Colorado School of Mines

Memo

To: Instructor and TA

From: Christian Prather

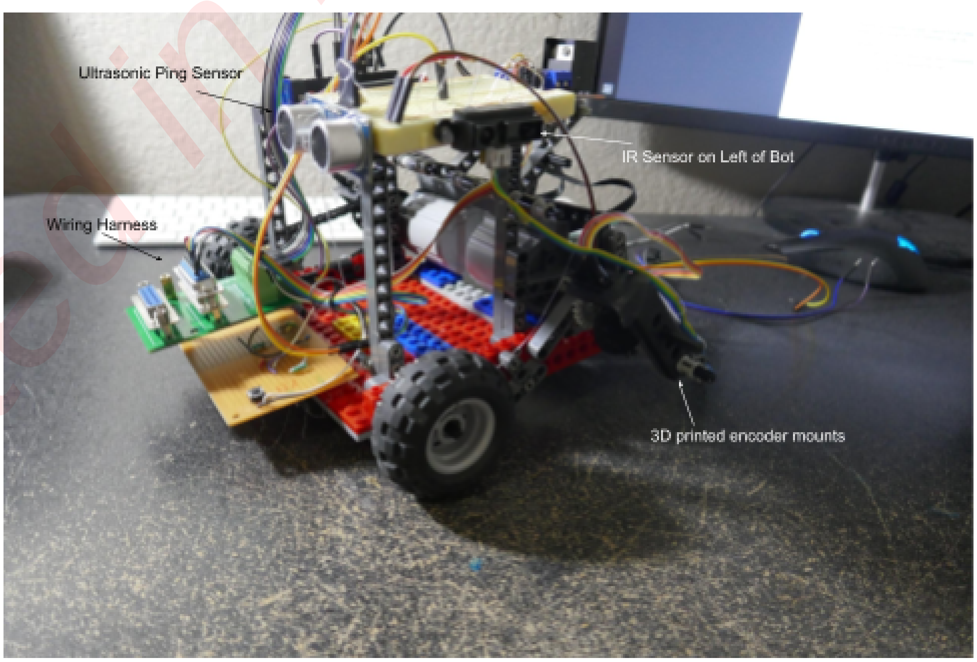
Team #: M420

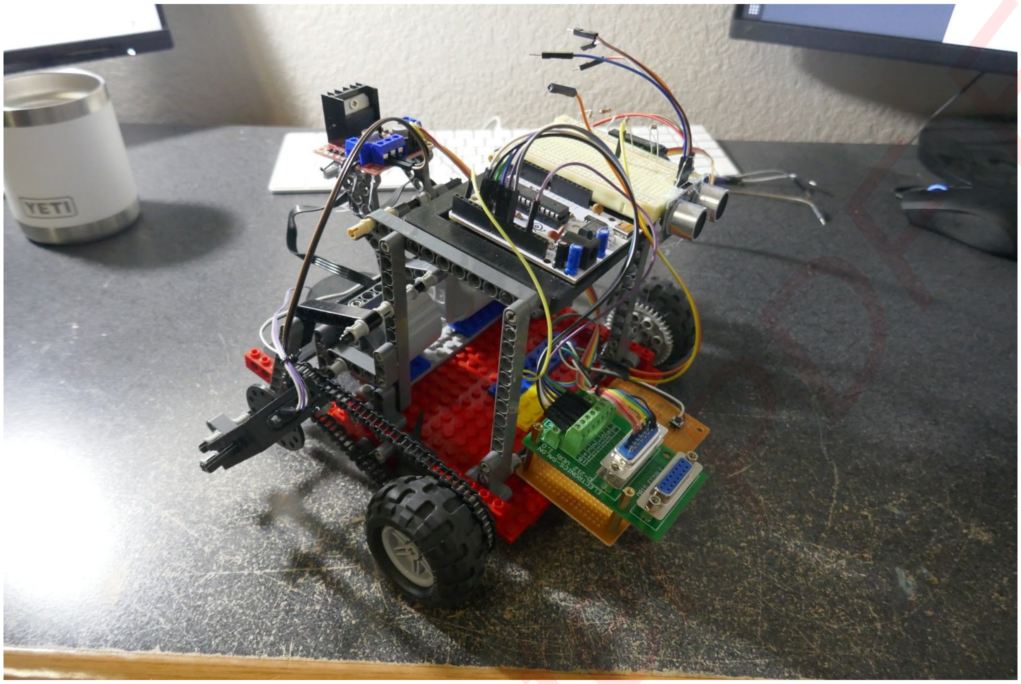
Date: 11-12-20

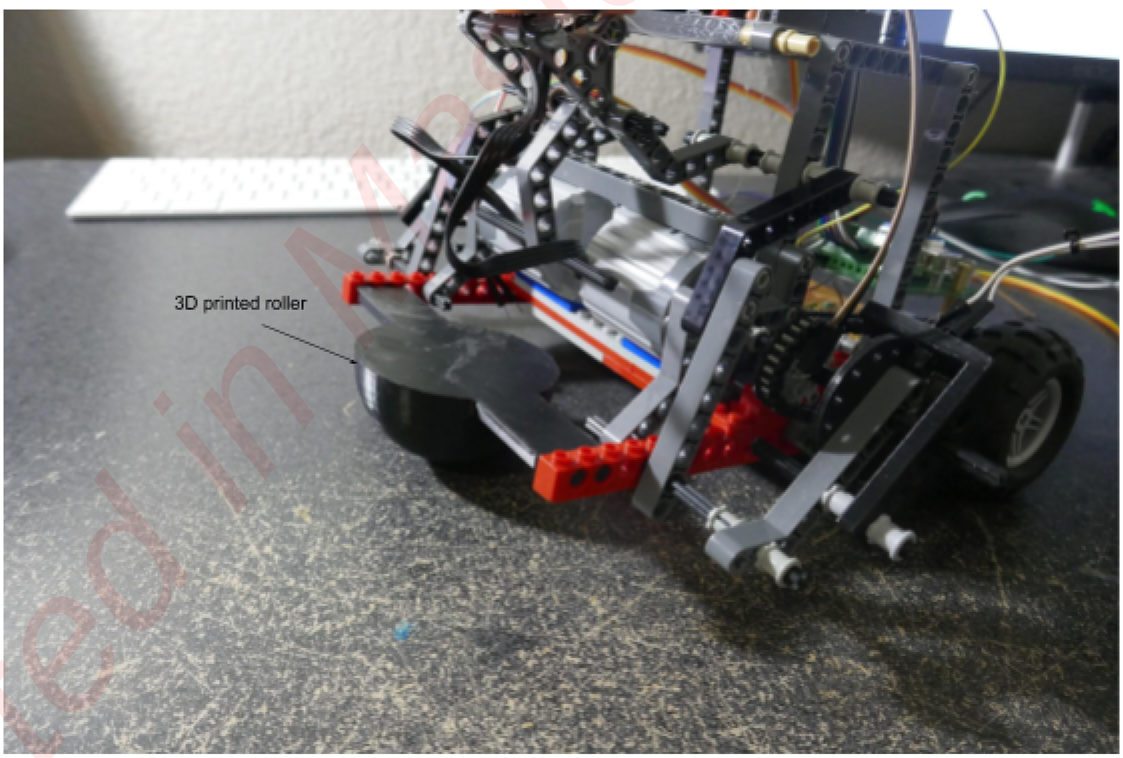
Re: Lab 4: Solving the Maze

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.




YETI


3D printed roller

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 ment 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.

