = false -- Set the robot to exit the 'stuck' phase
                            robotIsTurning = false -- Set the robot to exit the 'turning' phase
                        end -- End of the conditional statement
                    end -- End of the conditional statement
                else -- If the robot has not entered the 'stuck' phase, do the following
                    if (robotTurningLeft == true) then -- If the robot is turning left, do the following
                        if (debugMode == true) then -- If debug mode is active, do the following
                            printf("Speed [" .. string.format("%.2f", robotSpeed) .. " m/s]    Heading [" .. string.format("%.2f", robotHeading) .. " DEG]    "
                                .. "Position [" .. string.format("%.2f", robotPosition[1]) .. ", " .. string.format("%.2f", robotPosition[2]) .. "]    "
                                .. "Detection Left [" .. leftString .. "]    " .. "Detection Right [" .. rightString .. "]    "
                                .. "State [Avoiding] -----> Turning Left [After Reversing]    "
                                .. "RANSAC [" .. string.format("%.2f", ransacTargetCompletion) .. " PCT] of [" .. ransacTarget .. "]") -- Output the robot is turning left, after reversing
                        end -- End of the conditional statement

                        leftWheelVelocity = -(defaultVelocity) -- Set the robots left wheel motors velocity to the default velocity negated
```

```
            rightWheelVelocity = defaultVelocity -- Set the robots right wheel motors velocity to the default velocity

            sim.setJointTargetVelocity(leftWheelMotor, leftWheelVelocity) -- Set the robots left wheels motor velocity to the calculated velocity
            sim.setJointTargetVelocity(rightWheelMotor, rightWheelVelocity) -- Set the robots right wheels motor velocity to the calculated velocity

            reverseTurnTimer = reverseTurnTimer - sim.getSimulationTimeStep() -- Subtract and equal the time passed since the last frane was made from the turn timer value

            if (reverseTurnTimer < 0) then -- If the time has depleted, do the following
              robotIsTurning = false -- Set the robot to exit the 'turning' phase
            end -- End of the conditional statement
          elseif (robotTurningRight == true) then -- If the robot is turning right, do the following
            if (debugMode == true) then -- If debug mode is active, do the following
              printf("Speed [" .. string.format("%.2f", robotSpeed) .. " m/s]    Heading [" .. string.format("%.2f", robotHeading) .. " DEG]    "
                .. "Position [" .. string.format("%.2f", robotPosition[1]) .. ", " .. string.format("%.2f", robotPosition[2]) .. "]    "
                .. "Detection Left [" .. leftString .. "]    " .. "Detection Right [" .. rightString .. "]    "
                .. "State [Avoiding] -----> Turning Right [After Reversing]    "
                .. "RANSAC [" .. string.format("%.2f", ransacTargetCompletion) .. " PCT] of [" .. ransacTarget .. "]") -- Output the robot turning right, after reversing
            end -- End of the conditional statement

            leftWheelVelocity = defaultVelocity -- Set the robots left wheel motors velocity to the default velocity
            rightWheelVelocity = -(defaultVelocity) -- Set the robots right wheel motors velocity to the default velocity negated

            sim.setJointTargetVelocity(leftWheelMotor, leftWheelVelocity) -- Set the robots left wheels motor velocity to the calculated velocity
            sim.setJointTargetVelocity(rightWheelMotor, rightWheelVelocity) -- Set the robots right wheels motor velocity to the calculated velocity

            reverseTurnTimer = reverseTurnTimer - sim.getSimulationTimeStep() -- Subtract and equal the time passed since the last frane was made from the turn timer value

            if (reverseTurnTimer < 0) then -- If the time has depleted, do the following
              robotIsTurning = false -- Set the robot to exit the 'turning' phase
            end -- End of the conditional statement
          else -- If no rotation direction is assigned, do the following (debugging)
            if (debugMode == true) then -- If debug mode is active, do the following
              print("Speed [" .. string.format("%.2f", robotSpeed) .. " m/s]    Heading [" .. string.format("%.2f", robotHeading) .. " DEG]    "
                .. "Position [" .. string.format("%.2f", robotPosition[1]) .. ", " .. string.format("%.2f", robotPosition[2]) .. "]    "
                .. "Detection Left [" .. leftString .. "]    " .. "Detection Right [" .. rightString .. "]    "
                .. "Robot has no rotation direction set!    "
```

```
                       .. "RANSAC [" .. string.format("%.2f", ransacTargetCompletion) .. " PCT] of [" ..
ransacTarget .. "]") -- Output the robot has no rotation direction set
                  end -- End of the conditional statement
               end -- End of the conditional statement
            end -- End of the conditional statement
         end -- End of the conditional statement
         end -----[ TURNING DIRECTION ]-----

      end -- End of the conditional statement
   end -- End of the conditional statement
   end -----[ TURNING ]-----

   do -----[ EXIT ]-----
   if (robotIsTurning == false and robotIsReversing == false and robotIsStuck == false) then -- If the
robot has not entered the 'turning', 'reversing' and 'stuck' phase, do the following

      robotIsAvoiding = false -- Set the robot to exit the 'avoiding' phase

   end -- End of the conditional statement
   end -----[ EXIT ]-----

end -- End of the function declaration




function robotWandering() -- Robot wandering strategy

   if(robotWanderingReset == true) then -- If the robots 'wandering' phase requires to be reset
(other phase interruption), do the following

      do -----[ RESET ALL WANDERING VARIABLES ]-----
      wanderingTurnAngle = 0 -- Reset the robots wandering turn angle

      wanderingForwardDistance = 0 -- Reset the robots wandering forward distance

      robotPosition = { 0, 0, 0 } -- Reset the robots position incrementer

      currentRobotPosition = { 0, 0, 0 } -- Robots current position table/ array
      previousRobotPosition = { 0, 0, 0 } -- Robots previous position table/ array
      accumulatedForwardDistance = 0 -- Accumulated distance the robot has moved forwards

      wanderingForwardDistanceSet = false -- Reset the robots wandering forward distance

      robotRotation = 0 -- Reset the robots rotation incrementer

      currentRobotRotation = { 0, 0, 0 } -- Reset the robots current rotation table/ array
      accumulatedRotationAngle = 0 -- Reset the accumulated angle that the robot rotates towards

      currentRobotHeading = 0 -- Reset the robots current heading (facing direction)
      previousRobotHeading = 0 -- Reset the robots previous heading (facing direction)
```

```
    robotIsRotating = false -- The robot has exit the 'rotating' phase
    end -----[ RESET ALL WANDERING VARIABLES ]-----

    robotWanderingReset = false -- The robots 'wandering' phase configuration has been reset

  else -- If the robots 'wandering' phase does not require to be reset, do the following
    if (robotIsRotating == false) then -- If the robot has not entered the 'rotating' phase, do the
following

      do -----[ RESET ROTATING VARIABLES ]-----
      robotRotation = 0 -- Reset the robots rotation incrementer

      robotRotationSet = false -- Reset the robots rotation direction

      robotTurningLeft = false -- Reset the robots 'turning left' phase
      robotTurningRight = false -- Reset the robots 'turning right' phase

      wanderingTurnAngle = 0 -- Reset the robots wandering turning angle
      end -----[ RESET ROTATING VARIABLES ]-----

      if (wanderingForwardDistanceSet == false) then -- If the wandering forward distance has not
been set, do the following
        wanderingForwardDistance = math.random(1, 5) -- Generate a distance for the robot to
traverse forwards for

        do -----[ RESET POSITION VARIABLES ]-----
        accumulatedForwardDistance = 0 -- Reset the robots accumulated distance travelled

