

Milestone 2 Report

Authors: Kian Frassek, Cameron Clark and Thiyashan Pillay

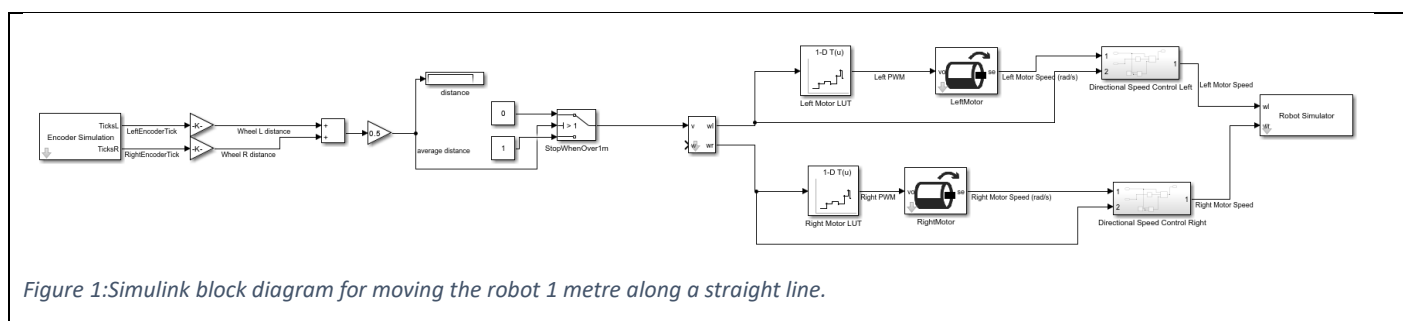
Common Blocks:

Below is a list of commonly used blocks in the different algorithms.

Encoder Simulation Block, Measures the rotation of the left and right wheels in ticks. The block outputs the tick count for the left and right motor, given as <i>LeftEncoderTick</i> and <i>RightEncoderTick</i> . If the wheel rotates clockwise, the tick counter will increment representing forward motion. If the wheel rotates counterclockwise, the tick counter will decrement representing reverse motion.
Gain Blocks, These blocks perform the following math operation to convert the number of ticks from the robot into a length in metres: $\frac{2\pi(\text{Wheel radius})}{\text{Number of ticks per rotation}}$
Line Sensor Simulation Block, This block will, based on the position of the robot and offset of the sensors, will use a simulation map to compute the line sensor values. The output of this block is an array called <i>LineSensorArray</i> .
Ultrasonic Sensor Simulation Block, This block maps the position of the robot and the sensor offsets and returns the distance to the object in its path.
To <i>wlwr</i> Block, This block converts the linear velocity of the robot given by the switch statement into left and right wheel angular velocities, <i>wl</i> and <i>wr</i> respectively. This is then fed into the left and right motor blocks.
Motor Blocks, The angular speeds, <i>wl</i> and <i>wr</i> , are fed into the motor blocks. There is one for the left motor and another for the right motor. Facilitates movement of the robot for the desired motion. In this specific simulation, it is an LTI system that will model the motor providing motion. There is one for the left and right motor. The output of these is both fed into the Robot simulator block.
Robot simulator block, This subsystem allows for the choice of different maps for the robot to follow depending on the simulation type.
Instruction for simulation: Prior to running any algorithms, first run the <i>installMRTToolbox</i> and <i>robotParameters</i> . Ensure that you set the points interactively (on the start line).
When extracting the zip folder, if ever there are any errors, just click skip. When the folder is extracted, all will work as desired.

i. Distance Control Algorithm:

This was demonstrated in the two Simulink files related to the robot moving 1 metre (with 10% error), as well as when the robot turns 90, 180, and 270 degrees (with 10% error) while remaining in the same place.



The link to the file is [here](#). The details of how all subsystems integrate with each other to get the desired motion is shown below. The table follows the logical order of how the system will work.

For Figure 1 (Robot moving 1 metre along a straight line),	
1.	Encoder Simulation Block, In this case both wheels will have equal tick counts.