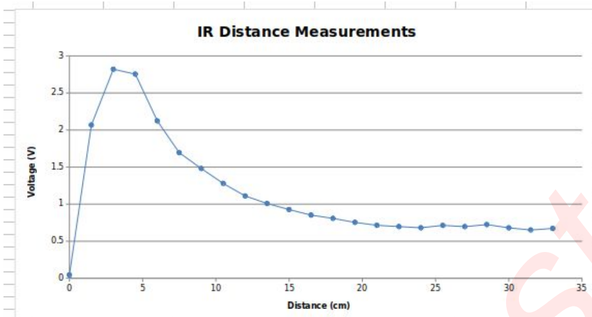
Optimization:

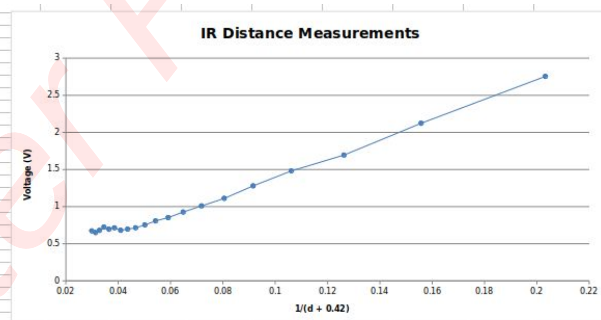
The approach algorithm explained above while simple to implement is highly inefficient in traversing the maze in an efficient way. So a post processing algorithm is run on the recorded moves to optimize the final path. The historic moves are iterated over in an attempt to locate local patterns in the 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.

