EEE3099S

Engineering Design: Mechatronics

Milestone 4

Department of Electrical Engineering

Written by

Crystal Jaftha [JFTCRY001]

Mohammed-Bilaal Sheik Hoosen [SHKMOH025]

DUE : 24 October 2021

1. Introduction **3**

2. Hardware Description **4**
2.1 Physical Hardware Components **4**
2.2 Robot Kinematic Model and Description **5**
2.2.1 Differential Drive 5
2.2.2 Dead Reckoning 5
2.2.3 Distance Travelled and Velocity 5
2.2.4 Rotation 6
2.2.5 Gravity TT Motor Encoders Kit 6
2.2.6 Encoder Input Algorithm 6
2.3 PWM **6**
2.3.1 Explanation 6
2.3.2 Pin Configuration 7
2.4 Interface **8**

3. Motion Control **8**
3.1 Distance Control Design & Implementation 8
3.1.1 Distance Control Algorithm 8
3.1.2 Distance Control Hardware 8
3.2. Angle Control Design & Implementation 9
3.2.1 Angle Control Algorithm 9
3.2.2 Angle Control Hardware 9
3.3. Motion Control Results 9

4. Line Sensing **10**
4.1. Line Sensing Design & Implementation 10
4.1.1. Line Sensing Simulation 10
4.1.2. Line Sensing Hardware 11
4.1.3. Line Sensing Algorithm 11
4.2. Line Configuration Design & Implementation 12
4.2.1. Line Configuration Overview 12
4.2.2. Left Turn, Left T Junction & Cross Configurations 12
4.2.3. Right Turn & Right T Configurations 13
4.2.4. End of Maze & T Junction Configurations 13
4.2.5. Dead End 14
4.3. Line Sensing & Configuration Results 14

5. Maze solver and shortest pathfinder algorithm design, implementation, and results **15**
5.1 Algorithms 15
5.1.1 Maze Solver Algorithm Summary 15
5.1.2 Shortest Path Algorithm Summary 15
5.2 Algorithm and implementation 15

5.2.1 Maze Solver 15
5.2.2 Shortest Path 16
2.4 Results 17

6. Conclusion **18**
6.1 Result Overview 18
6.2 Improvements and Recommendations 18

7. Reference List **19**

1. Introduction

The aim of this project was to develop an autonomous mobile robot to participate in a maze solving operation. A self-powered differential drive robot was provided and Matlab and Simulink were used to implement both a simulation and a hardware solution. There were two phases to this project. The first phase was the learning of a simple maze, where the robot needed to follow a black track with a width of 18mm to learn the maze and find the shortest path from the beginning to the end. The robot was required to start at the push of an accessible button and indicate its mode of operation with an LED. The robot had to then stop and indicate on the LED once it had learnt the maze. The robot needed to be prompted to optimise the path and indicate with a blinking LED that it can solve the maze using the optimised path once it is ready. This leads to the second phase which was a timed event where the robot was required to travel through the same maze on the shortest path from the beginning to the end of the maze, where the end of the maze was indicated by a solid black box. The first part of the solution was to design a motion control algorithm which allowed the robot to move forward in a straight line as well as rotate about its centre. The second part of the solution was to design a line sensing algorithm and a line configuration algorithm to sense the line of the maze and 8 different configurations. And finally, the third part of the solution was to implement an algorithm that made use of the previous solutions to learn the maze and race through the shortest path.

Specifications

The following specifications were given:

- The robot must start with a button push and indicate on the LED that it is busy learning the maze.
- The robot must sense and follow a line
- The robot must implement a maze learning algorithm
- The robot must stop once maze learning completed
- The robot must indicate once it has arrived at the end with an indication (LED)
- The robot must sense the following configurations: Dead-end, End of maze, Left T junction, Right T junction, Cross, T junction, Right Turn and Left Turn
- The robot must optimise the path on a button push and indicate(LED) once it’s ready to move through the shortest path.
- The robot must race through the maze using the optimised path and indicate(LED) after completing the timed race.