	The two outputs are fed into the system through two gain blocks. The output is also measured with a display block, to see if the correct distance is being travelled by the robot.
2.	Gain Blocks, The two gain blocks are fed into a summation block and the output is halved to get the distance travelled by the robot (since both wheels are inline and move the same distance). The final output is then fed into a switch block.
3.	Switch Block, This switch block checks if the distance travelled, in metres, exceeds 1 metre. If it does not, then the robot will be allowed to move, and the output sent to the $wlwr$ block will be 1. As soon as the statement evaluates as true, then the output to the $wlwr$ block will be cut off and it will stop.
4.	To $wlwr$ Block, The input comes from the switch block only for v . No input for ω since there is only linear motion.
5.	Motor Blocks.
6.	Robot simulator block, The map used here is a simple straight line where the robot will move 1 metre along (<i>straight_line.mat</i>). Instruction for running the sim: $Start\ X\ (m) \rightarrow 0.5; Start\ Y\ (m) \rightarrow 0.1; Theta\ (deg) = 90$

i. Angle Control Algorithm

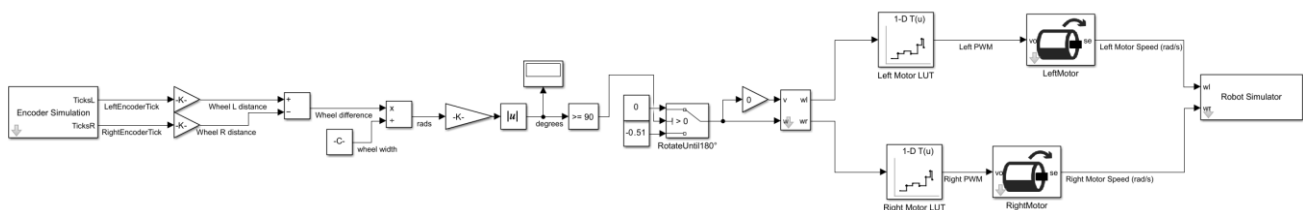


Figure 2: Simulink block diagram for moving the robot turning 90 degrees in place.

The link to the file is [here II 90°](#).

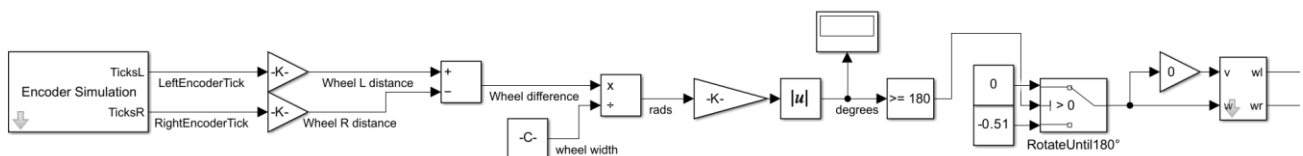


Figure 3: Simulink block diagram for moving the robot turning 180 degrees in place.

The link to the file is [here II 180°](#)

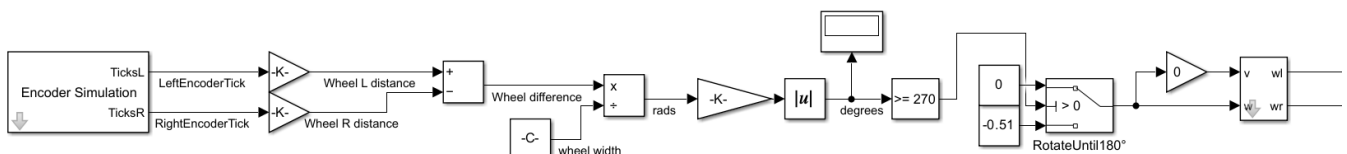


Figure 4: Simulink block diagram for moving the robot turning 270 degrees in place.

The link to the file is [here II 270°](#)

The details of how all subsystems integrate with each other to get the desired motion is shown below. The table follows the logical order of how the system will work. Please wait for the simulation to settle at a value. Initially when Frassek K, Clark C and Pillay T

rotating it may overshoot, but it settles near the desired value in the error range. For example, with a 90° rotation, it shoots to a value around 104° initially, but ends up settling at around 94° .

For Figure 2 to 4 (Robot rotating 90, 180, and 270 degrees in place),

1.	Encoder Simulation Block, In this case the two wheels rotate in opposite directions but over the same number of degrees. The ticks for each wheel will remain the same, just with different signs. The two outputs are fed into the system through two gain blocks.
2.	Gain Blocks, The two gain blocks are fed into a minus block to find the distance travelled by the outermost wheel. This is then fed into a divide block where the distance is divided by the axle length to get the radial distance travelled by the robot, in radians.
3.	Switch Block, The output from the divide block is converted to degrees and made to always be positive (abs block) before entering the switch block. This is then fed into the switch block. If the angle is less than the required angle, then the robot will rotate, but once the condition evaluates to true (angle of the robot exceeds the required angle), the robot stops dead in its tracks. The output is then fed into the $wlwr$.
4.	To $wlwr$ Block, The input comes from the switch block only for ω . The input of v is forced to zero to ensure the robot remains in place.
5.	Motor Blocks.
6.	Robot simulator block, There is no need to load a specific map here.