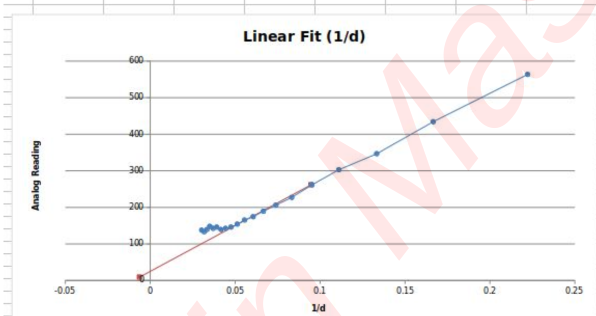
Results:

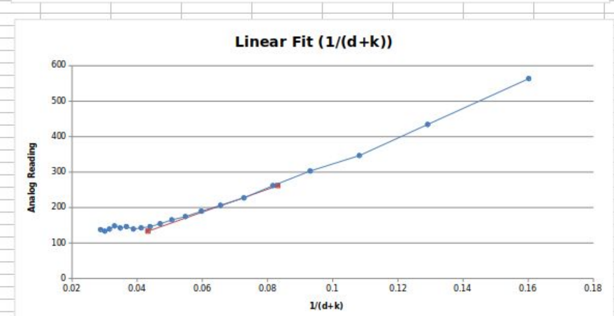
(Results all have an * by themas lab was finished at home with a simple maze construction acting as maze walls)

	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 mourning 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 standar 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 for full code documentation

```
/**
   @file custom_lab_4.ino
   @author Christian Prather
   @brief A basic feedback controlled system for an Arduino based robot with ultrasonic
          and IR distance sensors
   @version 0.1
   @date 2020-10-23

*/


/*! \mainpage Lab 4 Code Documentation
 *
 */

/// Libraries for interrupts and PID
#include <PinChangeInt.h>
#include <PID_v1.h>
#include <SR04.h>

/// Global Defines
/// Motor driver connections
#define IN1 5
#define IN2 6
#define IN3 7
#define IN4 8

/// Motor control
#define A 0
#define B 1
#define pwmA 3
#define dirA 9
#define pwmB 4
#define dirB 13

/// Start stop button
#define pushButton 2

/// Drive constants - dependent on robot configuration
#define EncoderCountsPerRev 12.0
#define DistancePerRev 51.0
#define DegreesPerRev 27.0

#define EncoderMotorLeft 7
#define EncoderMotorRight 8

/// Lab specific variables
double leftEncoderCount = 0;
double rightEncoderCount = 0;
int wallDist = 5; /// CM
#define DISTANCE_SEG 10

/// Enum defines
#define FORWARD 0
#define RIGHT 1
#define LEFT 2

/// IR sensors
int irSensor = A0;
```

```

/// Ultrasonic sensors
int trig = 12;
int echo = 11;
SR04 sideUS = SR04(trig, echo);

/// Default motor pwm values
int motorLeft_PWM = 180;
int motorRight_PWM = 200;

/// Time it takes to move 90 degrees
int milliSecondsPer90Deg = 900;

/// How many encoder counts for given distance
double desiredCount;

int movesCount = 0;
// Global array for tracking move order (move, distance) or (move, degree)
int moveList[50];

int optimizedMoves[50];

/**
   @brief PID values
   setpoints = desired counts, output = PWM, input = current counts
*/
double leftOutput, rightOutput;
PID leftPID(&leftEncoderCount, &leftOutput, &desiredCount, 2, 5, 2, DIRECT);
PID rightPID(&rightEncoderCount, &rightOutput, &desiredCount, 2, 5, 2, DIRECT);

/**
   @brief Helper function for setting the PWM back to default value
*/
void resetPWM()
{
    motorLeft_PWM = 180;
    motorRight_PWM = 200;
}

/**
   @brief Run the PID loop calculation and set out put to motors output in PWM

*/
void adjustPWM()
{
    // Compute the pid values
    leftPID.Compute();
    rightPID.Compute();

    // Set the pid values within range
    motorLeft_PWM = constrain(leftOutput, 150, 250);
    motorRight_PWM = constrain(rightOutput, 150, 235);
    Serial.print("Left PWM: ");
    Serial.print(motorLeft_PWM);
    Serial.print(" ");
    Serial.println(leftEncoderCount);
    Serial.print("Right PWM: ");
    Serial.print(motorRight_PWM);
```

```
    Serial.print(" ");
    Serial.println(rightEncoderCount);
}

/**
   @brief ISR for left encoder
*/
void indexLeftEncoderCount()
{
    leftEncoderCount++;
    //Serial.println("Left Encoder ++");
}

/**
   @brief ISR for incrementing right encoder
*/
void indexRightEncoderCount()
{
    rightEncoderCount++;
    //Serial.println("Right Encoder ++");
}

/**
   @brief Calculate how many encoder counts we expect given the distance provided
   based on the bot intrinsics

   @param distance
*/
void calculateDesiredCount(int distance)
{
    double revolutionsRequired = distance / DistancePerRev;

    desiredCount = revolutionsRequired * EncoderCountsPerRev;
    // Reset encoder counts
    leftEncoderCount = 0;
    rightEncoderCount = 0;
    Serial.print("Desired Count: ");
    Serial.println(desiredCount);
}

/**
 * @brief Calculate how many encoder counts we expect given the degrees provided
 *
 * @param degrees
 */
void calculateDesiredCountTurn(int degrees)
{
    double revolutionsRequired = degrees / DegreesPerRev;
    desiredCount = revolutionsRequired * EncoderCountsPerRev;
    leftEncoderCount = 0;
    rightEncoderCount = 0;
    Serial.print("Desired Count: ");
    Serial.println(desiredCount);
}

/**
   @brief Turn bot to given degrees

