

# Memo

To: Instructor and TA

From: Christian Prather

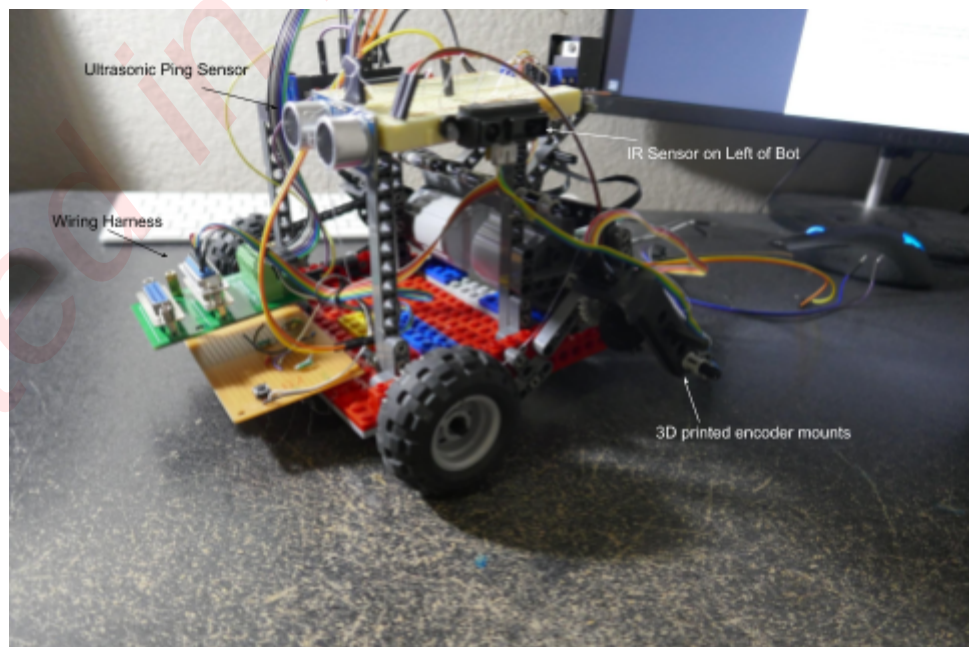
Team #: M420

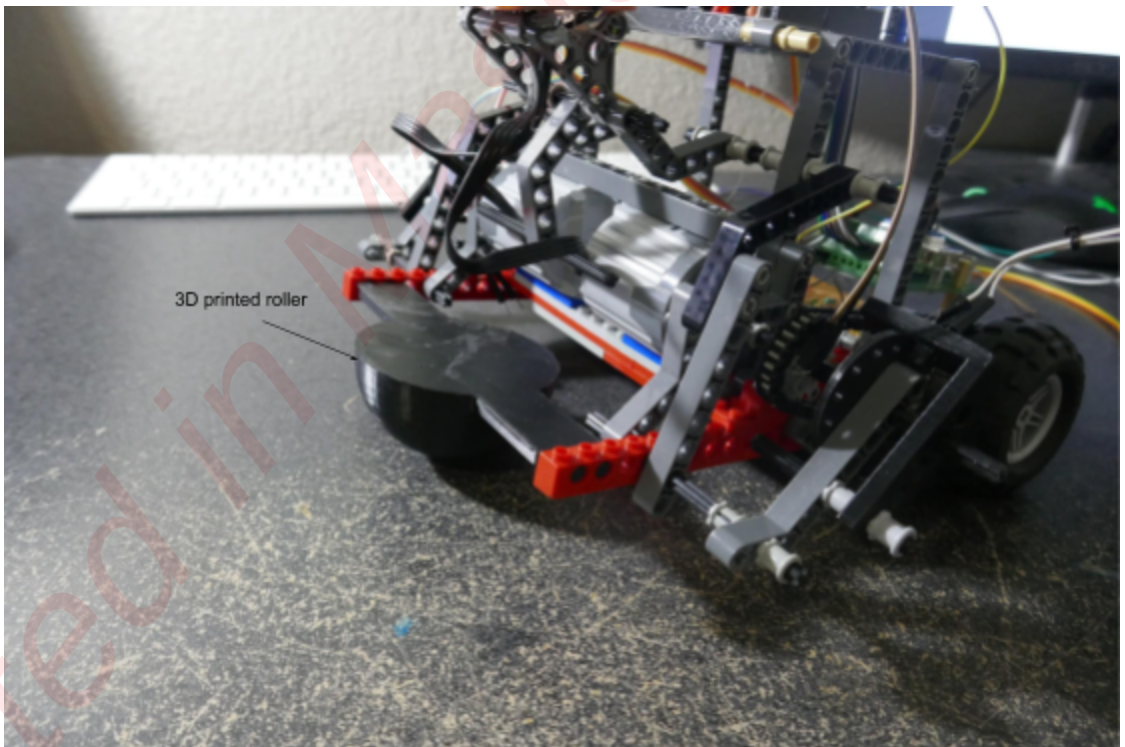
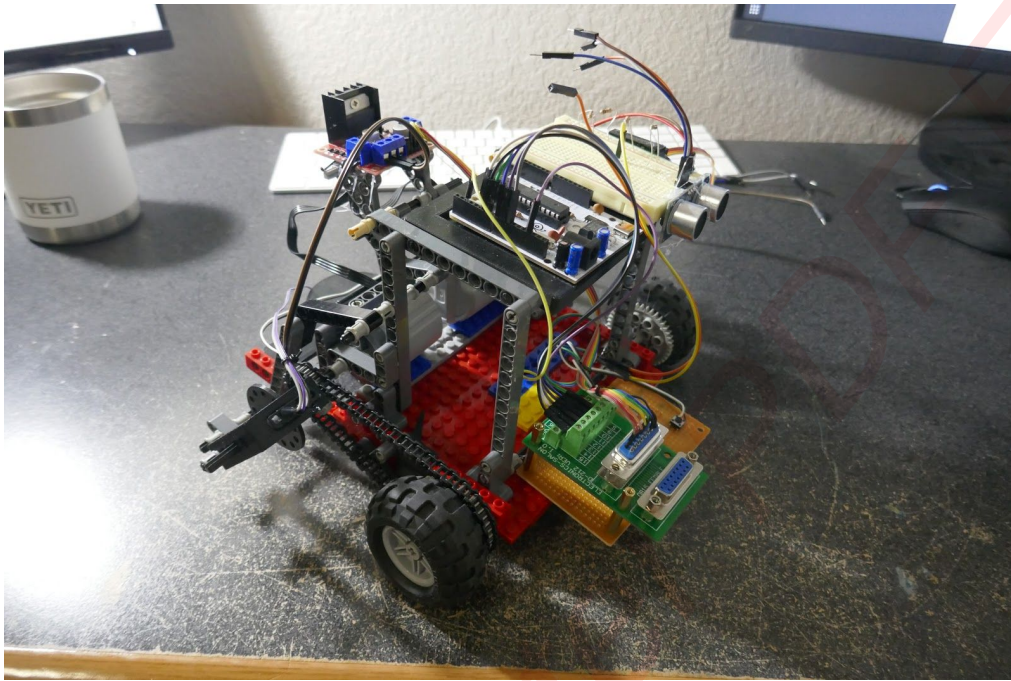
Date: 11-12-20

Re: Lab 4: Solving the Maze

## Problem Statement:

Up until now our robotic platform has only been able to handle the navigation of a predefined maze, that is we knew what the maze looked like before we ever set the robot loose on it. This is a very rare edge case however and it is far more likely that we will have to operate in an unknown environment. This lab incorporates sensors, specifically IR and Ultrasonic to provide the robot with feedback. The robot will explore its environment until it can establish its destination. This exploration is not guaranteed to have been an optimal path so at this point the robot will recalculate an optimized approach and re-run the maze.