        Position = { 0, 0, 0 } -- Reset the robots position incrementer

        currentRobotPosition = { 0, 0, 0 } -- Reset the robots current position table/ array

        previousRobotPosition = { 0, 0, 0 } -- Reset the robots previous position table/ array

        accumulatedForwardDistance = 0 -- Reset the accumulated distance that the robot moves
forwards
        end -----[ RESET POSITION VARIABLES ]-----

        wanderingForwardDistanceSet = true -- The robots wandering forward distance has been
set
      end -- End of the conditional statement

      if (previousRobotPosition[1] > 0 or previousRobotPosition[2] > 0 or previousRobotPosition[3]
> 0 ) then -- If the robots previous position is a position (not equal to '0'), do the following

        for i = 1, 2, 1 do -- For the number of elements in the table/ array, do the following
          Position[i] = currentRobotPosition[i] - previousRobotPosition[i] -- Set the robots position
to be the difference between the robots current position and the robots previous position
        end -- End of the iterative statement
```

```
        accumulatedForwardDistance = accumulatedForwardDistance + math.sqrt((Position[1] * Position[1]) + (Position[2] * Position[2])) -- Set the robots accumulated distance moved to the magnitude of the difference between vectors (previous and current positions)

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

          if (accumulatedForwardDistance >= (wanderingForwardDistance / 10)) then -- If the robots accumualted distance travelled is equal to or larger than the generated distance to travel, do the following

            wanderingForwardDistanceSet = false -- The robots wandering forward distance has not been set (reset)

            robotIsRotating = true -- Set the robot to enter the 'rotating' phase
          end -- End of the conditional statement
        end -- End of the conditional statement

      end -- End of the conditional statement

      previousRobotPosition = currentRobotPosition -- Set the robots previous position to the current position (end of frame)

      if (debugMode == true) then -- If debug mode is active, do the following
        if (allAreasExplored == true) then -- If robot has explored all of the areas in the environment, do the following
          printf("Speed [" .. string.format("%.2f", robotSpeed) .. " m/s]    Heading [" .. string.format("%.2f", robotHeading) .. " DEG]    "
            .. "Position [" .. string.format("%.2f", robotPosition[1]) .. ", " .. string.format("%.2f", robotPosition[2]) .. "]    "
            .. "Detection Left [" .. leftString .. "]    " .. "Detection Right [" .. rightString .. "]    "
            .. "State [Wandering] -----> Moving Forward    Distance Travelled [" .. string.format("%.2f", accumulatedForwardDistance)
            .. "]    Distance Travelling To [" ..  (wanderingForwardDistance / 10) .. "]    "
            .. "RANSAC [" .. string.format("%.2f", ransacTargetCompletion) .. " PCT] of [" .. ransacTarget .. "]") -- Output the robot entered the 'moving forward' phase
        else -- If robot has not explored all of the areas in the environment, do the following
          printf("Speed [" .. string.format("%.2f", robotSpeed) .. " m/s]    Heading [" .. string.format("%.2f", robotHeading) .. " DEG]    "
            .. "Position [" .. string.format("%.2f", robotPosition[1]) .. ", " .. string.format("%.2f", robotPosition[2]) .. "]    "
            .. "Detection Left [" .. leftString .. "]    " .. "Detection Right [" .. rightString .. "]    "
            .. "State [Wandering] -----> Exploring [" .. exploringAreaOutput .. "]     "
            .. "Distance Travelled [" .. string.format("%.2f", accumulatedForwardDistance)
            .. "]    Distance Travelling To [" ..  (wanderingForwardDistance / 10) .. "]    "
            .. "RANSAC [" .. string.format("%.2f", ransacTargetCompletion) .. " PCT] of [" .. ransacTarget .. "]") -- Output the robot entered the 'moving forward' phase
        end -- End of the conditional statement
      end -- End of the conditional statement
```

```
      sim.setJointTargetVelocity(leftWheelMotor, defaultVelocity) -- Set the robots left wheels
motor velocity to the defualt velocity
      sim.setJointTargetVelocity(rightWheelMotor, defaultVelocity) -- Set the robots right wheels
motor velocity to the default velocity

    else -- If the robot has entered the 'rotating' phase, do the following
      if (allAreasExplored == true) then -- If robot has explored all of the areas in the environment,
do the following
        if (robotRotationSet == false) then -- If the robots rotation direction has not been set, do
the following

          accumulatedRotationAngle = 0 -- Reset the robots accumulated angle rotated

          rotationDirection = math.random(1, 100) -- Generate a rotation direction for turning
(larger range presents more randomness)

          wanderingTurnAngle = math.random(30, 90) -- Generate an angle for the robot to rotate
towards

          if (rotationDirection % 2 == 0) then -- If the rotation direction is an even number (no
remainder), do the following
            robotTurningRight = true -- Set the robot to enter the 'turning right' phase
          else -- If the rotation direction is an odd number (has a remainder), do the following
            robotTurningLeft = true -- Set the robot to enter the 'turning left' phase
          end -- End of the conditional statement

          robotRotationSet = true -- Robots rotation direction has been set

        else -- If the robots rotation direction has been set, do the following
          robotRotation = currentRobotHeading - previousRobotHeading -- Set the robots rotation
to the difference between the robots current heading and the robots previous heading

          if (robotRotation < 0) then -- If the robots rotation is negative, do the following
            accumulatedRotationAngle = accumulatedRotationAngle + -(robotRotation) --
Accumulate the changes in rotation (made positive)
          else -- If the robots rotation is positive, do the following
            accumulatedRotationAngle = accumulatedRotationAngle + robotRotation --
Accumulate the changes in rotation
          end -- End of the conditional statement

          if (robotTurningLeft == true) then -- If the robot has entered the 'turning left' phase, do
the following
            if (debugMode == true) then -- If debug mode is active, do the following
              printf("Speed [" .. string.format("%.2f", robotSpeed) .. " m/s]    Heading [" ..
string.format("%.2f", robotHeading) .. " DEG]    "
                .. "Position [" .. string.format("%.2f", robotPosition[1]) .. ", " .. string.format("%.2f",
robotPosition[2]) .. "]    "
                .. "Detection Left [" .. leftString .. "]    " .. "Detection Right [" .. rightString .. "]    "
                .. "State [Wandering] -----> Turning Left    Angle Rotated [" .. string.format("%.2f",
accumulatedRotationAngle)
                .. "]    Angle Rotating To [" ..  wanderingTurnAngle .. "]    "
```

```
                .. "RANSAC [" .. string.format("%.2f", ransacTargetCompletion) .. " PCT] of [" ..
ransacTarget .. "]") -- Output the robot entered the 'turning left' phase
            end -- End of the conditional statement

            sim.setJointTargetVelocity(leftWheelMotor, defaultVelocity / 2) -- Set the robots left
wheels motor velocity to '2' times less than the default velocity
            sim.setJointTargetVelocity(rightWheelMotor, defaultVelocity * 1.5) -- Set the robots
right wheels motor velocity to '1.5' times more than the default velocity

            if (accumulatedRotationAngle >= wanderingTurnAngle) then -- If the robots
accumualted angle rotated is equal to or larger than the generated angle, do the following
              robotIsRotating = false -- Set the robot to exit the 'rotating' phase
            end -- End of the conditional statement