```

```
   @param degrees
*/
void turnRight(int degrees)
{
    resetPWM(); // Reset pwm
    calculateDesiredCountTurn(degrees);
    // While the encoders are not correct adjust PWM with PID loop
    // Loop unitl the encoders read correct

    while ((desiredCount - rightEncoderCount) > 3)
    {
        adjustPWM();
        //To drive forward, motors go in the same direction

        if ((desiredCount - leftEncoderCount) > 3)
        {
            run_motor(A, -motorLeft_PWM); //change PWM to your calibrations
        }
        if ((desiredCount - rightEncoderCount) > 3)
        {
            run_motor(B, motorRight_PWM); //change PWM to your calibrations
        }
    }

    // motors stop
    run_motor(A, 0);
    run_motor(B, 0);
    Serial.println("Done driving Right");
    Serial.print("L: ");
    Serial.println(leftEncoderCount);
    Serial.print("R: ");
    Serial.println(rightEncoderCount);
}

/**
   @brief Turn bot right to given degrees

   @param degrees
*/
void turnLeft(int degrees)
{
    resetPWM();
    calculateDesiredCountTurn(degrees);

    // Loop unitl the encoders read correct

    while ((desiredCount - leftEncoderCount) > 3)
    {
        adjustPWM();
        //To drive forward, motors go in the same direction

        if ((desiredCount - leftEncoderCount) > 3)
        {
            run_motor(A, motorLeft_PWM); //change PWM to your calibrations
        }
        if ((desiredCount - rightEncoderCount) > 3)
        {
            run_motor(B, -motorRight_PWM); //change PWM to your calibrations
        }
    }
```

```

    // motors stop
    run_motor(A, 0);
    run_motor(B, 0);
    Serial.println("Done driving Left");
    Serial.print("L: ");
    Serial.println(leftEncoderCount);
    Serial.print("R: ");
    Serial.println(rightEncoderCount);
}

/**
   @brief Function to drive bot forward until encoders are within range

   @param distance
*/
void driveForward(int distance)
{
    Serial.println("Driving Forward...");
    resetPWM();
    calculateDesiredCount(distance);

    // Loop unitl the encoders read correct

    while ((desiredCount - leftEncoderCount) > 3 || (desiredCount - rightEncoderCount) > 3)
    {
        adjustPWM();
        //To drive forward, motors go in the same direction

        if ((desiredCount - leftEncoderCount) > 3)
        {
            run_motor(A, -motorLeft_PWM); //change PWM to your calibrations
        }
        if ((desiredCount - rightEncoderCount) > 3)
        {
            run_motor(B, -motorRight_PWM); //change PWM to your calibrations
        }
    }

    // motors stop
    run_motor(A, 0);
    run_motor(B, 0);
    Serial.println("Done driving forward");
    Serial.print("L: ");
    Serial.println(leftEncoderCount);
    Serial.print("R: ");
    Serial.println(rightEncoderCount);
}

/**
   @brief Drive the bot backwards

   @param distance
*/
void driveBackward(int distance)
{
    resetPWM();
    calculateDesiredCount(distance);