## **Methods:**

The approach to this problem can be broken down into three primary sections, mechanical configuration, exploration, optimization.

### ***Mechanical:***

Mechanically in order to allow a robot to explore an unknown environment it needs to have some method of understanding what's around it. This was accomplished through two sensors. An ultrasonic and an IR distance sensor were used as they offered simple information on distance from robot to object. Both sensors had to go through a level of calibration as the IR needed to have a polynomial function established to convert 0-1024 analog output to a distance in cm. This calibration was done with a separate program (ir\_optimization.ino) and a simple excel file provided. The ultrasonic needed simple range limits established which was done with a trial and error check. The placement of these sensors was deliberate as well, knowing I would take a simple "walk the wall" approach to exploring/solving the maze I knew I would need to know when I was able to turn left or drive forward, this meant I would place the IR sensor on the left of the bot and the ultrasonic sensor on the front.

### ***Exploration:***

The exploration algorithm consisted of a simple decision tree based on robot state, to begin the robot is set in the center of the starting square, it follows a simple left hand rule saying if it's allowed to turn left do that otherwise drive forward. The moves taken are then recorded as well as the distance/ angle traveled. Travel was done by breaking moves this into small sections at a time, I chose to have a linear travel distance less than that of the min distance detected by the sensor as this ensures that while I am not using interrupts for the object detection I can safely drive forward without hitting an obstacle. Encoders are used to enforce a linear travel of a precise distance as discussed in the last lab. As each move is taken it is registered into an array storing the three possible moves (LEFT, RIGHT, FORWARD). The robot attempts to drive left, if that is not available then it will drive forward, and if that is not available it will turn right 90 degrees. This simulates an individual following the maze with their left hand on the wall. Once at the final destination section of the maze the robot can be notified it completed the maze with the push of a button.

### ***Optimization:***

The approach algorithm explained above while simple to implement is highly inefficient in traversing the maze in an efficient way. So a post processing algorithm is run on the recorded moves to optimize the final path. The historic moves are iterated over in an attempt to locate local patterns in the data that we know can be mapped to more optimized versions which provide the same translation from point A to B. These historic patterns to optimized mappings were stored in code and multiple arrays were used to search the historic list. (this is explained in much more detail in the code). The outcome of the optimization algorithm is a simple array of optimized moves with the concatenation of distance/ angle traveled, for example two right 90 degrees are converted to a single right 180 or multiple forwards are converted to a single forward of the summed distances.

## Results:

(Results all have an \* by the mas lab was finished at home with a simple maze construction acting as maze walls)

	Exploration time (mm:ss)	Run time	Solved
Run 1	10:36	6:23	No
Run 2	9:50	7:01	No
Run 3	10:26	7:43	No



Point 1	-0.00632	9.2	Point 1	0.043292	133.2
Point 2	0.09472	261.8	Point 2	0.083158	261.8



## Conclusions:

The robot performed fairly well, its primary downfall continues to be the mechanical design and structure of it. I was able to improve upon my prior design through the inclusion of a 3D printed ball pivot in back. This drastically reduces the chances of it being caught on something and being thrown too off track for the encoders to account for. It also did well at navigating from spot to spot and while I finished the lab at home with sudo walls I feel confident it could've done very well in its detection of the boundaries of the lab maze. I was most surprised as to the complexity involved in the optimization, while there are options of existing approaches I wanted to try and implement my own. To do this I wrote a simple C++ program (attached below main.cpp) to quickly test through ideas. My resulting algorithm is ugly to say the least and in no way would pass review of another programmer but was sufficient to get the job done. I felt I did a good job of understanding the architecture I wanted with the software and how each piece should go together, this helped me to write a program that overall I am proud of. I did not do well on the optimization algorithm as stated prior and as stated in all previous labs I am unhappy with my platform's mechanical design, though I am happy with the parts I have designed with Solidworks and printed. If I had to redo the lab I would focus my attention on two areas to adjust, one would be my mechanical structure, I had some issues with my encoder mounting causing inconsistent readings and I had multiple times when parts would come apart as certain sections are under tension due to forced fits. I would also spend more time on my optimization algorithm as it is about as efficient as a potato. This would require some better architecture in pattern matching primarily.

## References:

Lecture slides and the Arduino standard examples library were used as reference for this project