          elseif (robotTurningRight == true) then -- if the robot is entering the 'turning right' phase,
do the following
            if (debugMode == true) then -- If debug mode is active, do the following
              printf("Speed [" .. string.format("%.2f", robotSpeed) .. " m/s]    Heading [" ..
string.format("%.2f", robotHeading) .. " DEG]    "
                .. "Position [" .. string.format("%.2f", robotPosition[1]) .. ", " .. string.format("%.2f",
robotPosition[2]) .. "]    "
                .. "Detection Left [" .. leftString .. "]    " .. "Detection Right [" .. rightString .. "]    "
                .. "State [Wandering] -----> Turning Right    Angle Rotated [" .. string.format("%.2f",
accumulatedRotationAngle)
                .. "]    Angle Rotating To [" ..  wanderingTurnAngle .. "]    "
                .. "RANSAC [" .. string.format("%.2f", ransacTargetCompletion) .. " PCT] of [" ..
ransacTarget .. "]") -- Output the robot entered the 'turning right' phase
            end -- End of the conditional statement

            sim.setJointTargetVelocity(leftWheelMotor, defaultVelocity * 1.5) -- Set the robots left
wheels motor velocity to '1.5' times more than the default velocity
            sim.setJointTargetVelocity(rightWheelMotor, defaultVelocity / 2) -- Set the robots right
wheels motor velocity to '2' times less than the default velocity

            if (accumulatedRotationAngle >= wanderingTurnAngle) then -- If the robots
accumualted angle rotated is equal to or larger than the generated angle, do the following
              robotIsRotating = false -- Set the robot to exit the 'rotating' phase
            end -- End of the conditional statement
          end -- End of the conditional statement

        end -- End of the conditional statement
      else -- If robot has not explored all of the areas in the environment, do the following
        if (robotRotationSet == false) then -- If the robots rotation direction has not been set, do
the following

          if (robotExploredAreaSelected == true) then -- If the robot has explored the target area
assigned, do the following
            robotAreaToExploreSelected = false -- An unexplored area has not been assigned to the
robot to explore
```

```
robotExploringArea = math.random(1, 9) -- Generate a number, representing the area for the robot to explore

while (robotAreaToExploreSelected == false) do -- While an area has not been assigned to the robot to explore, do the following
   if (robotExploredArea[robotExploringArea] == true) then -- If the robot has explored the randomly selected area, do the following
      robotExploringArea = math.random(1, 9) -- Select another area for the robot to explore
   else -- If the area randomly selected has not been explored, do the following
      robotAreaToExploreSelected = true -- An unexplored area has been assigned to the robot to explore
   end -- End of the conditional statement
end -- End of the conditional statement

for i = 1, 9, 1 do -- For all of the target areas specified, do the following
   if (i ~= robotExploringArea) then -- If the currently iterated area is not the area that the robot is set to explore, do the following
      robotAreaExploring[i] = false -- The robot will not explore any other area
   else -- if the currently iterated area is the area that the robot is closest to, do the following
      robotAreaExploring[i] = true -- The robots has been assigned an area to explore
   end -- End of the conditional statement
end -- End of the iterative statement

if (robotExploringArea == 1) then -- If the robot has been set to explore the 'top-left' region of the environment, do the following
   exploringAreaOutput = "Top-left" -- Set the area explored output to the area the robot is exploring
elseif (robotExploringArea == 2) then -- If the robot has been set to explore the 'top-middle' region of the environment, do the following
   exploringAreaOutput = "Top-middle" -- Set the area explored output to the area the robot is exploring
elseif (robotExploringArea == 3) then -- If the robot has been set to explore the 'top-right' region of the environment, do the following
   exploringAreaOutput = "Top-right" -- Set the area explored output to the area the robot is exploring
elseif (robotExploringArea == 4) then -- If the robot has been set to explore the 'middle-left' region of the environment, do the following
   exploringAreaOutput = "Middle-left" -- Set the area explored output to the area the robot is exploring
elseif (robotExploringArea == 5) then -- If the robot has been set to explore the 'centre' region of the environment, do the following
   exploringAreaOutput = "Centre" -- Set the area explored output to the area the robot is exploring
elseif (robotExploringArea == 6) then -- If the robot has been set to explore the 'middle-right' region of the environment, do the following
   exploringAreaOutput = "Middle-right" -- Set the area explored output to the area the robot is exploring
elseif (robotExploringArea == 7) then -- If the robot has been set to explore the 'bottom-left' region of the environment, do the following
```

```
                            exploringAreaOutput = "Bottom-left" -- Set the area explored output to the area the
robot is exploring
                        elseif (robotExploringArea == 8) then -- If the robot has been set to explore the
'bottom-middle' region of the environment, do the following
                            exploringAreaOutput = "Bottom-middle" -- Set the area explored output to the area
the robot is exploring
                        elseif (robotExploringArea == 9) then -- If the robot has been set to explore the
'bottom-right' region of the environment, do the following
                            exploringAreaOutput = "Bottom-right" -- Set the area explored output to the area
the robot is exploring
                        end -- End of the conditional statement

                    end -- End of the conditional statement

                    robotTargetExploreAngle = math.deg(math.atan2(targetPositions[robotExploringArea][2]
- robotPosition[2],
                                                                  targetPositions[robotExploringArea][1] - robotPosition[1])) --
Calculate the angular difference between the robots current position and target position (excluding
robot heading)

                    if (robotTargetExploreAngle < 0) then -- If the angular difference between the robots
position (aswell as facing direction) and the target position is smaller than '0' degrees (negative), do
the following
                        robotTargetExploreAngle = robotTargetExploreAngle + 360 -- Set the angular difference
to be positive (relatively)
                    end -- End of the conditional statement

                    robotExploredAreaSelected = false -- The robot has not explored the area that it is
currently assigned to

                    if (robotCurrentHeading0to360 < robotTargetExploreAngle) then -- If the robots current
heading (translated) is smaller than the angle it will rotate to, do the following
                        if (robotTargetExploreAngle - robotCurrentHeading0to360 <
                            360 + (robotCurrentHeading0to360 - robotTargetExploreAngle)) then -- If the
difference between the angle to rotate to and the robots heading is smaller than its opposite
calculation, made positive (relatively), do the following

                            robotDifferenceBetweenAngles = robotTargetExploreAngle -
robotCurrentHeading0to360 -- Store the difference between the angles

                            if (robotDifferenceBetweenAngles < 0) then -- If the difference between the angles is
smaller than '0' degrees, do the following
                                robotDifferenceBetweenAngles = robotDifferenceBetweenAngles + 360  -- Add
'360' degrees to the difference
                            end -- End of the conditional statement

                            if (robotDifferenceBetweenAngles > 360) then -- If the difference between the angles
is larger than '360' degrees, do the following
                                robotDifferenceBetweenAngles = robotDifferenceBetweenAngles - 360 -- Subtract
'360' degrees to the difference
                            end -- End of the conditional statement
```

```
              robotTurningLeft = true -- Set the robot to turn left
            else -- If the difference between the angle to rotate to and the robots heading is larger
than its opposite calculation, made positive (relatively), do the following
              robotDifferenceBetweenAngles = 360 + (robotCurrentHeading0to360 -
robotTargetExploreAngle) -- Store the difference between the angles, made positive (relatively)

              if (robotDifferenceBetweenAngles < 0) then -- If the difference between the angles is
smaller than '0' degrees, do the following (not required but implemented in case of error)
                robotDifferenceBetweenAngles = robotDifferenceBetweenAngles + 360 -- Add
'360' degrees to the difference
              end -- End of the conditional statement

              if (robotDifferenceBetweenAngles > 360) then -- If the difference between the angles
is larger than '360' degrees, do the following
                robotDifferenceBetweenAngles = robotDifferenceBetweenAngles - 360 -- Subtract
'360' degrees to the difference
              end -- End of the conditional statement

              robotTurningRight = true -- Set the robot to turn right
            end -- End of the conditional statement
          elseif (robotCurrentHeading0to360 > robotTargetExploreAngle) then -- If the robots
current heading (translated) is larger than the angle it will rotate to, do the following
            if (360 + (robotTargetExploreAngle - robotCurrentHeading0to360) <
              robotCurrentHeading0to360 - robotTargetExploreAngle) then -- If the difference
between the angle to rotate to and the robots heading made positive (relatively) is smaller than its
opposite calculation, do the following