```

```
    // Loop unitl the encoders read correct

    while ((desiredCount - leftEncoderCount) > 3 || (desiredCount - rightEncoderCount) > 3)
    {
        adjustPWM();
        //To drive backward, motors go in the same direction

        if ((desiredCount - leftEncoderCount) > 3)
        {
            run_motor(A, motorLeft_PWM); //change PWM to your calibrations
        }
        if ((desiredCount - rightEncoderCount) > 3)
        {
            run_motor(B, motorRight_PWM); //change PWM to your calibrations
        }
    }

    // motors stop
    run_motor(A, 0);
    run_motor(B, 0);
    Serial.println("Done driving backwards");
    Serial.print("L: ");
    Serial.println(leftEncoderCount);
    Serial.print("R: ");
    Serial.println(rightEncoderCount);
}

/**
 * @brief Function for reading the distance sensors
 *
 * @param sensor 0 = IR, 1 = Ultrasonic
 * @return float distance (cm)
 */
float readDistance(int sensor)
{
    float distance = 0.0;
    switch (sensor)
    {
    case 0:
        int reading = analogRead(irSensor);
        distance = ((0.00031) * reading) + 0.002;
        break;
    case 1:
        distance = sideUS.Distance();
        break;

    default:
        break;
    }
    return distance;
}

/**
 * @brief The exploritory function to allow the system to navigate unseen envioronment
 * Using left hand rule
 */
void explore()
{
```

```
    while (digitalRead(pushButton) == 1)
    {
        float front = readDistance(0);
        float side = readDistance(1);

        /// There is no wall to left of bot
        if (side > wallDist)
        {
            turnLeft(90);
            /// Not recording degrees as the assumption is every turn on 90 degrees
            moveList[movesCount] = "LEFT";
            movesCount++;
        }
        /// Can drive forward
        else if (front > wallDist)
        {
            driveForward(DISTANCE_SEG);
            moveList[movesCount] = "FORWARD";
            movesCount++;
        }
        /// Trapped turn Right
        else
        {
            turnRight(90);
            moveList[movesCount] = "RIGHT";
            movesCount++;
        }
    }
}

/**
 * @brief This is what youve all been waiting for one darn good looking
 * solution to maze optimation. Iterates over the movesList looking for
 * specific patterns it can reduce into simpler sequences
 * Key assumption: Explored using Left hand rule
 */
void optimize()
{
    /// Key patterns 0 = F, 1 = R, 2 = L, 3 = DELETE
    int keyPatterns_6[2][6] = {{0, 0, 1, 1, 0, 0}, {2, 0, 1, 1, 0, 2}};
    int keyPatterns_5[2][5] = {{2, 0, 1, 1, 0}, {0, 1, 1, 0, 2}};
    int keyPatterns_4[1][4] = {{0, 1, 1, 0}};

    int optimizedPattern_6[1][8] = {{FORWARD, 2 * DISTANCE_SEG, RIGHT, 90, RIGHT, 90, FORWARD, DISTANCE_SEG}};
    int optimizedPattern_5[2][2] = {{RIGHT, 90}, {RIGHT, 90}};
    int optimizedPatter_4[1][4] = {{LEFT, 90, LEFT, 90}};
    /** This is going to be checking in a priority tree fashion given highest priority
    * given highest priority patterns are 6 long then 5 long then 4 I can batch this
    */
    for (int i = 0; i < movesCount; i++)
    {
        /// Get next move in explored list
        // int move = moveList[i];
        /// Get next 6 moves if enough in list

        // Check 6 out first
```

```
        int future[6];
        for (int j = 0; j < 6; j++)
        {
            if ((j + i) < movesCount)
            {
                future[j] = moveList[j + i];
            }
        }
        int tracker = 0;
        for (auto potential : keyPatterns_6)
        {
            bool match = true;
            for (int m = 0; m < 6; m++)
            {
                if (future[m] != potential[m])
                {
                    match = false;
                }
            }
            if (match)
            {
                int keyPatternLength = (sizeof(potential) / sizeof(potential[0]));
                // Insert optimized move
                for (int x = 0; x < (sizeof(optimizedPattern_6[tracker]) / sizeof(optimizedPattern_6[tracker][0])); x++)
                {
                    if (optimizedPattern_6[tracker][x] != 3)
                    {
                        optimizedMoves[x] = optimizedPattern_6[tracker][x];
                    }
                }
                i = i + 6;
                break;
            }
            tracker = tracker + 1;
        }

 ///////////////////////////////////////////////////////////////////////////////
        // Check 5 out first
        int future_5[5];
        for (int j = 0; j < 5; j++)
        {
            if ((j + i) < movesCount)
            {