## Appendices:

See

[https://github.com/Christian-Prather/Mines-Robotics/blob/main/Lab4/custom\\_lab\\_4/docs/latex/refman.pdf](https://github.com/Christian-Prather/Mines-Robotics/blob/main/Lab4/custom_lab_4/docs/latex/refman.pdf) for full code documentation

```
1  /**
2   @file custom_lab_4.ino
3   @author Christian Prather
4   @brief A basic feedback controlled system for an Arduino based robot with
5   ultrasonic
6   and IR distance sensors
7   @version 0.1
8   @date 2020-10-23
9  */
10
11
12  /*! \mainpage Lab 4 Code Documentation
13   *
14   */
15
16  /// Libraries for interrupts and PID
17  #include <PinChangeInt.h>
18  #include <PID_v1.h>
19  #include <SR04.h>
20
21  /// Global Defines
22  /// Motor driver connections
23  #define IN1 5
24  #define IN2 6
25  #define IN3 7
26  #define IN4 8
27
28  /// Motor control
29  #define A 0
30  #define B 1
31  #define pwmA 3
32  #define dirA 9
33  #define pwmB 4
34  #define dirB 13
35
36  /// Start stop button
37  #define pushButton 2
38
39  /// Drive constants - dependent on robot configuration
40  #define EncoderCountsPerRev 12.0
41  #define DistancePerRev 51.0
42  #define DegreesPerRev 27.0
43
44  #define EncoderMotorLeft 7
45  #define EncoderMotorRight 8
46
47  /// Lab specific variables
48  double leftEncoderCount = 0;
49  double rightEncoderCount = 0;
50  int wallDist = 5; /// CM
51  #define DISTANCE_SEG 10
52
53  /// Enum defines
54  #define FORWARD 0
55  #define RIGHT 1
56  #define LEFT 2
57
58  /// IR sensors
59  int irSensor = A0;
```

```
60
61 /// Ultrasonic sensors
62 int trig = 12;
63 int echo = 11;
64 SR04 sideUS = SR04(trig, echo);
65
66 /// Default motor pwm values
67 int motorLeft_PWM = 180;
68 int motorRight_PWM = 200;
69
70 /// Time it takes to move 90 degrees
71 int milliSecondsPer90Deg = 900;
72
73 /// How many encoder counts for given distance
74 double desiredCount;
75
76 int movesCount = 0;
77 // Global array for tracking move order (move, distance) or (move, degree)
78 int moveList[50];
79
80 int optimizedMoves[50];
81
82 /**
83  @brief PID values
84  setpoints = desired counts, output = PWM, input = current counts
85 */
86 double leftOutput, rightOutput;
87 PID leftPID(&leftEncoderCount, &leftOutput, &desiredCount, 2, 5, 2, DIRECT);
88 PID rightPID(&rightEncoderCount, &rightOutput, &desiredCount, 2, 5, 2,
DIRECT);
89
90 /**
91  @brief Helper function for setting the PWM back to default value
92 */
93 void resetPWM()
94 {
95     motorLeft_PWM = 180;
96     motorRight_PWM = 200;
97 }
98
99 /**
100  @brief Run the PID loop calculation and set out put to motors output in
PWM
101
102 */
103 void adjustPWM()
104 {
105     // Compute the pid values
106     leftPID.Compute();
107     rightPID.Compute();
108
109     // Set the pid values within range
110     motorLeft_PWM = constrain(leftOutput, 150, 250);
111     motorRight_PWM = constrain(rightOutput, 150, 235);
112     Serial.print("Left PWM: ");
113     Serial.print(motorLeft_PWM);
114     Serial.print(" ");
115     Serial.println(leftEncoderCount);
116     Serial.print("Right PWM: ");
117     Serial.print(motorRight_PWM);
```

```
118     Serial.print(" ");
119     Serial.println(rightEncoderCount);
120 }
121
122 /**
123  * @brief ISR for left encoder
124  */
125 void indexLeftEncoderCount()
126 {
127     leftEncoderCount++;
128     //Serial.println("Left Encoder ++");
129 }
130
131 /**
132  * @brief ISR for incrementing right encoder
133  */
134 void indexRightEncoderCount()
135 {
136     rightEncoderCount++;
137     //Serial.println("Right Encoder ++");
138 }
139
140 /**
141  * @brief Calculate how many encoder counts we expect given the distance
142  * provided
143  * based on the bot intrinsics
144  * @param distance
145  */
146 void calculateDesiredCount(int distance)
147 {
148     double revolutionsRequired = distance / DistancePerRev;
149
150     desiredCount = revolutionsRequired * EncoderCountsPerRev;
151     // Reset encoder counts
152     leftEncoderCount = 0;
153     rightEncoderCount = 0;
154     Serial.print("Desired Count: ");
155     Serial.println(desiredCount);
156 }
157
158 /**
159  * @brief Calculate how many encoder counts we expect given the degrees
160  * provided
161  * @param degrees
162  */
163 void calculateDesiredCountTurn(int degrees)
164 {
165     double revolutionsRequired = degrees / DegreesPerRev;
166     desiredCount = revolutionsRequired * EncoderCountsPerRev;
167     leftEncoderCount = 0;
168     rightEncoderCount = 0;
169     Serial.print("Desired Count: ");
170     Serial.println(desiredCount);
171 }
172
173 /**
174  * @brief Turn bot to given degrees
175  */
```



```
176 @param degrees
177 */
178 void turnRight(int degrees)
179 {
180     resetPWM(); // Reset pwm
181     calculateDesiredCountTurn(degrees);
182     // While the encoders are not correct adjust PWM with PID loop
183     // Loop until the encoders read correct
184
185     while ((desiredCount - rightEncoderCount) > 3)
186     {
187         adjustPWM();
188         //To drive forward, motors go in the same direction
189
190         if ((desiredCount - leftEncoderCount) > 3)
191         {
192             run_motor(A, -motorLeft_PWM); //change PWM to your calibrations
193         }
194         if ((desiredCount - rightEncoderCount) > 3)
195         {
196             run_motor(B, motorRight_PWM); //change PWM to your calibrations
197         }
198     }
199
200     // motors stop
201     run_motor(A, 0);
202     run_motor(B, 0);
203     Serial.println("Done driving Right");
204     Serial.print("L: ");
205     Serial.println(leftEncoderCount);
206     Serial.print("R: ");
207     Serial.println(rightEncoderCount);
208 }
209
210 /**
211  @brief Turn bot right to given degrees
212
213  @param degrees
214  */
215 void turnLeft(int degrees)
216 {
217     resetPWM();
218     calculateDesiredCountTurn(degrees);
219
220     // Loop until the encoders read correct
221
222     while ((desiredCount - leftEncoderCount) > 3)
223     {
224         adjustPWM();
225         //To drive forward, motors go in the same direction
226
227         if ((desiredCount - leftEncoderCount) > 3)
228         {
229             run_motor(A, motorLeft_PWM); //change PWM to your calibrations
230         }
231         if ((desiredCount - rightEncoderCount) > 3)
232         {
233             run_motor(B, -motorRight_PWM); //change PWM to your calibrations
234         }
235     }
```

```
236
237 // motors stop
238 run_motor(A, 0);
239 run_motor(B, 0);
240 Serial.println("Done driving Left");
241 Serial.print("L: ");
242 Serial.println(leftEncoderCount);
243 Serial.print("R: ");
244 Serial.println(rightEncoderCount);
245 }
246
247 /**
248  @brief Function to drive bot forward until encoders are within range
249
250  @param distance
251  */
252 void driveForward(int distance)
253 {
254     Serial.println("Driving Forward...");
255     resetPWM();
256     calculateDesiredCount(distance);
257
258     // Loop until the encoders read correct
259
260     while ((desiredCount - leftEncoderCount) > 3 || (desiredCount -
rightEncoderCount) > 3)
261     {
262         adjustPWM();
263         //To drive forward, motors go in the same direction
264
265         if ((desiredCount - leftEncoderCount) > 3)
266         {
267             run_motor(A, -motorLeft_PWM); //change PWM to your calibrations
268         }
269         if ((desiredCount - rightEncoderCount) > 3)
270         {
271             run_motor(B, -motorRight_PWM); //change PWM to your calibrations
272         }
273     }
274
275     // motors stop
276     run_motor(A, 0);
277     run_motor(B, 0);
278     Serial.println("Done driving forward");
279     Serial.print("L: ");
280     Serial.println(leftEncoderCount);
281     Serial.print("R: ");
282     Serial.println(rightEncoderCount);
283 }
284
285 /**
286  @brief Drive the bot backwards
287
288  @param distance
289  */
290 void driveBackward(int distance)
291 {
292     resetPWM();
293     calculateDesiredCount(distance);
294 }
```

```
295 // Loop until the encoders read correct
296
297 while ((desiredCount - leftEncoderCount) > 3 || (desiredCount -
rightEncoderCount) > 3)
298 {
299     adjustPWM();
300     //To drive backward, motors go in the same direction
301
302     if ((desiredCount - leftEncoderCount) > 3)
303     {
304         run_motor(A, motorLeft_PWM); //change PWM to your calibrations
305     }
306     if ((desiredCount - rightEncoderCount) > 3)
307     {
308         run_motor(B, motorRight_PWM); //change PWM to your calibrations
309     }
310 }
311
312 // motors stop
313 run_motor(A, 0);
314 run_motor(B, 0);
315 Serial.println("Done driving backwards");
316 Serial.print("L: ");
317 Serial.println(leftEncoderCount);
318 Serial.print("R: ");
319 Serial.println(rightEncoderCount);
320 }
321
322 /**
323  * @brief Function for reading the distance sensors
324  *
325  * @param sensor 0 = IR, 1 = Ultrasonic
326  * @return float distance (cm)
327  */
328 float readDistance(int sensor)
329 {
330     float distance = 0.0;
331     switch (sensor)
332     {
333     case 0:
334         int reading = analogRead(irSensor);
335         distance = ((0.00031) * reading) + 0.002;
336         break;
337     case 1:
338         distance = sideUS.Distance();
339         break;
340
341     default:
342         break;
343     }
344     return distance;
345 }
346
347 /**
348  * @brief The exploritory function to allow the system to navigate unseen
349  * environment
350  * Using left hand rule
351  */
352 void explore()
353 {
```

```

353 while (digitalRead(pushButton) == 1)
354 {
355     float front = readDistance(0);
356     float side = readDistance(1);
357
358     /// There is no wall to left of bot
359     if (side > wallDist)
360     {
361         turnLeft(90);
362         /// Not recording degrees as the assumption is every turn on 90
degrees
363         moveList[movesCount] = "LEFT";
364         movesCount++;
365     }
366     /// Can drive forward
367     else if (front > wallDist)
368     {
369         driveForward(DISTANCE_SEG);
370         moveList[movesCount] = "FORWARD";
371         movesCount++;
372     }
373     /// Trapped turn Right
374     else
375     {
376         turnRight(90);
377         moveList[movesCount] = "RIGHT";
378         movesCount++;
379     }
380 }
381 }
382
383 /**
384  * @brief This is what youve all been waiting for one darn good looking
385  * solution to maze optimatation. Iterates over the movesList looking for
386  * specific patterns it can reduce into simpler sequences
387  * Key assumption: Explored using Left hand rule
388  */
389 void optimize()
390 {
391     /// Key patterns 0 = F, 1 = R, 2 = L, 3 = DELETE
392     int keyPatterns_6[2][6] = {{0, 0, 1, 1, 0, 0}, {2, 0, 1, 1, 0, 2}};
393     int keyPatterns_5[2][5] = {{2, 0, 1, 1, 0}, {0, 1, 1, 0, 2}};
394     int keyPatterns_4[1][4] = {{0, 1, 1, 0}};
395
396     int optimizedPattern_6[1][8] = {{FORWARD, 2 * DISTANCE_SEG, RIGHT, 90,
RIGHT, 90, FORWARD, DISTANCE_SEG}};
397     int optimizedPattern_5[2][2] = {{RIGHT, 90}, {RIGHT, 90}};
398     int optimizedPatter_4[1][4] = {{LEFT, 90, LEFT, 90}};
399     /** This is going to be checking in a priority tree fashion given highest
priority
400     * given highest priority patterns are 6 long then 5 long then 4 I can
batch this
401     */
402     for (int i = 0; i < movesCount; i++)
403     {
404         /// Get next move in explored list
405         // int move = moveList[i];
406         /// Get next 6 moves if enough in list
407
408         // Check 6 out first