ii. Line Following Algorithm:

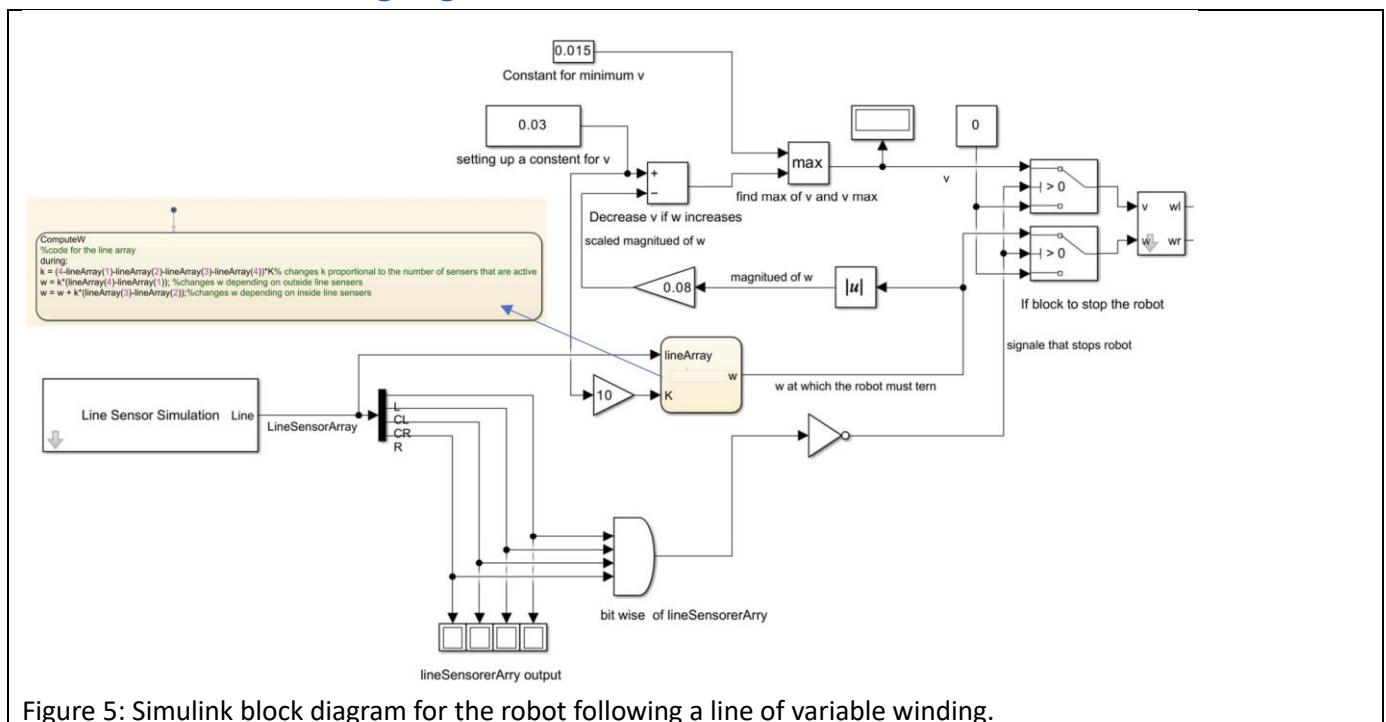


Figure 5: Simulink block diagram for the robot following a line of variable winding.