2. Hardware Description

2.1 Physical Hardware Components

- 1 x Turtle: 2WD Mobile Robot Platform.
  https://wiki.dfrobot.com/2WD_Mobile_Platform_for_Arduino__SKU_ROB0005_

This serves as the housing and chassis for the robot.

- 4 x Gravity: Digital Line Tracking(Following) Sensor
  https://wiki.dfrobot.com/Line_Tracking_Sensor_for_Arduino_V4_SKU_SEN0017

The digital line tracking sensors are used to detect the line and environment for the robot with the sensors positioned as seen in the table and figure below. The sensors return a 1 for the environment and a 0 for the line that it is attempting to follow.

Sensor Number	X Position [mm]	Y Position [mm]
1	64.3	35.4
2	67.5	5
3	67.5	-5
4	64.3	- 35.4

Table 1: Sensor positions

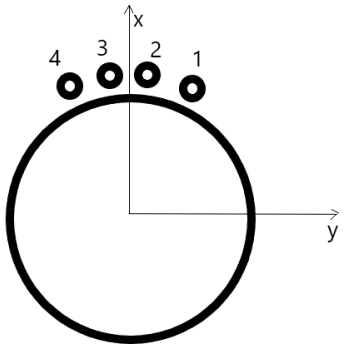

Figure 1: diagram showing position of sensors

- 2 x Gravity: TT Motor Encoders Kit.
  https://www.dfrobot.com/wiki/index.php/Wheel_Encoders_for_DFRobot_3PA_and_4WD_Rovers_(SKU:SEN0038)
  The encoders are responsible for determining the distance travelled through the use of ticks per rotation.

2.2 Robot Kinematic Model and Description

This project made use of a self-powered differential drive robot that consisted of a Romeo V2 Robot Control Board with Motor Driver, a 2WD Mobile Robot Platform, Gravity Digital Line Tracking Sensors, and Gravity TT Motor Encoders Kit. The robot makes use of these sensors and components in order to move set distances, rotate fixed degree arcs as well as differentiate between the maze course and its environment in order to facilitate successful navigation.

2.2.1 Differential Drive

A differential drive robot's movement is controlled using two wheels positioned on each side of the robot. To implement a differential drive model, the radius of the wheels and the distance between both wheels i.e. the axle length, needs to be known. In this case, the wheel radius was given to be 3.1cm and the axle length was given to be 13.6cm. If both wheels rotate in the same direction with equal angular velocities, the robot will move in the direction of wheel rotation in a straight line. If both wheels rotate with equal angular velocities in opposite directions, the robot will rotate about its centre [1]. Now, an algorithm is needed to determine how far the robot actually travels.

2.2.2 Dead Reckoning

Dead reckoning is a method used to determine the position of an object by calculating the distance travelled by that object [2]. The dead reckoning algorithm for this project was implemented using encoder sensors. The encoder sensors were used to determine the distance travelled by the robot by determining the number of wheel rotations.

2.2.3 Distance Travelled and Velocity

The Gravity TT Motor Encoders have 20 ticks per full rotation of the wheels. The number of wheel rotations is therefore equal to the number of ticks counted when the robot moves, divided by 20: $Number\ of\ Wheel\ Rotations\ = \frac{Total\ Encoder\ Ticks}{Tick\ Count\ per\ Rotation} = \frac{Total\ Encoder\ Ticks}{20}$.