```

```

409     int future[6];
410     for (int j = 0; j < 6; j++)
411     {
412         if ((j + i) < movesCount)
413         {
414             future[j] = moveList[j + i];
415         }
416     }
417     int tracker = 0;
418     for (auto potential : keyPatterns_6)
419     {
420         bool match = true;
421         for (int m = 0; m < 6; m++)
422         {
423             if (future[m] != potential[m])
424             {
425                 match = false;
426             }
427         }
428         if (match)
429         {
430             int keyPatternLength = (sizeof(potential) /
sizeof(potential[0]));
431             // Insert optimized move
432             for (int x = 0; x < (sizeof(optimizedPattern_6[tracker]) /
sizeof(optimizedPattern_6[tracker][0])); x++)
433             {
434                 if (optimizedPattern_6[tracker][x] != 3)
435                 {
436                     optimizedMoves[x] = optimizedPattern_6[tracker][x];
437                 }
438             }
439             i = i + 6;
440             break;
441         }
442         tracker = tracker + 1;
443     }
444
445     ////////////////////////////////////////
446     //
447     // Check 5 out first
448     int future_5[5];
449     for (int j = 0; j < 5; j++)
450     {
451         if ((j + i) < movesCount)
452         {
453             future_5[j] = moveList[j + i];
454         }
455     }
456     tracker = 0;
457     for (auto potential : keyPatterns_6)
458     {
459         bool match = true;
460         for (int m = 0; m < 5; m++)
461         {
462             if (future_5[m] != potential[m])
463             {
464                 match = false;

```



```

465     }
466     if (match)
467     {
468         int keyPatternLength = (sizeof(potential) /
sizeof(potential[0]));
469         // Insert optimized move
470         for (int x = 0; x < (sizeof(optimizedPattern_6[tracker]) /
sizeof(optimizedPattern_6[tracker][0])); x++)
471         {
472             if (optimizedPattern_6[tracker][x] != 3)
473             {
474                 optimizedMoves[x] = optimizedPattern_6[tracker][x];
475             }
476         }
477         i = i + 5;
478         break;
479     }
480     tracker = tracker + 1;
481 }
482
483
////////////////////////////////////
//
484 // Check 4 out first
485 int future_4[4];
486 for (int j = 0; j < 4; j++)
487 {
488     if ((j + i) < movesCount)
489     {
490         future_4[j] = moveList[j + i];
491     }
492 }
493 tracker = 0;
494 for (auto potential : keyPatterns_6)
495 {
496     bool match = true;
497     for (int m = 0; m < 4; m++)
498     {
499         if (future_4[m] != potential[m])
500         {
501             match = false;
502         }
503     }
504     if (match)
505     {
506         int keyPatternLength = (sizeof(potential) /
sizeof(potential[0]));
507         // Insert optimized move
508         for (int x = 0; x < (sizeof(optimizedPattern_6[tracker]) /
sizeof(optimizedPattern_6[tracker][0])); x++)
509         {
510             if (optimizedPattern_6[tracker][x] != 3)
511             {
512                 optimizedMoves[x] = optimizedPattern_6[tracker][x];
513             }
514         }
515         i = i + 4;
516         break;
517     }
518     tracker = tracker + 1;