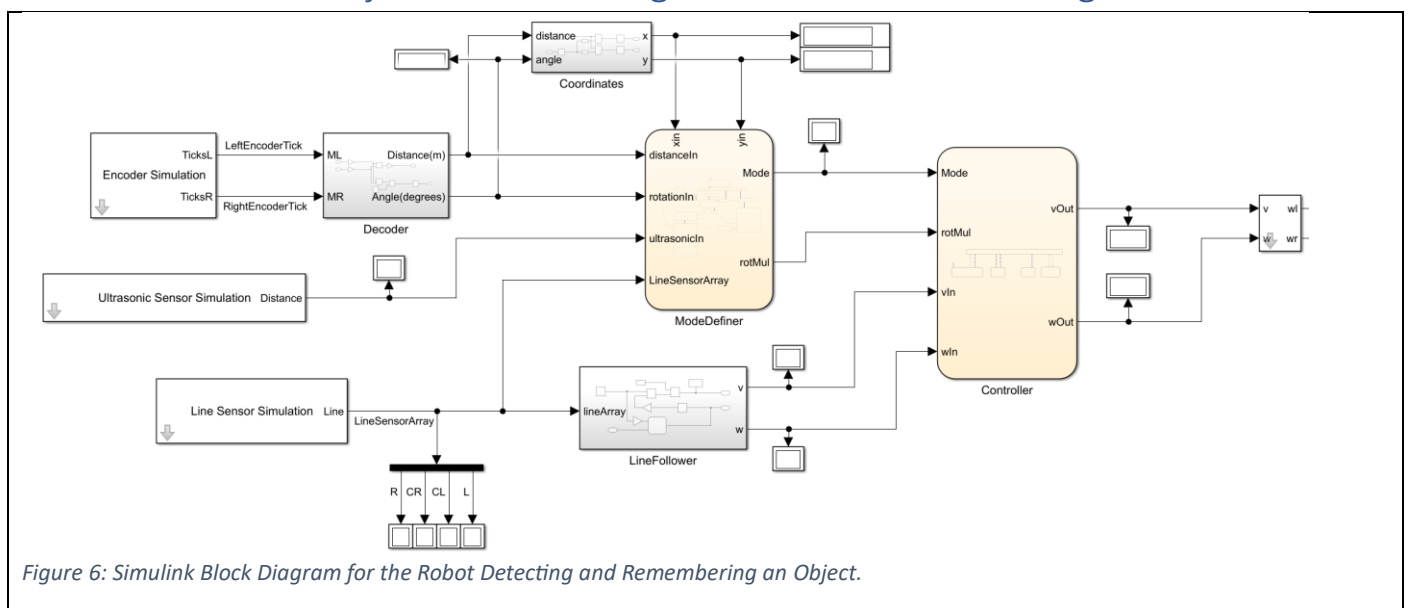
The link to the file is [here III](#). The details of how all subsystems integrate with each other to get the desired motion is shown below. The table follows the logical order of how the system will work.

For Figure 5,

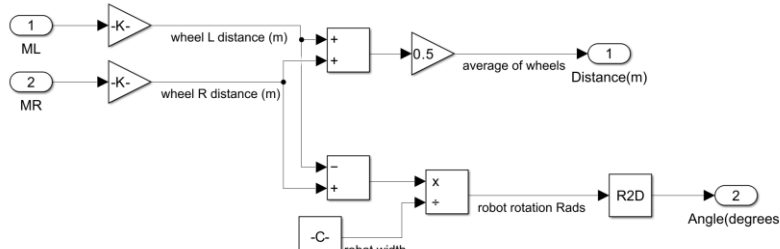
1.	Line Sensor Simulation Block, The output of this block is fed into a <i>demux</i> block as well as the <i>lineArray</i> block.
2.	<i>lineArray</i> Chart Block,

	Speed control for when the robot is on the line. There are two inputs, one is the <i>LineSensorArray</i> , and the other is the value K , which is the gain based on the speed v . The output is ω and will be fed directly into the switch block relating to ω .
3.	Switch Blocks, There are two switch blocks, one for v and another for w . The switch blocks cut the power supply off to the robot once it leaves the line using logic gates. The v will decrease based on the magnitude of ω .
4.	To <i>wlwr</i> Block, The input to this block comes from the switch blocks. The output then leads to the motor blocks, and the process follows the same route as the angle following algorithm.
5.	Motor Blocks.
6.	Robot simulator block, The map loaded here are the <i>winding_v1.mat</i> , <i>winding_v2.mat</i> , and <i>winding_v3.mat</i> . Set the start point interactively, starting on the line for optimal results.

iii. And V. Object Detection Algorithm & Localisation Algorithm:



The link to the file is [here IV and V](#). An explanation for how the different blocks interacts can be found below,

For Figure 6,	
1.	Encoder Simulation Block, The output is fed into the decoder.
2.	Decoder Block, Uses logic from the distance and angle control algorithms (see i. and ii.). Takes the input from the encoder and outputs a distance in metres and angle in degrees. The output is then fed into the coordinates block and the mode definer. 
3.	Coordinates Block, Takes in the distance and angle. Uses the discrete time difference of the distance and adds to the x and y of the current position using the <i>cos</i> and <i>sin</i> of the angle respectively.