                future_5[j] = moveList[j + i];
            }
        }
        tracker = 0;
        for (auto potential : keyPatterns_6)
        {
            bool match = true;
            for (int m = 0; m < 5; m++)
            {
                if (future_5[m] != potential[m])
                {
                    match = false;
                }
```

```
            }
            if (match)
            {
                int keyPatternLength = (sizeof(potential) / sizeof(potential[0]));
                // Insert optimized move
                for (int x = 0; x < (sizeof(optimizedPattern_6[tracker]) / sizeof(optimizedPattern_6[tracker][0])); x++)
                {
                    if (optimizedPattern_6[tracker][x] != 3)
                    {
                        optimizedMoves[x] = optimizedPattern_6[tracker][x];
                    }
                }
                i = i + 5;
                break;
            }
            tracker = tracker + 1;
        }

 ///////////////////////////////////////////////////////////////////////////////
        // Check 4 out first
        int future_4[4];
        for (int j = 0; j < 4; j++)
        {
            if ((j + i) < movesCount)
            {
                future_4[j] = moveList[j + i];
            }
        }
        tracker = 0;
        for (auto potential : keyPatterns_6)
        {
            bool match = true;
            for (int m = 0; m < 4; m++)
            {
                if (future_4[m] != potential[m])
                {
                    match = false;
                }
            }
            if (match)
            {
                int keyPatternLength = (sizeof(potential) / sizeof(potential[0]));
                // Insert optimized move
                for (int x = 0; x < (sizeof(optimizedPattern_6[tracker]) / sizeof(optimizedPattern_6[tracker][0])); x++)
                {
                    if (optimizedPattern_6[tracker][x] != 3)
                    {
                        optimizedMoves[x] = optimizedPattern_6[tracker][x];
                    }
                }
                i = i + 4;
                break;
            }
            tracker = tracker + 1;
```

```
        }
    }
}


/**
   @brief Function for configuration of pin states and interrupts
*/
void configure()
{
    // set up the motor drive ports
    pinMode(pwmA, OUTPUT);
    pinMode(dirA, OUTPUT);
    pinMode(pwmB, OUTPUT);
    pinMode(dirB, OUTPUT);

    pinMode(pushButton, INPUT_PULLUP);

    pinMode(EncoderMotorLeft, INPUT_PULLUP); //set the pin to input
    PCintPort::attachInterrupt(EncoderMotorLeft, indexLeftEncoderCount,
CHANGE);

    pinMode(EncoderMotorRight, INPUT_PULLUP); //set the pin to input
    PCintPort::attachInterrupt(EncoderMotorRight, indexRightEncoderCount,
CHANGE);
}

/**
   @brief Default behavior when not driving, waits for the pushButton to
   be pressed so it can execute next command
   Blocking function
*/
void idle()
{
    Serial.println("Idle..");
    while (digitalRead(pushButton) == 1)
        ; // wait for button push
    while (digitalRead(pushButton) == 0)
        ;        // wait for button release
    delay(2000); // Give time to move hand
}

/**
   @brief Entry point of program handles serial setup and PID config
*/
void setup()
{
    Serial.begin(9600);
    Serial.println("Setting up.....");
    configure();
    leftPID.SetMode(AUTOMATIC);
    rightPID.SetMode(AUTOMATIC);
}

/**
   @brief This is the logic to execute if we hit a push button
   ideally this is never executed as we shoudl never actually hit the walls
*/
void react_left()
{
```

```
    // TODO: Check which button was hit

    driveBackward(20);
    turnRight(30);
}
void react_right()
{
    // TODO: Check which button was hit

    driveBackward(20);
    turnLeft(30);
}
void react_forward()
{
    // TODO: Check which button was hit
    driveBackward(50);
}

/**
   @brief Main drive execution of program, iterates through moves list
executing
   next move with corresponding distance or degrees
*/
void drive()
{
    // Iterate over the list jumping by two each time
    for (int i = 0; i < sizeof(optimizedMoves); i += 2)
    {
        idle();
        switch (moveList[i])
        {
        case LEFT:
            turnLeft(moveList[i + 1]);
            break;
        case RIGHT:
            turnRight(moveList[i + 1]);
            break;
        case FORWARD:
            driveForward(moveList[i + 1]);
            break;
        default:
            break;
        }
    }

}

/**
   @brief Loop execution of the program
*/
void loop()
{
    explore();
    optimize();
    drive();
}
```