```

```

519     }
520 }
521 }
522
523
524 /**
525  @brief Function for configuration of pin states and interrupts
526  */
527 void configure()
528 {
529     // set up the motor drive ports
530     pinMode(pwmA, OUTPUT);
531     pinMode(dirA, OUTPUT);
532     pinMode(pwmB, OUTPUT);
533     pinMode(dirB, OUTPUT);
534
535     pinMode(pushButton, INPUT_PULLUP);
536
537     pinMode(EncoderMotorLeft, INPUT_PULLUP); //set the pin to input
538     PCintPort::attachInterrupt(EncoderMotorLeft, indexLeftEncoderCount,
CHANGE);
539
540     pinMode(EncoderMotorRight, INPUT_PULLUP); //set the pin to input
541     PCintPort::attachInterrupt(EncoderMotorRight, indexRightEncoderCount,
CHANGE);
542 }
543
544 /**
545  @brief Default behavior when not driving, waits for the pushButton to
546  be pressed so it can execute next command
547  Blocking function
548  */
549 void idle()
550 {
551     Serial.println("Idle..");
552     while (digitalRead(pushButton) == 1)
553         ; // wait for button push
554     while (digitalRead(pushButton) == 0)
555         ; // wait for button release
556     delay(2000); // Give time to move hand
557 }
558
559 /**
560  @brief Entry point of program handles serial setup and PID config
561  */
562 void setup()
563 {
564     Serial.begin(9600);
565     Serial.println("Setting up.....");
566     configure();
567     leftPID.SetMode(AUTOMATIC);
568     rightPID.SetMode(AUTOMATIC);
569 }
570
571 /**
572  @brief This is the logic to execute if we hit a push button
573  ideally this is never executed as we should never actually hit the walls
574  */
575 void react_left()
576 {

```

```
577 // TODO: Check which button was hit
578
579 driveBackward(20);
580 turnRight(30);
581 }
582 void react_right()
583 {
584 // TODO: Check which button was hit
585
586 driveBackward(20);
587 turnLeft(30);
588 }
589 void react_forward()
590 {
591 // TODO: Check which button was hit
592 driveBackward(50);
593 }
594
595 /**
596 @brief Main drive execution of program, iterates through moves list
executing
597 next move with corresponding distance or degrees
598 */
599 void drive()
600 {
601 // Iterate over the list jumping by two each time
602 for (int i = 0; i < sizeof(optimizedMoves); i += 2)
603 {
604 idle();
605 switch (moveList[i])
606 {
607 case LEFT:
608 turnLeft(moveList[i + 1]);
609 break;
610 case RIGHT:
611 turnRight(moveList[i + 1]);
612 break;
613 case FORWARD:
614 driveForward(moveList[i + 1]);
615 break;
616 default:
617 break;
618 }
619 }
620 }
621 }
622
623 /**
624 @brief Loop execution of the program
625 */
626 void loop()
627 {
628 explore();
629 optimize();
630 drive();
631 }
632
```

```
1  /// Basic calibration program for IR sensor
2  int irSensor = A1;
3  int pushButton = 2;
4
5  void setup()
6  {
7      pinMode(pushButton, INPUT_PULLUP);
8      Serial.begin(9600);
9      Serial.println("Setup Complete..");
10 }
11
12 void loop()
13 {
14     Serial.println("Waiting...");
15     while (digitalRead(pushButton) == 1)
16         ; // wait for button push
17     while (digitalRead(pushButton) == 0)
18         ; // wait for button release
19
20     double averageValue = 0;
21     for (int i = 0; i < 5; i++)
22     {
23         averageValue += analogRead(irSensor);
24         delay(250);
25     }
26     averageValue = averageValue / 5;
27     Serial.print("Sensor Reading: ");
28     Serial.println(averageValue);
29 }
30 }
31
```

```

1  /**
2   * @file main.cpp
3   * @author Christian Prather
4   * @brief Testing algorithm for the optimization algorithm
5   * @version 0.1
6   * @date 2020-11-12
7   *
8   * @copyright Copyright (c) 2020
9   *
10  */
11 #include <iostream>
12 using namespace std;
13 /// Enum defines
14 #define FORWARD 0
15 #define RIGHT 1
16 #define LEFT 2
17 #define DISTANCE_SEG 10
18
19 int movesCount = 6;
20 // Global array for tracking move order (move, distance) or (move, degree)
21 int moveList[50] = {FORWARD, FORWARD, RIGHT, RIGHT, FORWARD, FORWARD};
22
23 int optimizedMoves[50];
24
25 void optimize()
26 {
27     /// Key patterns 0 = F, 1 = R, 2 = L, 3 = DELETE
28     int keyPatterns_6[2][6] = {{0, 0, 1, 1, 0, 0}, {2, 0, 1, 1, 0, 2}};
29     int keyPatterns_5[2][5] = {{2, 0, 1, 1, 0}, {0, 1, 1, 0, 2}};
30     int keyPatterns_4[1][4] = {{0, 1, 1, 0}};
31
32     int optimizedPattern_6[1][8] = {{FORWARD, 2 * DISTANCE_SEG, RIGHT, 90,
33     RIGHT, 90, FORWARD, DISTANCE_SEG}};
34     int optimizedPattern_5[2][2] = {{RIGHT, 90}, {RIGHT, 90}};
35     int optimizedPatter_4[1][4] = {{LEFT, 90, LEFT, 90}};
36     /** This is going to be checking in a priority tree fashion given highest
37     priority
38     * given highest priority patterns are 6 long then 5 long then 4 I can
39     batch this
40     */
41     for (int i = 0; i < movesCount; i++)
42     {
43         /// Get next move in explored list
44         /// int move = moveList[i];
45         /// Get next 6 moves if enough in list
46
47         /// Check 6 out first
48         int future[6];
49         for (int j = 0; j < 6; j++)
50         {
51             if ((j + i) < movesCount)
52             {
53                 /// j (0-5) i (0-movesCount)
54                 future[j] = moveList[j + i];
55                 cout << "Move: " << future[j] << endl;
56             }
57         }
58         int tracker = 0;
59         for (auto potential : keyPatterns_6)