The output is fed into the *ModeDefiner* block.

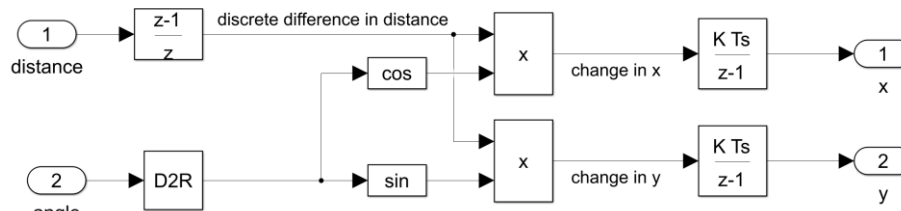


Figure 8: Showing a block diagram of the Coordinates Block from figure 6

4. Ultrasonic Sensor Simulation Block,
The output is then fed into the *ModeDefiner* block.
5. Line Sensor Simulation Block,
The output is fed into the *LineFollower* Block.
6. *LineFollower* Block,
Takes in the *LineArray* input from the Line Sensor Simulation block.
Implements a simplified algorithm described in iii. to dictate forward motion of the robot, where only the two front line sensors are used.
7. Controller Block,
This takes in 4 inputs: *Mode*, *rotMul* (Rotation Multiplier) from the *ModeDefiner* Block and, *vIn*, *ωIn* from the *LineFollower* block.
This block drives the robot based on various modes, which are as follows:
 - Mode 0: Drive using the *LineFollower*.
 - Mode 1: Drive straight forward.
 - Mode 3: Stop.
 - Mode 4: Rotate in a direction based on the Rotation Multiplier parameter.
 The outputs are *vOut* and *ωOut* are fed into the *wlwr* block.
8. *ModeDefiner* Block, (P.S. Good Luck :P)

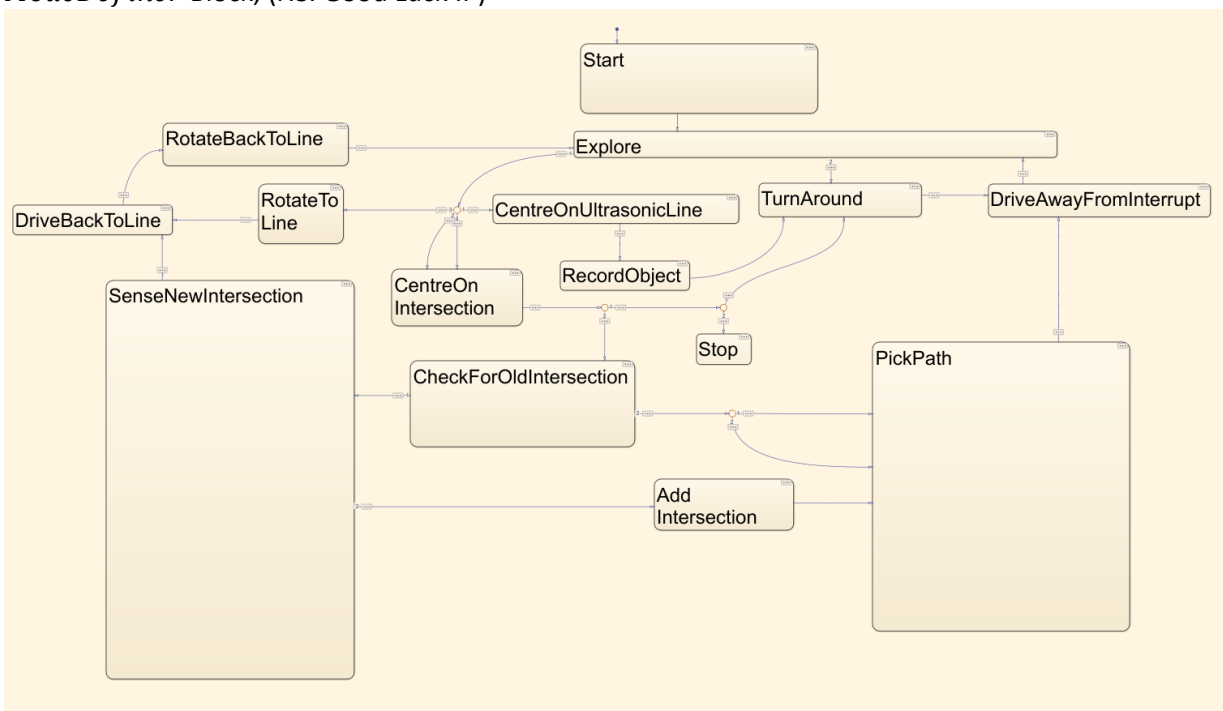


Figure 9: Showing a block diagram of the ModeDefiner Block from figure 6 (showing only the state names)