              robotDifferenceBetweenAngles = 360 + (robotTargetExploreAngle -
robotCurrentHeading0to360) -- Store the difference between the angles, made positive (relatively)

              if (robotDifferenceBetweenAngles < 0) then -- If the difference between the angles is
smaller than '0' degrees, do the following (not required but implemented in case of error)
                robotDifferenceBetweenAngles = robotDifferenceBetweenAngles + 360 -- Add
'360' degrees to the difference
              end -- End of the conditional statement

              if (robotDifferenceBetweenAngles > 360) then -- If the difference between the angles
is larger than '360' degrees, do the following
                robotDifferenceBetweenAngles = robotDifferenceBetweenAngles - 360 -- Subtract
'360' degrees to the difference
              end -- End of the conditional statement


              robotTurningLeft = true -- Set the robot to turn left
            else -- If the difference between the angle to rotate to and the robots heading made
positive (relatively) is larger than its opposite calculation, do the following
              robotDifferenceBetweenAngles = robotCurrentHeading0to360 -
robotTargetExploreAngle -- Store the difference between the angles
```

```
                    if (robotDifferenceBetweenAngles < 0) then -- If the difference between the angles is smaller than '0' degrees, do the following
                        robotDifferenceBetweenAngles = robotDifferenceBetweenAngles + 360 -- Add '360' degrees to the difference
                    end -- End of the conditional statement

                    if (robotDifferenceBetweenAngles > 360) then -- If the difference between the angles is larger than '360' degrees, do the following
                        robotDifferenceBetweenAngles = robotDifferenceBetweenAngles - 360 -- Subtract '360' degrees to the difference
                    end -- End of the conditional statement

                    robotTurningRight = true -- Set the robot to turn right
                end -- End of the conditional statement
            end -- End of the conditional statement

            robotRotatedToTarget = 0 -- Reset the robots rotation accumulated since the last frame was made

            rotateAccumulatedRotatedToTarget = 0 -- Reset the robots rotation accumulated towards the current area target

            robotRotationSet = true -- Robots rotation direction has been set

        else -- If the robots rotation direction has been set, do the following

            robotRotatedToTarget = currentRobotHeading - previousRobotHeading -- Store the robots accumulated rotation since the last frame was made

            if (robotRotatedToTarget < 0) then -- If the robot has rotated negatively, do the following
                rotateAccumulatedRotatedToTarget = rotateAccumulatedRotatedToTarget + -(robotRotatedToTarget) -- Accumulate the robots rotation towards to the area target (negated)
            else -- If the robot has rotated positively, do the following
                rotateAccumulatedRotatedToTarget = rotateAccumulatedRotatedToTarget + robotRotatedToTarget -- Accumulate the robots rotation towards to the area target
            end -- End of the conditional statement

            if (robotTurningLeft == true) then -- If the robot has entered the 'turning left' phase, do the following
                if (debugMode == true) then -- If debug mode is active, do the following
                    printf("Speed [" .. string.format("%.2f", robotSpeed) .. " m/s]    Heading [" .. string.format("%.2f", robotHeading) .. " DEG]    "
                        .. "Position [" .. string.format("%.2f", robotPosition[1]) .. ", " .. string.format("%.2f", robotPosition[2]) .. "]    "
                        .. "Detection Left [" .. leftString .. "]    " .. "Detection Right [" .. rightString .. "]    "
                        .. "State [Wandering] -----> Exploring [" .. exploringAreaOutput .. "]     "
                        .. "Angle Rotated [" .. string.format("%.2f", rotateAccumulatedRotatedToTarget)
                        .. "]    Angle Rotating To [" ..  string.format("%.2f", robotDifferenceBetweenAngles) .. "]    "
                        .. "RANSAC [" .. string.format("%.2f", ransacTargetCompletion) .. " PCT] of [" .. ransacTarget .. "]") -- Output the robot entered the 'turning left' phase
```

```
                        end -- End of the conditional statement

                        sim.setJointTargetVelocity(leftWheelMotor, defaultVelocity / 2) -- Set the robots left
wheels motor velocity to '2' times less than the default velocity
                        sim.setJointTargetVelocity(rightWheelMotor, defaultVelocity * 1.5) -- Set the robots
right wheels motor velocity to '1.5' times more than the default velocity

                        if (rotateAccumulatedRotatedToTarget >= robotDifferenceBetweenAngles) then -- If
the robots accumualted angle rotated is equal to or larger than the angular difference between the
robot and target area, do the following
                            robotIsRotating = false -- Set the robot to exit the 'rotating' phase
                        end -- End of the conditional statement

                    elseif (robotTurningRight == true) then -- if the robot is entering the 'turning right' phase,
do the following
                        if (debugMode == true) then -- If debug mode is active, do the following
                            printf("Speed [" .. string.format("%.2f", robotSpeed) .. " m/s]    Heading [" ..
string.format("%.2f", robotHeading) .. " DEG]    "
                                .. "Position [" .. string.format("%.2f", robotPosition[1]) .. ", " .. string.format("%.2f",
robotPosition[2]) .. "]    "
                                .. "Detection Left [" .. leftString .. "]    " .. "Detection Right [" .. rightString .. "]    "
                                .. "State [Wandering] -----> Exploring [" .. exploringAreaOutput .. "]     "
                                .. "Angle Rotated [" .. string.format("%.2f", rotateAccumulatedRotatedToTarget)
                                .. "]    Angle Rotating To [" ..  string.format("%.2f", robotDifferenceBetweenAngles)
.. "]    "
                                .. "RANSAC [" .. string.format("%.2f", ransacTargetCompletion) .. " PCT] of [" ..
ransacTarget .. "]") -- Output the robot entered the 'turning right' phase
                        end -- End of the conditional statement

                        sim.setJointTargetVelocity(leftWheelMotor, defaultVelocity * 1.5) -- Set the robots left
wheels motor velocity to '1.5' times more than the default velocity
                        sim.setJointTargetVelocity(rightWheelMotor, defaultVelocity / 2) -- Set the robots right
wheels motor velocity to '2' times less than the default velocity

                        if (rotateAccumulatedRotatedToTarget >= robotDifferenceBetweenAngles) then -- If
the robots accumualted angle rotated is equal to or larger than the angular difference between the
robot and target area, do the following
                            robotIsRotating = false -- Set the robot to exit the 'rotating' phase
                        end -- End of the conditional statement
                    end -- End of the conditional statement

                end -- End of the conditional statement

            end -- End of the conditional statement

            previousRobotHeading = currentRobotHeading -- Set the robots previous heading to the
current heading (end of frame)

        end -- End of the conditional statement
    end
```

```
end -- End of the function declaration



function robotEdgeFollowing() -- Robot edge following strategy

    robotWanderingReset = true -- The robots 'wandering' phase configuration requires to be reset (was interrupted)