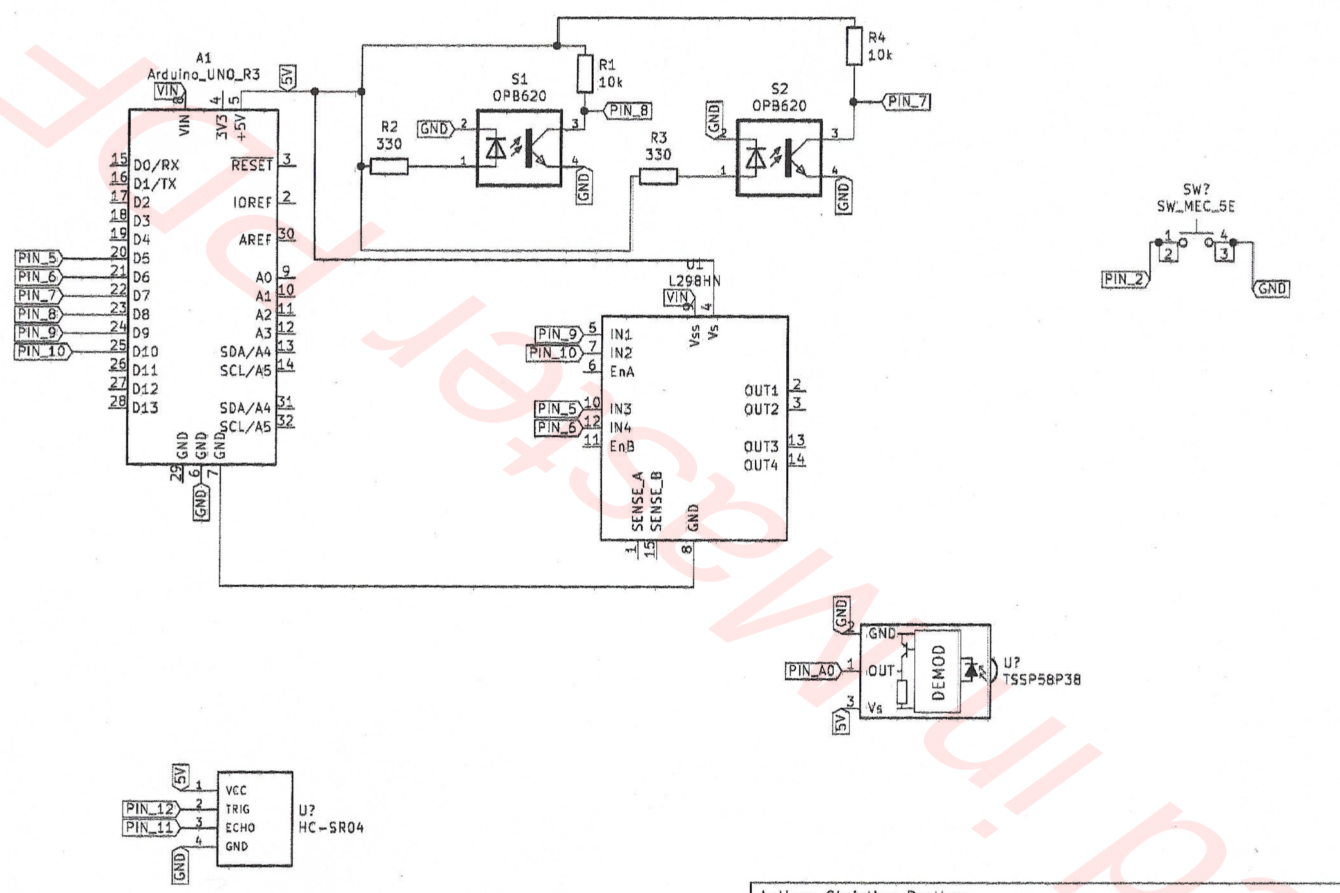
```
/// Basic calibration program for IR sensor
int irSensor = A1;
int pushButton = 2;

void setup()
{
    pinMode(pushButton, INPUT_PULLUP);
    Serial.begin(9600);
    Serial.println("Setup Complete..");
}

void loop()
{
    Serial.println("Waiting...");
    while (digitalRead(pushButton) == 1)
        ; // wait for button push
    while (digitalRead(pushButton) == 0)
        ; // wait for button release

    double averageValue = 0;
    for (int i = 0; i < 5; i++)
    {
          averageValue += analogRead(irSensor);
          delay(250);

    }
    averageValue = averageValue / 5;
    Serial.print("Sensor Reading: ");
    Serial.println(averageValue);
}