When the wheel travels a full rotation, the distance travelled is equal the to circumference of the wheel which is given by: $Circumference\ = 2\ *\ pi\ *\ wheelRadius.$

The distance travelled by each wheel can therefore be calculated using the following equation: $Distance\ Travelled\ = \frac{Total\ Encoder\ Ticks}{20}\ *\ 2\ *\ pi\ *\ wheelRadius.$

The velocity of the robot itself determined by finding the average velocity of both wheels:

$Robot\ Velocity\ = \frac{v_{right} + v_{left}}{2}$

2.2.4 Rotation

When the robot rotates about its centre, each wheel travels along the circumference of a circle with a radius of half the axle length of the robot. Assuming just the right wheel is taken into perspective, the distance it travels is equivalent to the whole circumference of the circle when the motor rotates 360°. The circumference of the circle can be calculated using the equation: $Circumference = 2 * pi * (\frac{axleLength}{2}) = pi * axleLength$. For the robot to rotate 90°, the right wheel only needs to travel a distance of a quarter of the circumference. To rotate 180°, it needs to travel a distance of half the circumference and to rotate 270°, it needs to travel a distance of three quarters of the circumference.

2.2.5 Gravity TT Motor Encoders Kit

An encoder is a device that can be attached to a wheel to determine the number of times the wheel has rotated [3]. It performs this by counting the number of ticks which have passed where the resolution is determined by the number of cuts in the ring. The Gravity TT Motor Encoders Kit was used for this project and provided 20 ticks per rotation. The specifications for this kit are:

- Voltage: +5V
- Current: <20mA
- Weight: 20g
- Resolution: 20 PPR

2.2.6 Encoder Input Algorithm

Digital input pins 2 and 3 were connected to an edge detector to detect the rising and falling edges of the binary input. Each binary input was then converted into a double and fed into memory blocks which held the inputs for one iteration and generated single outputs. The distance travelled was determined using the tick account according to the distance travelled equation mentioned in section 1.3: $Distance\ Travelled = \frac{Total\ Encoder\ Ticks}{20} * 2 * pi * wheelRadius$

2.3 PWM

2.3.1 Explanation

PWM stands for Pulse Width Modulation and can be used in this scenario to control motor speed.

PWM operates like a switch which constantly cycles between on and off and is a technique used to regulate the amount of power a motor will get by varying the pulses[4].

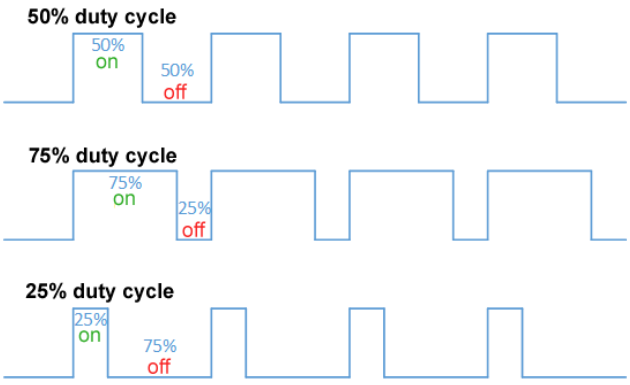

Figure 2: Duty cycle

In the figure above, if the Vmax is given to be 5V, with a 50% duty cycle the motor will receive the average voltage of 2.5V. A 75% duty cycle will yield 3.75V and a 25% duty cycle will yield 1.25V.

Since the motor of this robot is not controlled with linear or analog voltage change - this can be achieved through PWM which is intended to mimic linear control. By adjusting the "on-off" pulses and the duty cycle, the motor takes the average voltage from the pulses and thus speed control can be achieved.

2.3.2 Pin Configuration

The RoMeo v2 pins are operated in PWM mode with the pin allocations seen in the figure below. To enable the forward movement of the robot, both digital pins 4 and 7 are set to a value of 1 which specifies the direction control is to move forward. Pins 5 and 6 are connected to the left and right motor inputs where the duty cycles are altered by the inputs to enable the motor to more or less - depending on the distance left to travel.

Pin Allocation

Pin	Function
Digital 4	Motor 1 Direction control
Digital 5	Motor 1 PWM control
Digital 6	Motor 2 PWM control
Digital 7	Motor 2 Direction control

Figure 3: pin allocation for PWM mode

2.4 Interface

Matlab and simulink R2021a was the software used to interface and implement algorithms on the robot with the use of the arduino support package which supplied many of the necessary simulink modules and components.