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

        leftSensorResult, leftSensorDistance = sim.readProximitySensor(sonarSensors[1]) -- Store object detection and object distance (left-most sensor)
        frontLeftSensorResult, frontLeftSensorDistance = sim.readProximitySensor(sonarSensors[3]) -- Store object detection and object distance (front sensor)

        if (frontLeftSensorResult > 0) and (frontLeftSensorDistance <= noDetectionDistance) then -- If an object was detected and the detected distance was within the robots no detection distance, do the following

            robotIsEdgeFollowing = false -- Set the robot to exit the 'edge following' phase

        else -- Else if an object was not detected or within the given distance to the front of the robot, do the following

            if (leftSensorResult > 0) and (leftSensorDistance ~= setPoint) then -- If an object was detected and the detcted distance is not equal to the setpoint, do the following
                if (leftSensorResult < 0) then -- If there was an error with the left-most sensor detecting, do the following
                    leftSensorDistance = maxDistance -- Set the detected distance to the maximum distance
                end -- End of the conditional statement

                leftError = setPoint - leftSensorDistance -- Set the left error to the difference between the set point and the detected distance of the robots left-most sensor

                leftErrorSum[leftErrorCounter] = leftError -- Store the error into the array, indexed at the current error count

                leftCurrentError = leftError -- Set the current left error to the current error detected

                leftErrorCounter = leftErrorCounter + 1 -- Increment the error counter

                if (leftErrorCounter > integralThreshold) then -- If the error counter is greater than the integral threshold, do the following
                    leftErrorCounter = 1 -- Reset the error counter
                end -- End of the conditional statement

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

```
          leftLastError = leftErrorSum[table.getn(leftErrorSum)] -- Set the last left error to the last
element in the array, relative to the size of the array
        else -- If the left error counter is not equal to '1', do the following
          leftLastError = leftErrorSum[leftErrorCounter - 1] -- Set the last left error to the previously
set element in the array
        end -- End of the conditional statement

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

          accumulatedLeftError = 0 -- Reset the accumulated error sum

          for i = 1, table.getn(leftErrorSum), 1 do -- For the size of the left error sum array, do the
following
            accumulatedLeftError = accumulatedLeftError + leftErrorSum[i] -- Add and equal the
currently iterated error to the accumulated error sum
          end -- End of the iterative statement

          if (table.getn(leftErrorSum) == integralThreshold) then -- If the size of the left error sum
array is equal to the integral threshold, do the following

            accumulatedLeftRMSE = 0 -- Reset the accumulated left RMSE value

            for i = 1, integralThreshold, 1 do -- For the value of integral threshold, do the following
              accumulatedLeftRMSE = accumulatedLeftRMSE + ((leftErrorSum[i])^2) -- Add and
equal the currently iterated error, squared, to the accumulated left RMSE value
            end -- End of the iterative statement

            RMSE = math.sqrt(accumulatedLeftRMSE / integralThreshold) -- Set the RMSE value to
the square root of the accumulated RMSE divided by the integral threshold

          end -- End of the conditional statement

          -- Set the robots left wheels motor velocity to the default velocity whilst adding the
proportional, integral and derivative gains (PID)
          leftWheelVelocity = defaultVelocity +
          (proportionalGain * leftError) + -- Proportional gain
          (integralGain * (accumulatedLeftError / integralThreshold)) + -- Integral gain
          (derivativeGain * (leftLastError - leftCurrentError)) -- Derivative gain

          rightWheelVelocity = defaultVelocity -- Set the robots right wheels motor velocity to the
default velocity

          if (debugMode == true) then -- If debug mode is active, do the following
            printf("Speed [" .. string.format("%.2f", robotSpeed) .. " m/s]    Heading [" ..
string.format("%.2f", robotHeading) .. " DEG]    "
              .. "Position [" .. string.format("%.2f", robotPosition[1]) .. ", " .. string.format("%.2f",
robotPosition[2]) .. "]    "
              .. "Detection Left [" .. leftString .. "]    " .. "Detection Right [" .. rightString .. "]    "
              .. "State [Edge Following] Left -----> Turning Outwards    [RMSE " ..
string.format("%.5f", RMSE) .. "]    "
```

```
                    .. "RANSAC [" .. string.format("%.2f", ransacTargetCompletion) .. " PCT] of [" .. ransacTarget .. "]") -- Output the robot entered the 'turning outwards' phase
            end -- End of the conditional statement

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

            accumulatedLeftError = 0 -- Reset the accumulated error sum

            for i = 1, table.getn(leftErrorSum), 1 do -- For the size of the left error sum array, do the following
                accumulatedLeftError = accumulatedLeftError + leftErrorSum[i] -- Add and equal the currently iterated error to the accumulated error sum
            end -- End of the iterative statement

            if (table.getn(leftErrorSum) == integralThreshold) then -- If the size of the left error sum array is equal to the integral threshold, do the following

                accumulatedLeftRMSE = 0 -- Reset the accumulated left RMSE value

                for i = 1, integralThreshold, 1 do -- For the value of integral threshold, do the following
                    accumulatedLeftRMSE = accumulatedLeftRMSE + ((leftErrorSum[i])^2) -- Add and equal the currently iterated error, squared, to the accumulated left RMSE value
                end -- End of the iterative statement

                RMSE = math.sqrt(accumulatedLeftRMSE / integralThreshold) -- Set the RMSE value to the square root of the accumulated RMSE divided by the integral threshold

            end -- End of the conditional statement

            -- Set the robots left wheels motor velocity to the default velocity whilst adding the proportional, integral and derivative gains (PID)
            leftWheelVelocity = defaultVelocity +
            (proportionalGain * leftError) + -- Proportional gain
            (integralGain * (accumulatedLeftError / integralThreshold)) + -- Integral gain
            (derivativeGain * (leftLastError - leftCurrentError)) -- Derivative gain

            rightWheelVelocity = defaultVelocity -- Set the robots right wheels motor velocity to the default velocity

            if (debugMode == true) then -- If debug mode is active, do the following
                printf("Speed [" .. string.format("%.2f", robotSpeed) .. " m/s]    Heading [" .. string.format("%.2f", robotHeading) .. " DEG]    "
                    .. "Position [" .. string.format("%.2f", robotPosition[1]) .. ", " .. string.format("%.2f", robotPosition[2]) .. "]    "
                    .. "Detection Left [" .. leftString .. "]    " .. "Detection Right [" .. rightString .. "]    "
                    .. "State [Edge Following] Left -----> Turning Inwards    [RMSE " .. string.format("%.5f", RMSE) .. "]    "
                    .. "RANSAC [" .. string.format("%.2f", ransacTargetCompletion) .. " PCT] of [" .. ransacTarget .. "]") -- Output the robot entered the 'turning inwards' phase
            end -- End of the conditional statement
```

```
        end -- End of the conditional statement

        sim.setJointTargetVelocity(leftWheelMotor, leftWheelVelocity) -- Set the velocity value of the robots 'left' motor component
        sim.setJointTargetVelocity(rightWheelMotor, rightWheelVelocity) -- Set the velocity value of the robots 'right' motor component
      end
    end
  elseif (edgeFollowingRightDetected == true) then -- If the robot has been set to follow an edge of an object on its right side, do the following

    rightSensorResult, rightSensorDistance = sim.readProximitySensor(sonarSensors[8]) -- Store object detection and object distance (left-most sensor)
    frontRightSensorResult, frontRightSensorDistance = sim.readProximitySensor(sonarSensors[6]) -- Store object detection and object distance (front sensor)

    if (frontRightSensorResult > 0) and (frontRightSensorDistance <= noDetectionDistance) then -- If an object was detected and the detected distance was within the robots no detection distance, do the following

      robotIsEdgeFollowing = false -- Set the robot to exit the 'edge following' phase

    else -- Else if an object was not detected or within the given distance to the front of the robot, do the following

      if (rightSensorResult > 0) and (rightSensorDistance ~= setPoint) then -- If an object was detected and the detcted distance is not equal to the setpoint, do the following
        if (rightSensorResult < 0) then -- If there was an error with the left-most sensor detecting, do the following
          rightSensorDistance = maxDistance -- Set the detected distance to the maximum distance
        end -- End of the conditional statement

        rightError = setPoint - rightSensorDistance -- Set the right error to the difference between the set point and the detected distance of the robots right-most sensor

        rightErrorSum[rightErrorCounter] = rightError -- Set the error into the array, relative to the current error count

        rightCurrentError = rightError -- Set the current right error to the current error detected

        rightErrorCounter = rightErrorCounter + 1 -- Increment the error counter

        if (rightErrorCounter > integralThreshold) then -- If the error counter is greater than the integral threshold, do the following
          rightErrorCounter = 1 -- Reset the error counter
        end -- End of the conditional statement

        if (rightErrorCounter == 1) then -- If the right error counter is equal to '1', do the following
          rightLastError = rightErrorSum[table.getn(rightErrorSum)] -- Set the last right error to the last element in the array, relative to the size of the array
        else -- If the right error counter is not equal to '1', do the following
```

```
                rightLastError = rightErrorSum[rightErrorCounter - 1] -- Set the last right error to the
previously set element in the array
            end -- End of the conditional statement

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

                accumulatedRightError = 0 -- Reset the accumulated error sum

                for i = 1, table.getn(rightErrorSum), 1 do -- For the size of the right error sum array, do
the following
                    accumulatedRightError = accumulatedRightError + rightErrorSum[i] -- Add and equal
the currently iterated error to the accumulated error sum
                end -- End of the iterative statement

                if (table.getn(rightErrorSum) == integralThreshold) then -- If the size of the right error
sum array is equal to the integral threshold, do the following

                    accumulatedRightRMSE = 0 -- Reset the accumulated right RMSE value

                    for i = 1, integralThreshold, 1 do -- For the value of integral threshold, do the following
                        accumulatedRightRMSE = accumulatedRightRMSE + ((rightErrorSum[i])^2) -- Add and
equal the currently iterated error, squared, to the accumulated right RMSE value
                    end -- End of the iterative statement

                    RMSE = math.sqrt(accumulatedRightRMSE / integralThreshold) -- Set the RMSE value
to the square root of the accumulated RMSE divided by the integral threshold

                end -- End of the conditional statement

                -- Set the robots right wheels motor velocity to the default velocity whilst adding the
proportional, integral and derivative gains (PID)
                rightWheelVelocity = defaultVelocity +
                (proportionalGain * rightError) + -- Proportional gain
                (integralGain * (accumulatedRightError / integralThreshold)) + -- Integral gain
                (derivativeGain * (rightLastError - rightCurrentError)) -- Derivative gain

                leftWheelVelocity = defaultVelocity -- Set the robots left wheels motor velocity to the
default velocity

                if (debugMode == true) then -- If debug mode is active, do the following
                    printf("Speed [" .. string.format("%.2f", robotSpeed) .. " m/s]    Heading [" ..
string.format("%.2f", robotHeading) .. " DEG]    "
                        .. "Position [" .. string.format("%.2f", robotPosition[1]) .. ", " .. string.format("%.2f",
robotPosition[2]) .. "]    "
                        .. "Detection Left [" .. leftString .. "]    " .. "Detection Right [" .. rightString .. "]    "
                        .. "State [Edge Following] Right -----> Turning Outwards    [RMSE " ..
string.format("%.5f", RMSE) .. "]    "
                        .. "RANSAC [" .. string.format("%.2f", ransacTargetCompletion) .. " PCT] of [" ..
ransacTarget .. "]") -- Output the robot entered the 'turning outwards' phase
                end -- End of the conditional statement
```