```



```

58     {
59         bool match = true;
60         for (int m = 0; m < 6; m++)
61         {
62             if (future[m] != potential[m])
63             {
64                 match = false;
65             }
66         }
67         if (match)
68         {
69             cout << "Matched " << tracker << endl;
70             int keyPatternLength = (sizeof(potential) /
sizeof(potential[0]));
71             // Insert optimized move
72             for (int x = 0; x < (sizeof(optimizedPattern_6[tracker]) /
sizeof(optimizedPattern_6[tracker][0])); x++)
73             {
74                 if (optimizedPattern_6[tracker][x] != 3)
75                 {
76                     optimizedMoves[x] = optimizedPattern_6[tracker][x];
77                 }
78             }
79             i = i + 6;
80             break;
81         }
82         tracker = tracker + 1;
83     }
84
85     ////////////////////////////////////////
86     //
87     // Check 5 out first
88     int future_5[5];
89     for (int j = 0; j < 5; j++)
90     {
91         if ((j + i) < movesCount)
92         {
93             /// j (0-5) i (0-movesCount)
94             future_5[j] = moveList[j + i];
95             cout << "Move5: " << future_5[j] << endl;
96         }
97     }
98     tracker = 0;
99     for (auto potential : keyPatterns_6)
100     {
101         bool match = true;
102         for (int m = 0; m < 5; m++)
103         {
104             if (future_5[m] != potential[m])
105             {
106                 match = false;
107             }
108         }
109         if (match)
110         {
111             cout << "Matched " << tracker << endl;
112             int keyPatternLength = (sizeof(potential) /
sizeof(potential[0]));
113             // Insert optimized move

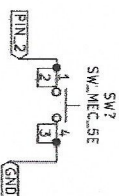
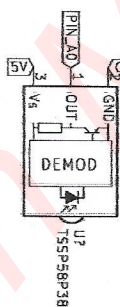
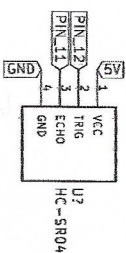
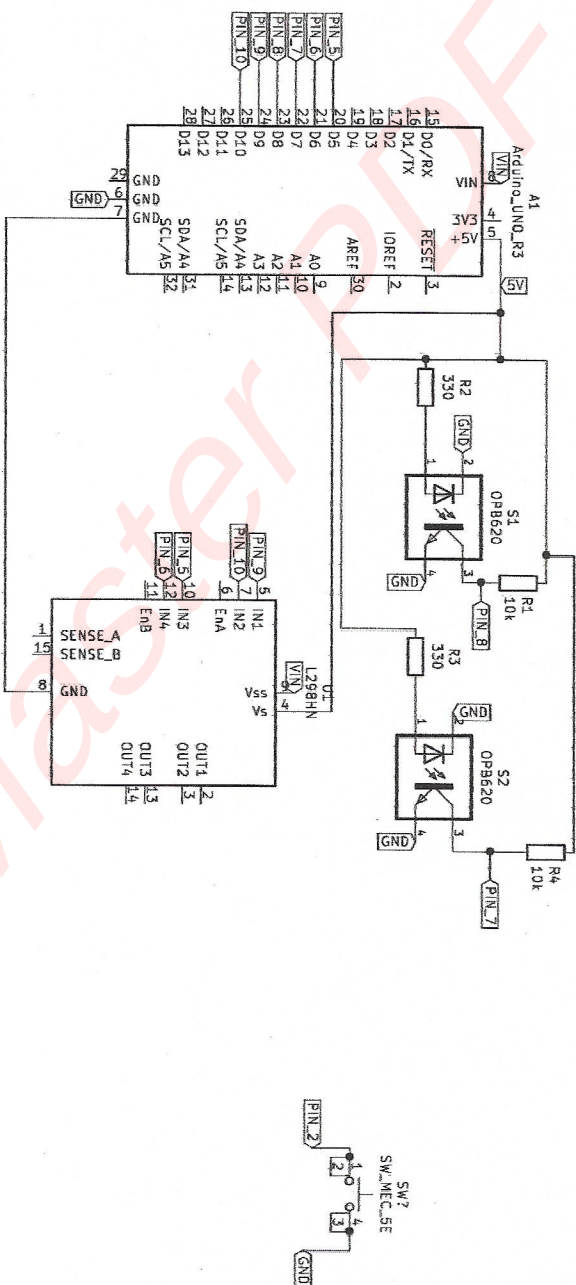
```

```

113         for (int x = 0; x < (sizeof(optimizedPattern_6[tracker]) /
sizeof(optimizedPattern_6[tracker][0])); x++)
114         {
115             if (optimizedPattern_6[tracker][x] != 3)
116             {
117                 optimizedMoves[x] = optimizedPattern_6[tracker][x];
118             }
119         }
120         i = i+ 5;
121         break;
122     }
123     tracker = tracker + 1;
124 }
125
126
127 ////////////////////////////////////////////////////
128 //
129 // Check 4 out first
130 int future_4[4];
131 for (int j = 0; j < 4; j++)
132 {
133     if ((j + i) < movesCount)
134     {
135         /// j (0-5) i (0-movesCount)
136         future_4[j] = moveList[j + i];
137         cout << "Move4: " << future_4[j] << endl;
138     }
139 }
140 tracker = 0;
141 for (auto potential : keyPatterns_6)
142 {
143     bool match = true;
144     for (int m = 0; m < 4; m++)
145     {
146         if (future_4[m] != potential[m])
147         {
148             match = false;
149         }
150     }
151     if (match)
152     {
153         cout << "Matched " << tracker << endl;
154         int keyPatternLength = (sizeof(potential) /
sizeof(potential[0]));
155         // Insert optimized move
156         for (int x = 0; x < (sizeof(optimizedPattern_6[tracker]) /
sizeof(optimizedPattern_6[tracker][0])); x++)
157         {
158             if (optimizedPattern_6[tracker][x] != 3)
159             {
160                 optimizedMoves[x] = optimizedPattern_6[tracker][x];
161             }
162         }
163         i = i+4;
164         break;
165     }
166     tracker = tracker + 1;
167 }
168 }
169 }

```

```
168
169 int main()
170 {
171     optimize();
172     cout << "Optimized" << endl;
173     for (auto move : optimizedMoves)
174     {
175         cout << move << ", ";
176     }
177 }
```