3. Motion Control

For the robot to complete both phases, it required the ability to drive and steer. The robot had to be able to move forward in a straight line and rotate around its centre. Two algorithms were created for motion control: a Distance Control Algorithm and an Angle Control Algorithm, and PID controllers were used to implement them.

3.1 Distance Control Design & Implementation

3.1.1 Distance Control Algorithm

The distance control algorithm is as follows:

- Get total encoder ticks for left and right wheel
- Calculate distance each wheel moves:

  $$Distance\ Travelled\ = \frac{Total\ Encoder\ Ticks}{20}\ *\ 2\ *\ pi\ *\ wheelRadius$$

- Add the distance travelled by each wheel
- Divide the above value by 2 to get the distance travelled by the robot
- Subtract robot distance travelled from 1(assuming we want the robot to move 1m)
- The above calculation output will be referred to as the error
- Error fed into PID controller
- PID adjusts motor speed and activation according to error size. Large error = more activation, small error = less activation.
- PID value sent to a converter that converts velocity to angular velocity for the motors
- Once the error is reduced to 0, the input to the PID will be 0 and thus the PID controller will not feed any value for activation to the motors.

3.1.2 Distance Control Hardware

Both pins 4 and 7 were set to a value of 1 to enable forward motion with pins 5 and 6 being used for PWM control.

3.2. Angle Control Design & Implementation

3.2.1 Angle Control Algorithm

The angle control algorithm is as follows:

The circumference of the circle that the right wheel travels along can be calculated using the equation: $Circumference = pi * axleLength$. The reference input for the PID controller was therefore $pi * axleLength$. However, this input allows the robot to rotate 360° only. To allow the robot to rotate for a specified number of degrees, the input is multiplied by a gain block with a value of $\frac{1}{360}$ and a constant block which is set by the user to the desired degree of rotation. For example, if it is required that the robot rotates 90°, the constant block should be set to 90°, which will be multiplied by the gain of $\frac{1}{360}$. This factor is equivalent to a quarter, which will allow the robot to rotate 90°. The distance travelled by the right wheel of the robot, which is modelled using the distance travelled equation, was subtracted from the reference input to obtain the error. The error was fed into the PID controller which adjusts motor speed and activation according to error size.

3.2.2 Angle Control Hardware

Digital pins 4 and 7 of the Romeo V2 were used for motor control of the left and the right motors, respectively (PWM mode). Digital output pin 7 was fed in a constant of 1 (high) to instruct the right wheel to rotate forward. Digital output pin 4 was fed in a constant of 0 (low) to instruct the left wheel to rotate backwards. Digital output pins 5 and 6 were connected to the motors and used for PWM control for the right and the left wheel motors, respectively.

3.3. Motion Control Results

For the simulation, the robot was able to move in a straight line and was stopped at one meter with no overshoot. It was also able to rotate 90°, 180° and 270° within a 10% error. For the hardware part, the robot travelled in a straight line without overshoot, however, the error for all three of the rotations was slightly more than 10%. This could have have been caused by setting the speed too high or too low

4. Line Sensing

The robot's four sensors were used to detect the black line of the maze, as well as 8 different configurations: Dead-end, End of maze, Left T junction, Right T junction, Cross, T junction, Right Turn and Left Turn. Two algorithms were created: a Line Sensing Algorithm and a Line Configuration Algorithm. Once a configuration was detected, the robot needed to stop and output the configuration detected.

4.1. Line Sensing Design & Implementation

4.1.1. Line Sensing Simulation

For the purpose of simulation, the Line Sensor block was used to simulate the digital line sensors and the sensor array positions were set. This gave access to the lineValue variable which comes from the Line Sensor Block.

Even though the first two sensors are less than 18mm apart, it was acknowledged that there may be a discrepancy between how the sensors act in simulation versus in the real world and thus for the sake of accuracy, the line following only deemed sensors 2 and 3 to be necessary.

The value from the Line Sensor block was used as an input to the controller which was implemented with the use of a Stateflow chart with the default state being a "Stop" state since the robot was required to only start the line following once button S1 is pressed.