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

          accumulatedRightError = 0 -- Reset the accumulated error sum

          for i = 1, table.getn(rightErrorSum), 1 do -- For the size of the right error sum array, do
the following
            accumulatedRightError = accumulatedRightError + rightErrorSum[i] -- Add and equal
the currently iterated error to the accumulated error sum
          end -- End of the iterative statement

          if (table.getn(rightErrorSum) == integralThreshold) then -- If the size of the right error
sum array is equal to the integral threshold, do the following

            accumulatedRightRMSE = 0 -- Reset the accumulated right RMSE value

            for i = 1, integralThreshold, 1 do -- For the value of integral threshold, do the following
              accumulatedRightRMSE = accumulatedRightRMSE + ((rightErrorSum[i])^2) -- Add and
equal the currently iterated error, squared, to the accumulated right RMSE value
            end -- End of the iterative statement

            RMSE = math.sqrt(accumulatedRightRMSE / integralThreshold) -- Set the RMSE value
to the square root of the accumulated RMSE divided by the integral threshold

          end -- End of the conditional statement

          -- Set the robots right wheels motor velocity to the default velocity whilst adding the
proportional, integral and derivative gains (PID)
          rightWheelVelocity = defaultVelocity +
          (proportionalGain * rightError) + -- Proportional gain
          (integralGain * (accumulatedRightError / integralThreshold)) + -- Integral gain
          (derivativeGain * (rightLastError - rightCurrentError)) -- Derivative gain

          leftWheelVelocity = defaultVelocity -- Set the robots left wheels motor velocity to the
default velocity

          if (debugMode == true) then -- If debug mode is active, do the following
            printf("Speed [" .. string.format("%.2f", robotSpeed) .. " m/s]    Heading [" ..
string.format("%.2f", robotHeading) .. " DEG]    "
              .. "Position [" .. string.format("%.2f", robotPosition[1]) .. ", " .. string.format("%.2f",
robotPosition[2]) .. "]    "
              .. "Detection Left [" .. leftString .. "]    " .. "Detection Right [" .. rightString .. "]    "
              .. "State [Edge Following] Right -----> Turning Inwards    [RMSE " ..
string.format("%.5f", RMSE) .. "]    "
              .. "RANSAC [" .. string.format("%.2f", ransacTargetCompletion) .. " PCT] of [" ..
ransacTarget .. "]") -- Output the robot entered the 'turning inwards' phase
          end -- End of the conditional statement

        end -- End of the conditional statement
```

```
            sim.setJointTargetVelocity(leftWheelMotor, leftWheelVelocity) -- Set the velocity value of the robots 'left' motor component
            sim.setJointTargetVelocity(rightWheelMotor, rightWheelVelocity) -- Set the velocity value of the robots 'right' motor component
         end -- End of the conditional statement
      end -- End of the conditional statement
  end -- End of the conditional statement

end -- End of the function declaration




function calculateMapping() -- Calculate the sonar readings in the 'X' and 'Y' dimensions, relative to the robots position

   if (mainMap == true) then -- If the robot is currently in the primary environment, do the following

      graphPositionX = { 0, 0, 0, 0, 0, 0, 0, 0 } -- Create and initialise an array for storing the plot position coordinate in the 'X' dimension, for each sensor
      graphPositionY = { 0, 0, 0, 0, 0, 0, 0, 0 } -- Create and initialise an array for storing the plot position coordinate in the 'Y' dimension, for every sensor

      scenePositionX = { 0, 0, 0, 0, 0, 0, 0, 0 } -- Create and initialise an array for storing the scene position coordinate in the 'X' dimension, for every sensor
      scenePositionY = { 0, 0, 0, 0, 0, 0, 0, 0 } -- Create and initialise an array for storing the scene position coordinate in the 'Y' dimension, for every sensor

      mapSpacePositionX = 0 -- Reset the graph plot position in the 'X' axis, relative to the maps space
      mapSpacePositionY = 0 -- Reset the graph plot position in the 'Y' axis, relative to the maps space

      pioneerPosition = sim.getObjectPosition(pioneerObject, -1) -- Store the robots current position

      for i = 1, 8, 1 do -- For all of the robots front facing sensors, do the following
         if (sonarReadings[i] ~= -1) then -- If the currently iterated sonar reading has a detected distance value, do the following

            for i = 1, 8, 1 do -- For all of the robots front facing sensors, do the following
               sonarSensorPositions[i] = sim.getObjectPosition(sonarSensors[i], -1) -- Store the position of the currently iterated sonar sensor
            end -- End of the iterative statement

            distanceFromRobot = { 0, 0, 0, 0, 0, 0, 0, 0 } -- Create and initialise an array for storing the difference in distance between the robots positions and its sensors
            difference = { 0, 0 } -- Create and initialise an array for storing the difference between the robots sonar sensor positions and the robots position

            for i = 1, 8, 1 do -- For all of the robots front facing sensors, do the following
               difference[1] = sonarSensorPositions[i][1] - pioneerPosition[1] -- Set the 'X' value of the position to be the difference between the robots position and sensor position in the 'X' axis
```