Author: Christian Prether

Sheet: /  
File: robotLab4.sch

Title: **Lab 3 Robot**

Size: A4

Date: 2020-10-19

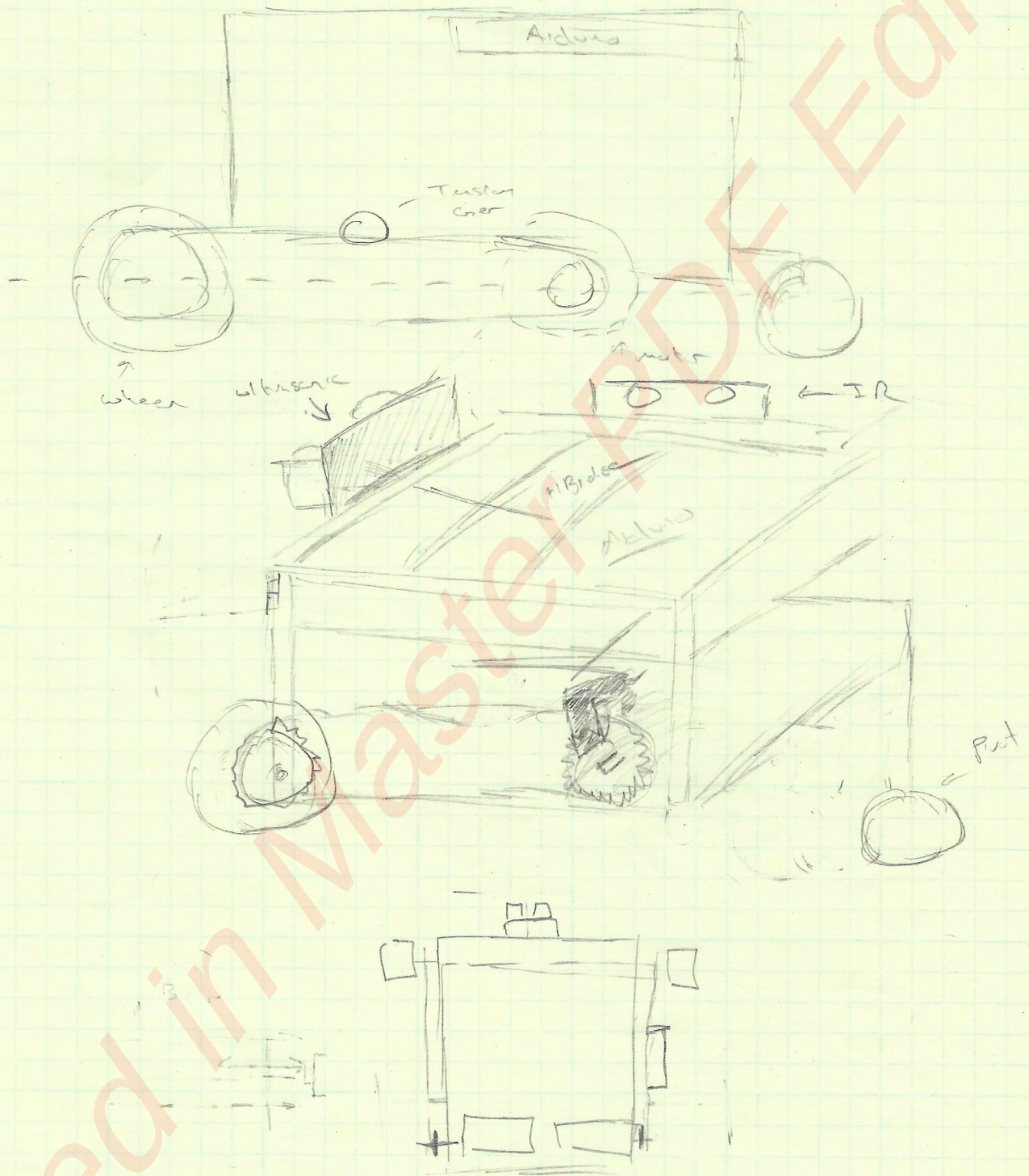
KiCad E.D.A. eeschema 5.1.5+dfsg1-2build2