There are only two more states used by the controller to implement line following. That being a "Turn Left" and "Turn Right" state. Once the button is pressed, the robot immediately enters the "Turn Left" state and upon entry sets the velocity (v) to a constant value of 0.06 and sets the rotation (w) to 1 which indicates an anti-clockwise or left turn.

To simulate moving along the black line, the state must be changed to "Turn Right" after an appropriate amount of time. When turning left, eventually sensor 3 will not be on the line and yield a value of 0 (lineValue(3)==0) whilst sensor 2 will then be on the line due to the rotation and yield a value of 1 (lineValue(2)==1). The value of sensor 2 is used as the entry condition into the "Turn Right" state again setting the velocity (v) to 0.06 but changing the rotation (w) to -1 which indicates clockwise turning.

After a certain amount of time the opposite scenario will occur with sensor 3 == 1 and sensor 2 == 0 which indicates that the robot has turned too far right and needs to turn left, entering the "Turn Left" state with the lineValue(2==1) condition. The robot will continue to switch between these states as long as there is a line to follow.

A condition has also been placed from both the "Turn Left" and "Turn Right" states which enables re-entry into the "Stop" state; that being; after a delay of one second, if the lineValue of

both sensor 2 and 3 are 0 (neither sensor can see the black route), then the robot should stop since it has veered off course.

The v and w values were set as outputs of the stateflow chart and converted to wl and wr to be sent to the left and right motors of the robot.

Table 2 seen below showcases this logic and stateflow.

Sensor 2	Sensor 3	Current state	Next state
0	1	Turn right	Turn left
1	0	Turn left	Turn right
1	1	Turn left/right (constant state switching)	Turn left/right (constant state switching)
0	0	Turn left/right	(after 1 second) transition to Stop

Table 2: Stateflow Logic

4.1.2. Line Sensing Hardware

To implement the line sensing algorithm on the hardware, the Line Sensor block was removed and replaced by digital inputs which correlate to the pins that the digital sensors on the robot are connected to, which are pins 8, 9, 10 and 11. Since this creates multiple inputs and the stateflow chart requires a single input, a MISO device, a mux in this case, was used to allow a single output. The stateflow chart algorithm functions exactly as mentioned in the simulation section above. The output v and w is still converted to wl and wr and sent through the left and right motor lookup table however the values after passing through the LuT are now sent to the Arduino PWM blocks connected to pins 5 and 6 which are connected to the left and right motors to then adjust the motor speed and rotation.

4.1.3. Line Sensing Algorithm

- Get inputs from Digital Line sensors; pins 8, 9, 10 and 11
- Feed inputs through a MISO MUX
- Enter Stateflow chart
- Robot enters the “Stop” state by default
- Button S1 pressed
- Enters “Turn Left” state; sets v= 0.06 and w = 1
- (If both sensors are 0 for more than 1 second, enter the “Stop” state, sets v = 0, w = 0)
- Line sensor 2 == 1
- Enters “Turn Right” state
- (If both sensors are 0 for more than 1 second, enter the “Stop” state, sets v = 0, w = 0)

- Line sensor 3 == 1
- Enters “Turn Left” state; sets v= 0.06 and w = 1
- Output v and w are converted to PWM signals and fed to Pins 5 and 6.
- Repeats the state switching process until it has navigated the whole line

4.2. Line Configuration Design & Implementation

4.2.1. Line Configuration Overview

The sensors that were used in the line following part of the line configuration algorithm were sensors 2 and 3. When sensor 2 detected the environment, the robot would move to the right, and when sensor 3 detected the environment, it would move to the left.This was to ensure that the robot moved more accurately over the line which therefore made implementing the line configuration detection algorithm simpler. When the robot detects a certain line value logic while it is in either the “Turn Left” or “Turn Right” state, the line configuration algorithm will be implemented and the configuration that the robot has reached will be detected and outputted.

All four sensors were used to determine the line configurations. It was determined (by observation) which of the sensors would detect the black line once the robot reached each configuration and the logic is described in the flow charts.