```
          difference[2] = sonarSensorPositions[i][2] - pioneerPosition[2] -- Set the 'Y' value of the
position to be the difference between the robots position and sensor position in the 'Y' axis

          distanceFromRobot[i] = math.sqrt((difference[1]^2) + (difference[2]^2)) + 0.05 -- Set the
distance to the magnitude of the position difference added with extra distance
        end -- End of the iterative statement

        sonarPositionX = math.cos(math.rad(sonarAngles[i])) * (sonarReadings[i] +
distanceFromRobot[i]) -- Calculate the detected objects 'X' coordinate relative to the robots current
position (robots radius is '0.5m')
        sonarPositionY = math.sin(math.rad(sonarAngles[i])) * (sonarReadings[i] +
distanceFromRobot[i]) -- Calculate the detected objects 'Y' coordinate relative to the robots current
position (robots radius is '0.5m')

        pioneerRotation = sim.getObjectOrientation(pioneerObject, -1) -- Store the robots current
orientation
        pioneerRotation = pioneerRotation[3] -- Set the variable to be the front facing sensor only
(heading)

        rotationX = sonarPositionX * math.cos(pioneerRotation) + sonarPositionY * -
(math.sin(pioneerRotation)) -- Rotate the 'X' coordinate to CoppeliaSim's global coordinate system
        rotationY = sonarPositionX * math.sin(pioneerRotation) + sonarPositionY *
(math.cos(pioneerRotation)) -- Rotate the 'Y' coordinate to CoppeliaSim's global coordinate system

        drawingPointX = rotationX + pioneerPosition[1] -- Translate the drawing point by the robots
'X' position coordinate
        drawingPointY = rotationY + pioneerPosition[2] -- Translate the drawing point by the robots
'Y' position coordinate

        sensorReadingToDrawPoint[i][1] = drawingPointX -- Set the 'X' value of the sonar sensors
position, used to determine if a point is drawn to the calculated 'X' position value
        sensorReadingToDrawPoint[i][2] = drawingPointY -- Set the 'Y' value of the sonar sensors
position, used to determine if a point is drawn to the calculated 'Y' position value

        graphPointX = round(rotationX + pioneerPosition[1], graphPointRoundDecimalPlaces) --
Translate the drawing point by the robots 'X' position coordinate
        graphPointY = round(rotationY + pioneerPosition[2], graphPointRoundDecimalPlaces) --
Translate the drawing point by the robots 'Y' position coordinate

        sensorReadingToDrawGraph[i][1] = graphPointX -- Set the 'X' value of the sonar sensors
position, used to determine if a point is drawn to the calculated 'X' position value
        sensorReadingToDrawGraph[i][2] = graphPointY -- Set the 'Y' value of the sonar sensors
position, used to determine if a point is drawn to the calculated 'Y' position value
      end -- End of the conditional statement
    end -- End of the iterative statement [Remove for original]

    do -----[ PLOT GRAPH POINTS ]-----
    drawGraphX = { 0, 0, 0, 0, 0, 0, 0, 0 } -- Create and initialise an array for storing the robots front
facing sensor readings, in the 'X' dimension, used for plotting graph points
    drawGraphY = { 0, 0, 0, 0, 0, 0, 0, 0 } -- Create and initialise an array for storing the robots front
facing sensor readings, in the 'Y' dimension, used for plotting graph points
```

```
    for i = 1, 8, 1 do -- For all of the robots front facing sensors, do the following
       drawGraphX[i] = sensorReadingToDrawGraph[i][1] -- Store the 'X' value of the currently iterated sensors reading, used to plot graph points
       drawGraphY[i] = sensorReadingToDrawGraph[i][2] -- Store the 'Y' value of the currently iterated sensors reading, used to plot graph points
    end -- End of the iterative statement

    for i = 1, 8, 1 do -- For all of the robots front facing sensors, do the following
       for j = i + 1, 8, 1 do -- For all of the robots front facing sensors, do the following
          if (drawGraphX[i] + drawGraphX[j] / 2 <= drawGraphX[i] + 0.05 or drawGraphX[i] + drawGraphX[j] / 2 >= drawGraphX[i] - 0.05 and -- If the difference between the points is less than '0.05', do the following
             drawGraphY[i] + drawGraphY[j] / 2 <= drawGraphX[i] + 0.05 or drawGraphY[i] + drawGraphY[j] / 2 >= drawGraphX[i] - 0.05) then

             graphPositionX[i] = drawGraphX[i] -- Store the final position in the 'X' dimension for the currently iterated sensor
             graphPositionY[i] = drawGraphY[i] -- Store the final position in the 'Y' dimension for the currently iterated sensor
          end -- End of the conditional statement
       end -- End of the iterative statement
    end

    for i = 1, 8, 1 do -- For all of the robots front facing sensors, do the following
       if (graphPositionX[i] ~= 0 or graphPositionY[i] ~= 0) then -- If the graph position is not the centre of the resizable floor (50, 50), do the following
          mapSpacePositionX = (graphPositionY[i] * 10) + 50 -- Store the 'X' value of the currently iterated graph plots position (flipped), relative to the maps size and round amount used (becomes '1' if value is '0.1')
          mapSpacePositionY = (graphPositionX[i] * 10) + 50 -- Store the 'Y' value of the currently iterated graph plots position (flipped), relative to the maps size and round amount used (becomes '1' if value is '0.1')

          --printf("X: " .. mapSpacePositionY .. "   Y: " .. mapSpacePositionX .. "  Count: " .. offlineMapCounters[mapWidth - mapSpacePositionX + 1][mapSpacePositionY]) -- Output the calculated position of the robot relative to the offline map space

          if (offlineMapCounters[mapWidth - mapSpacePositionX + 1][mapSpacePositionY] == 0) then -- If the current position of a detected object has not already been detected, do the following
             sim.setGraphUserData(sim.getObjectHandle("MappingGraph"), "PositionX" .. i, graphPositionX[i]) -- Set graph data (user defined), draw final 'X' coordinates for the currently iterated sensor reading
             sim.setGraphUserData(sim.getObjectHandle("MappingGraph"), "PositionY" .. i, graphPositionY[i]) -- Set graph data (user defined), draw final 'Y' coordinates for the currently iterated sensor reading

             do -----[ RANSAC ]-----
             --for j = 1, 2, 1 do -- For the number of iterations (duplicate points for faster RANSAC calculations), do the following
```