Rev Version 3

Id: 1/1



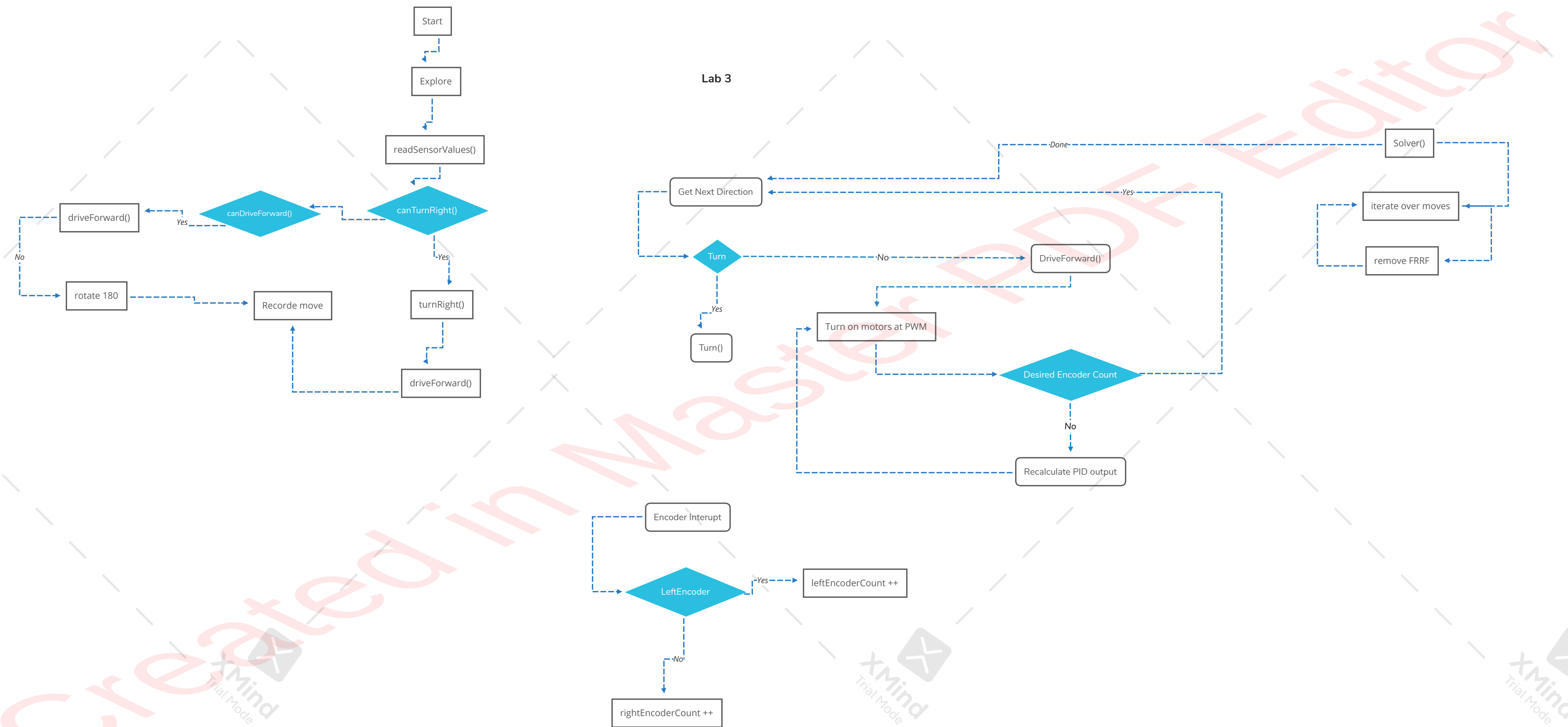
Mechanical

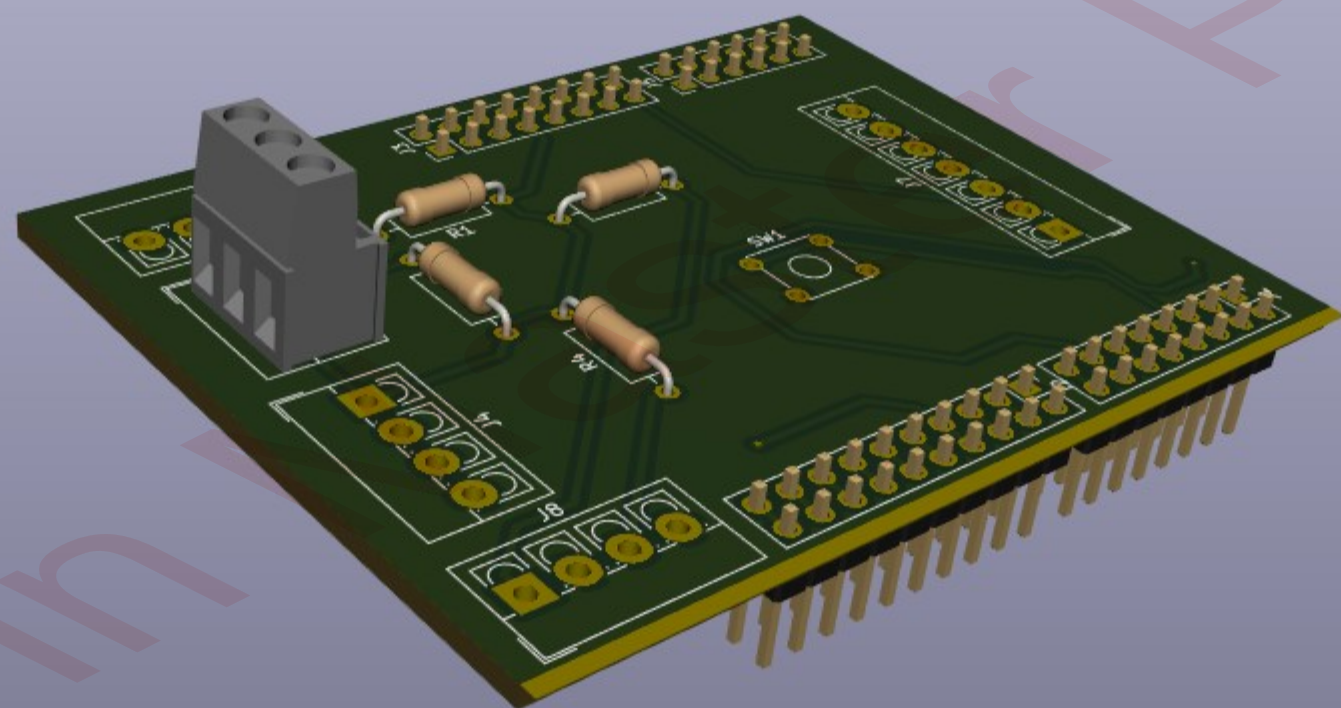
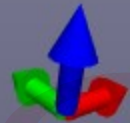


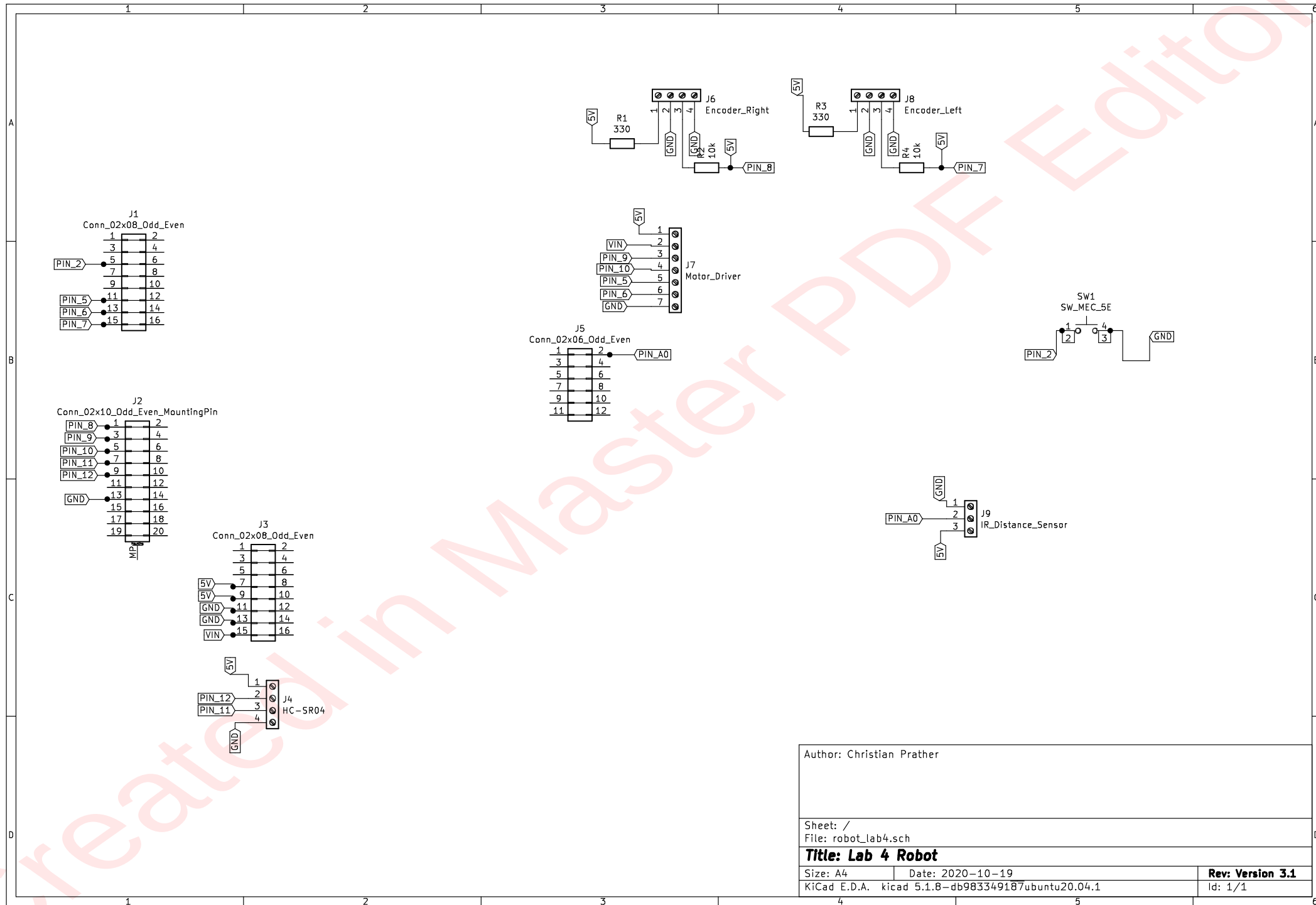
ms



### Lab 3







Author: Christian Prather		
Sheet: /		
File: robot_lab4.sch		
<b>Title: Lab 4 Robot</b>		
Size: A4	Date: 2020-10-19	Rev: Version 3.1
KiCad E.D.A. kicad 5.1.8-db983349187ubuntu20.04.1		Id: 1/1