Delays were used to accurately detect the different configurations. Some configurations would have the same line value logic when the configuration was reached, and therefore allowing the robot to travel a small distance further would change the line value logic and output the correct configuration.

Once the robot detects a configuration, it stops and outputs the configuration.

4.2.2. Left Turn, Left T Junction & Cross Configurations

The Cross, Left Turn and a Left T Junction configurations are all initially detected with the same line value logic: 0 0 0 1, after a delay of 0.2 seconds. When this logic is detected, the robot then moves straight (without following the line) until a new logic is detected. If a logic of 1 0 0 1 is detected, the robot will stop and the Left T Junction configuration will be outputted. If a logic of 0 0 0 0 is detected, the robot will stop and the Cross configuration will be outputted. If a logic of, 1 1 1 1 is detected the robot will stop and the Left Turn configuration will be outputted. The sensor logic is described in Figure 2 below.

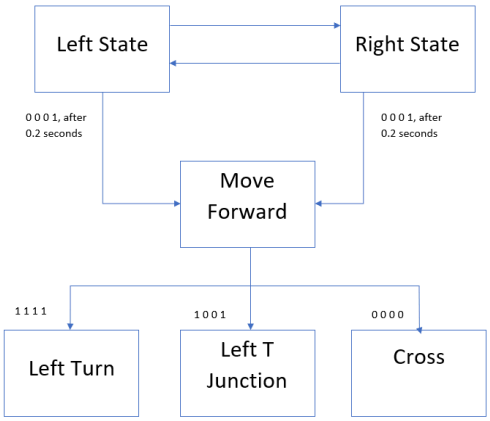

Figure 2: Cross, Left Turn and Left T Junction sensor logic

4.2.3. Right Turn & Right T Configurations

The Right Turn and Right T Junction configurations are both initially detected with the same line value logic: 1 0 0 0 after a delay of 0.2 seconds. When this logic is detected, the robot then moves straight (without following the line) until a new logic is detected. If a logic of 1 0 0 1 is detected, the robot will stop and the Right T Junction configuration will be outputted. If a logic of, 1 1 1 1 is detected, the robot will stop and the Right Turn configuration will be outputted. The sensor logic is described in Figure 3 below.

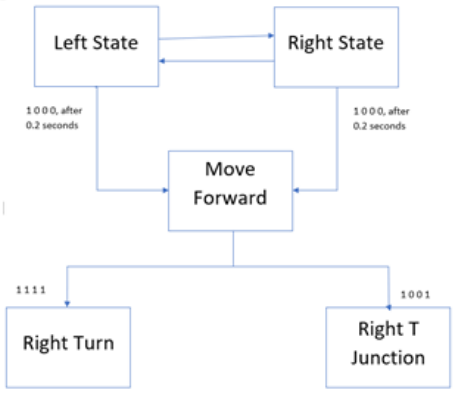

Figure 3: Right Turn and Right T Junction sensor logic

4.2.4. End of Maze & T Junction Configurations

The End of Maze and T Junction Configurations are both initially detected with the same line value logic: 0 0 0 0. When this logic is detected, the robot then moves straight (without following the line) until a new logic is detected. If a logic of 0 0 0 0 is still detected after a delay of 0.2 seconds, the End of Maze configuration will be outputted. If a logic of 1 1 1 1 is detected, The T junction configuration will be detected. The sensor logic is described in Figure 4 below.

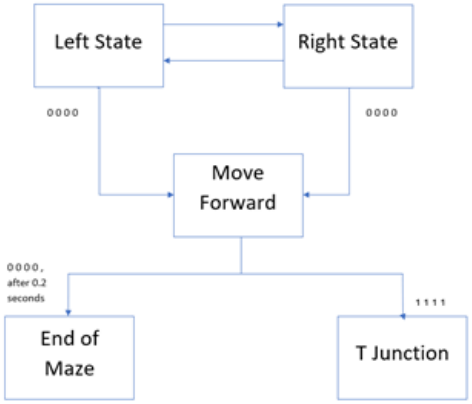

Figure 4: End of Maze and T Junction sensor logic

4.2.5. Dead End

The Dead End configuration is detected with the line value logic of 1 1 1 1. When this logic is detected, the Dead End configuration will be outputted. The sensor logic is described in Figure 5 below.