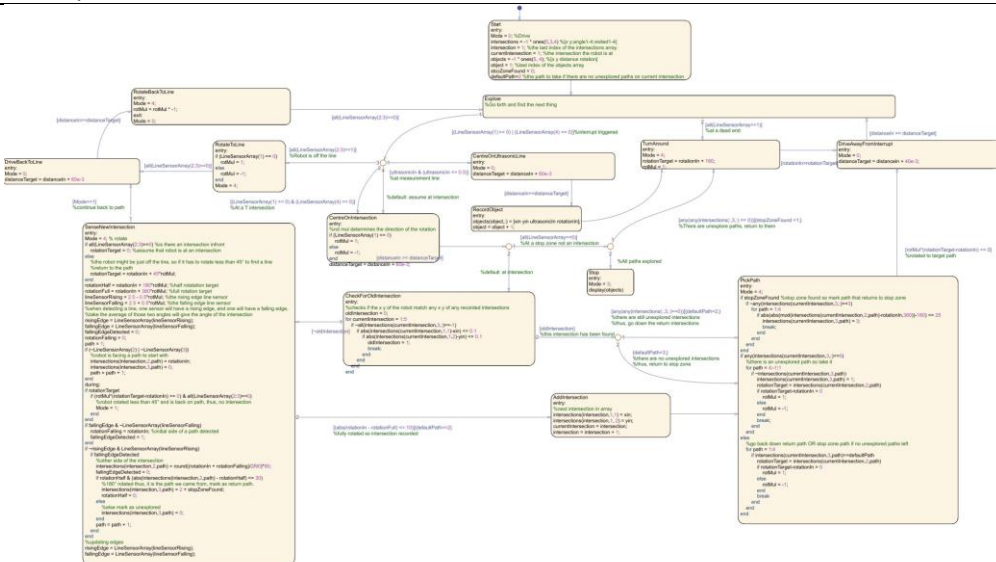


Figure 10: Showing a block diagram of the ModeDefiner Block from figure 6 (look at [pdf](#) for a higher resolution image)

This chart takes in 6 inputs. *xin*, *yin*, *distanceIn*, *rotationIn*, *ultrasonicIn*, *LineSensorArray*. Initially, the block defines two empty arrays that store the intersections and the objects the robot encounters, and the mode is also set to zero.

The robot then transitions to the Explore block, which enables the robot to simply drive (in mode zero). The robot continues in this mode until either of the two back sensors detects a line.

Thereafter, the robot must decide what has caused the interrupt. The following cases for the interrupts are:

1. Is the robot near an object?
2. Has the robot lost its path?
3. If it is none of these, then the robot moves forward by a small amount, to assess whether it is:
 - a. at the stop zone (indicated by all sensors sensing a line) or,
 - b. at an intersection.

For 1., The robot centres itself on the measuring line, records coordinates of the robot, the distance to the object, and the angle of the robot in the objects array. It then turns and continues in mode 0.

For 2., The robot will turn until it senses the path again. Then it will drive forward (in mode 1) to centre itself on the path and then turn back to face the path. It then turns and continues in mode 0.

For 3.a., The robot will look through the intersection array for any unexplored paths. If there are any, then it will turn around and use the intersection algorithm to travel to all unexplored paths. When it uses the algorithm this time, it will mark a "return to stop zone" path, to follow when all the intersections have been explored. If there aren't any unexplored paths, it will stop in mode 3 and terminate the program. The objects will then be displayed to the terminal.

For 3.b., The robot will first check if it is an intersection, it has encountered before. If it is, it will go down the next unexplored path. If there are no unexplored paths, it will take the specified return path. If it is a new intersection, the robot will do a spin, sensing all the paths. It will save the intersection positions, angle of the paths, and which path is the return path. In either case, when it chooses a path to explore, it will mark this path as explored in the intersections array.

Finally, if all the line sensors lose the line, the robot will assume it is at a dead end and turn around.

9.	To <i>wlwr</i> Block.
10.	Motor Blocks.
11.	Robot simulator block, The map that must be loaded here is the <i>Straight line with objects fixed.mat</i> .