```
            if (allDetectedCoordinates ~= ransacTarget) then -- If the number of coorindates
detected is not equal to the target number of coordinates required for RANSAC, do the following
              allCoordinatesDetected[allCoordinatesCounter] = { } -- Create an array for storing
detected object positions

              allCoordinatesDetected[allCoordinatesCounter][1] = 0 -- Intialise the first element in
the array
              allCoordinatesDetected[allCoordinatesCounter][2] = 0 -- Initialise the second element
in the array

              allCoordinatesDetected[allCoordinatesCounter][1] = mapSpacePositionY -- Store the
translated detected coordinate value of an object in the 'X' axis
              allCoordinatesDetected[allCoordinatesCounter][2] = mapSpacePositionX -- Store the
translated detected coordinate value of an object in the 'Y' axis

              allCoordinatesCounter = allCoordinatesCounter + 1 -- Increment the number of
coordinates stored (used to index map coordinates array)
              allDetectedCoordinates = allDetectedCoordinates + 1 -- Increment the number of
coordinates stored
            end -- End of the conditional statement

            if (allDetectedCoordinates == ransacTarget) then -- If the number of coordinates
detected (subtracted by one due to starting '1' for array indexing) is equal to or more than the
RANSAC target, do the following
              printf("Ending Simulation [RANSAC TARGET MET] -----> Detected Positions [" ..
allDetectedCoordinates .. "]    Target [" .. ransacTarget .. "]") -- Output simulation end
              sim.stopSimulation() -- Stop the simulation (RANSAC target was met)
            end -- End of the conditional statement
          --end -- End of the iterative statement
          end -----[ RANSAC ]-----

          offlineMapCounters[mapWidth - mapSpacePositionX + 1][mapSpacePositionY] =
offlineMapCounters[mapWidth - mapSpacePositionX + 1][mapSpacePositionY] + 1 -- Incremenet the
counter for the position that an object was detected at
        else -- If the current position of a detected object has already been detected, do the
following
          if (previousGraphPositionX[i] ~= mapSpacePositionX and previousGraphPositionY[i] ~=
mapSpacePositionY) then -- If the previously detected object positions is the same as the currently
detected object position (not detecting object any longer), do the following
            offlineMapCounters[mapWidth - mapSpacePositionX + 1][mapSpacePositionY] =
offlineMapCounters[mapWidth - mapSpacePositionX + 1][mapSpacePositionY] + 1 -- Incremenet the
counter for the position that an object was detected at

            do -----[ RANSAC ]-----
            --for k = 1, 2, 1 do -- For the number of iterations (duplicate points for faster RANSAC
calculations), do the following
              if (allDetectedCoordinates ~= ransacTarget) then -- If the number of coorindates
detected is not equal to the target number of coordinates required for RANSAC, do the following
                allCoordinatesDetected[allCoordinatesCounter] = { } -- Create an array for storing
detected object positions
```

```
                allCoordinatesDetected[allCoordinatesCounter][1] = 0 -- Intialise the first element
in the array
                allCoordinatesDetected[allCoordinatesCounter][2] = 0 -- Initialise the second
element in the array

                allCoordinatesDetected[allCoordinatesCounter][1] = mapSpacePositionY -- Store
the translated detected coordinate value of an object in the 'X' axis
                allCoordinatesDetected[allCoordinatesCounter][2] = mapSpacePositionX -- Store
the translated detected coordinate value of an object in the 'Y' axis

                allCoordinatesCounter = allCoordinatesCounter + 1 -- Increment the number of
coordinates stored (used to index map coordinates array)
                allDetectedCoordinates = allDetectedCoordinates + 1 -- Increment the number of
coordinates stored
              end -- End of the conditional statement

              if (allDetectedCoordinates == ransacTarget) then -- If the number of coordinates
detected (subtracted by one due to starting '1' for array indexing) is equal to or more than the
RANSAC target, do the following
                printf("Ending Simulation [RANSAC TARGET MET] -----> Detected Positions [" ..
allDetectedCoordinates .. "]    Target [" .. ransacTarget .. "]") -- Output simulation end
                sim.stopSimulation() -- Stop the simulation (RANSAC target was met)
              end -- End of the conditional statement
            --end -- End of the iterative statement
            end -----[ RANSAC ]-----

          end -- End of the conditional statement

        end -- End of the conditional statement

        previousGraphPositionX[i] = mapSpacePositionX -- Store the graph plot position in the 'X'
axis, for the currently iterated sensor
        previousGraphPositionY[i] = mapSpacePositionY -- Store the graph plot position in the 'X'
axis, for the currently iterated sensor

        ransacTargetCompletion = (allDetectedCoordinates / ransacTarget) * 100 -- Calculate the
target number of coordinates used by RANSAC

      end -- End of the conditional statement
    end -- End of the iterative statement
    end -----[ PLOT GRAPH POINTS ]-----

    do -----[ DRAW SCENE POINTS ]-----
      scenePointX = { 0, 0, 0, 0, 0, 0, 0, 0 } -- Create and initialise an array for storing the robots front
facing sensor readings, in the 'X' dimension, used for drawing points in the scene
      scenePointY = { 0, 0, 0, 0, 0, 0, 0, 0 } -- Create and initialise an array for storing the robots front
facing sensor readings, in the 'Y' dimension, used for drawing points in the scene

      for i = 1, 8, 1 do -- For all of the robots front facing sensors, do the following
        scenePointX[i] = sensorReadingToDrawPoint[i][1] -- Store the 'X' value of the currently
iterated sensors reading, used to draw points in the scene
```

```
        scenePointY[i] = sensorReadingToDrawPoint[i][2] -- Store the 'Y' value of the currently
iterated sensors reading, used to draw points in the scene
      end -- End of the iterative statement

      for i = 1, 8, 1 do -- For all of the robots front facing sensors, do the following
        for j = i + 1, 8, 1 do -- For all of the robots front facing sensors, do the following
          if (scenePointX[i] + scenePointX[j] / 2 <= scenePointX[i] + 0.05 or scenePointX[i] +
scenePointX[j] / 2 >= scenePointX[i] - 0.05 and -- If the difference between the points is less than
'0.05', do the following
            scenePointY[i] + scenePointY[j] / 2 <= scenePointX[i] + 0.05 or scenePointY[i] +
scenePointY[j] / 2 >= scenePointX[i] - 0.05) then

            scenePositionX[i] = scenePointX[i] -- Store the final position in the 'X' dimension for the
currently iterated sensor
            scenePositionY[i] = scenePointY[i] -- Store the final position in the 'Y' dimension for the
currently iterated sensor
          end -- End of the conditional statement
        end -- End of the iterative statement

        sim.addDrawingObjectItem(sceneDrawingPoints, { scenePositionX[i], scenePositionY[i], 0.81
}) -- Add a drawing point per function iteration

      end -- End of the iterative statement

    end -----[ DRAW SCENE POINTS ]-----

    do -----[ DRAW ROBOT PATH POINTS ]-----

    sim.setGraphUserData(sim.getObjectHandle("MappingGraph"), "RobotPositionX",
pioneerPosition[1]) -- Set graph data (user defined), draw final 'X' coordinates for the currently
iterated sensor reading
    sim.setGraphUserData(sim.getObjectHandle("MappingGraph"), "RobotPositionY",
pioneerPosition[2]) -- Set graph data (user defined), draw final 'Y' coordinates for the currently
iterated sensor reading

    end -----[ DRAW ROBOT PATH POINTS ]-----

  end -- End of the conditional statement
end -- End of the function declaration



function round(number, decimalPlaces) -- Round a given number to given decimal places

   local roundAmount = 10^(decimalPlaces or 0) -- Set the rounding amount (decimal places passed
or '0')

   number = number * roundAmount -- Multiply the passed number by the round amount calculated

   if (number >= 0) then -- If the passed number is larger than or equal to '0' (positive), do the
following
```

```
        number = math.floor(number + 0.5) -- Round the passed number up (add)
    else -- If the number is not positive, do the following
        number = math.ceil(number - 0.5) -- Round the passed number down (subtract)
    end -- End of the conditional statement

    return number / roundAmount -- Return the number divided by the amount amount
end -- End of the function declaration
```