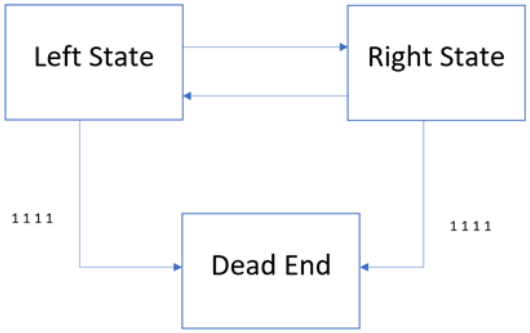

Figure 5: Dead End sensor logic

4.3. Line Sensing & Configuration Results

For both the simulation and hardware, the robot was able to follow the line of the maze accurately without straying away from the line. It was also able to detect all 8 configurations, stop at each one, and output the configuration.

5. Maze solver and shortest pathfinder algorithm design, implementation, and results

5.1 Algorithms

5.1.1 Maze Solver Algorithm Summary

- Employ wall following algorithm on button press.
- Stick to the left hand side.
- Use the line following algorithm as in milestone 2 to follow the line.
- Track how long it takes the robot to turn at a junction.
- Determine if it is a left, right or 180 degree turn.
- Attach each turn to a value.
- Store that value in a variable for processing.
- Stop when all sensors detect a line value (end of maze).
- Store the turn values in an array.
- Turn the double array to string if necessary.

5.1.2 Shortest Path Algorithm Summary

- Loop through the array and check for B(back) values.
- Analyse the preceding and following array values.
- Reduce the array of turns created with a set of rules.
- Add a counter to turn blocks in the stateflow to use as an index for the map array.
- Navigate the map normally on straight lines and stop at junctions.
- Use the map array to decide what turn to make at a junction.
- Repeat previous two points until the end of the map.

5.2 Algorithm and implementation

5.2.1 Maze Solver

The robot was first required to navigate the maze and learn it. This was done using the wall following algorithm, where the robot would travel along the left-hand side of the maze only, until it reaches the end of the maze. The line following algorithm that was implemented in Milestone 2 was used again to detect and follow the line of the maze. This time however, a value would be stored in a variable ‘s’ everytime the robot detected a certain configuration. While the robot travels along a straight line, the value stored in the variable s is 0. When the robot makes a left turn, the value 1 is stored in ‘s’. When it makes a right turn, the value 3 is stored in ‘s’. And when it turns back (after reaching a deadend), the value 2 is stored in ‘s’. Delays were used to determine whether the robot made 1 turn (rotated by 90°) or 2 turns (rotated by 180°) into a deadend. The numbers are evaluated by a To Workspace Matlab function and stored in an array. Turn signals were logged every few milliseconds. When the robot makes a back turn to

return to the correct path after turning into a deadend, the values of the turns that it made into the deadend get removed from the array. A Detect Change block was used to observe when the value of the turns changed from one to another (creating a boolean array) since the data logging stored values for s at a constant rate. The array was traversed through, and all the values of 1 in the detect change corresponded to the robot turning. The value was captured into a new array from the turn array, using the index of the detect change array. An example is shown below.

Turn Array

0	0	1	1	1	0	2

Detect Change Array(boolean)

0	0	1	0	0	1	1

New Array (path learnt)

1	0	2

0 = S; 1 = L ; 2 = B; 3 = R

New Array (path learnt) = [L S B]

Direction	Value stored in s
Straight along the line	0
Left Turn	1
Right Turn	3
Back Turn (90°)	2 (after 0.45s)
Back Turn (180°)	2 (after 0.6s)

5.2.2 Shortest Path

The Shortest Path Algorithm makes use of the array created from the maze learning algorithm. The array is checked for specific combinations of turns to determine if a dead-end was hit in order to simplify the array and create a quicker course by reducing the number of turns that deviate from the optimal path.

The array is searched for the indexes of entries that contain ‘B’ which mean the robot has had to do a 180 degree turn since this means it encountered a deadend and the path can be simplified.

Our navigation algorithm works in a manner where a right turn is initiated and if the turn continues for a longer period of time than necessary - it realises that it was actually a U turn. Bearing this in mind, The array entry preceding ‘B’ should be ‘R’.

We then focus on the array index of entry ‘B’ - 2 and ‘B’+1 for some of the reductions. These can be seen in the table below.

Combination	Reduced to	Explanation
LRBL	S	Left, Deadend, Left = Straight
LRBS	R	Left, Deadend, Straight = Right
SRBL	R	Straight, Back, Left = Right

To complete the maze, the robot then makes use of a counter to keep track of how many turns it has made and uses the array to decide which way to turn when it encounters a junction. It uses the counter as an index for the array to determine which entry it needs to use until the maze is complete.

2.4 Results

The simulation results of the maze solving were as intended with the robot completing the maze. However the turn values captured required intense processing as it was not as linear a process as expected. This resulted in the inability to reduce certain turn combinations that could have been reduced due to lack of turn information. In both the maze solver and race operation, the robot was unable to detect the end of the maze since the wall following method was employed and the sensors were never in a reliable enough position for all of them to detect the maze block.

The hardware implementation was suboptimal due to the use of delays in the algorithm. These did not transition well between simulation and real world implementation and even varied from time to time in different runs in the real world so no set delay values could be determined. This signalled the need for a better algorithm to complete this task by using a method that has less discrepancy between the simulation and real world application.

6. Conclusion

The broad project goal which entailed completion of a maze in the shortest possible time frame required many submodules to work. Three aspects of motion control had to be perfected to a certain degree and then combined in order to facilitate acceptable maze racing performance - that being general motion control, line following and sensing and lastly maze learning.

6.1 Result Overview

The motion control which was created for milestone one and required forward movement for a set distance and rotation to four angles separately performed well in simulation. In terms of hardware implementation - the forward distance control worked well however the angle control performed suboptimally which was attributed to the use of a PID controller compared to using a stateflow control method.

Milestone 2's line sensing and following worked well to follow the line in both simulation and practice however there were some non-recurring errors with the junction detection which were possibly due to hardware inconsistencies with the Line Sensors on the robot as well as problems with the maze materials.

The maze solving was the most extreme case of the simulation contradicting real world implementation. Since delays were used in the algorithm to determine and differentiate between turn types - the array which stored the turns was not correctly populated since the hardware delays did not work as intended in the real world implementation which also affected the ability to simply follow the maze.

6.2 Improvements and Recommendations

The accuracy of the robot could have been improved by sticking to stateflow control as it is a more accurate method instead of PID for the straight line movement, as well as the rotations. In terms of maze learning, since the algorithm relied on the use of delays, perhaps a better algorithm could have been created which relied on an aspect which correlated with the hardware implementation.

Throughout the project duration the time spent with the physical hardware was limited and more time would have allowed further experimentations and allowed for both better results and a more comprehensive understanding of the system.

7. Reference List

[1] T. E. o. E. Britannica, “Dead reckoning,” n.d.. [Online]. Available: https://www.britannica.com/technology/dead-reckoning-navigation. [Accessed 1 September 2021].

[2] S. M. LaValle, “13.1.2.2 A differential drive,” Cambridge University Presss, 20 April 2012. [Online]. Available: http://planning.cs.uiuc.edu/node659.html. [Accessed 1 September 2021].

[3] M. Frontz, “Wheel Encoders,” 2019. [Online]. Available: https://docs.idew.org/code-robotics/references/physical-inputs/wheel-encoders. [Accessed 1 September 2021].