```

```cpp
/**
 * @file main.cpp
 * @author Christian Prather
 * @brief Testing algorithm for the optimization algorithm
 * @version 0.1
 * @date 2020-11-12
 *
 * @copyright Copyright (c) 2020
 *
 */
#include <iostream>
using namespace std;
/// Enum defines
#define FORWARD 0
#define RIGHT 1
#define LEFT 2
#define DISTANCE_SEG 10

int movesCount = 6;
// Global array for tracking move order (move, distance) or (move, degree)
int moveList[50] = {FORWARD, FORWARD, RIGHT, RIGHT, FORWARD, FORWARD};

int optimizedMoves[50];

void optimize()
{
    /// Key patterns 0 = F, 1 = R, 2 = L, 3 = DELETE
    int keyPatterns_6[2][6] = {{0, 0, 1, 1, 0, 0}, {2, 0, 1, 1, 0, 2}};
    int keyPatterns_5[2][5] = {{2, 0, 1, 1, 0}, {0, 1, 1, 0, 2}};
    int keyPatterns_4[1][4] = {{0, 1, 1, 0}};

    int optimizedPattern_6[1][8] = {{FORWARD, 2 * DISTANCE_SEG, RIGHT, 90, RIGHT, 90, FORWARD, DISTANCE_SEG}};
    int optimizedPattern_5[2][2] = {{RIGHT, 90}, {RIGHT, 90}};
    int optimizedPatter_4[1][4] = {{LEFT, 90, LEFT, 90}};
    /** This is going to be checking in a priority tree fashion given highest priority
    * given highest priority patterns are 6 long then 5 long then 4 I can batch this
    */

    for (int i = 0; i < movesCount; i++)
    {
        /// Get next move in explored list
        // int move = moveList[i];
        /// Get next 6 moves if enough in list

        // Check 6 out first
        int future[6];
        for (int j = 0; j < 6; j++)
        {
            if ((j + i) < movesCount)
            {
                /// j (0-5) i (0-movesCount)
                future[j] = moveList[j + i];
                cout << "Move: " << future[j] << endl;
            }
        }
        int tracker = 0;
        for (auto potential : keyPatterns_6)
```

```
        {
            bool match = true;
            for (int m = 0; m < 6; m++)
            {
                if (future[m] != potential[m])
                {
                    match = false;
                }
            }
            if (match)
            {
                cout << "Matched " << tracker << endl;
                int keyPatternLength = (sizeof(potential) / sizeof(potential[0]));
                // Insert optimized move
                for (int x = 0; x < (sizeof(optimizedPattern_6[tracker]) / sizeof(optimizedPattern_6[tracker][0])); x++)
                {
                    if (optimizedPattern_6[tracker][x] != 3)
                    {
                        optimizedMoves[x] = optimizedPattern_6[tracker][x];
                    }
                }
                i = i+ 6;
                break;
            }
            tracker = tracker + 1;
        }

 //////////////////////////////////////////////////////////////////////////////
        // Check 5 out first
        int future_5[5];
        for (int j = 0; j < 5; j++)
        {
            if ((j + i) < movesCount)
            {
                /// j (0-5) i (0-movesCount)
                future_5[j] = moveList[j + i];
                cout << "Move5: " << future_5[j] << endl;
            }
        }
        tracker = 0;
        for (auto potential : keyPatterns_6)
        {
            bool match = true;
            for (int m = 0; m < 5; m++)
            {
                if (future_5[m] != potential[m])
                {
                    match = false;
                }
            }
            if (match)
            {
                cout << "Matched " << tracker << endl;
                int keyPatternLength = (sizeof(potential) / sizeof(potential[0]));
                // Insert optimized move
```

```cpp
                for (int x = 0; x < (sizeof(optimizedPattern_6[tracker]) / sizeof(optimizedPattern_6[tracker][0])); x++)
                {
                    if (optimizedPattern_6[tracker][x] != 3)
                    {
                        optimizedMoves[x] = optimizedPattern_6[tracker][x];
                    }
                }
                i = i+ 5;
                break;
            }
            tracker = tracker + 1;
        }

        //////////////////////////////////////////////////////////////////////////////
        // Check 4 out first
        int future_4[4];
        for (int j = 0; j < 4; j++)
        {
            if ((j + i) < movesCount)
            {
                /// j (0-5) i (0-movesCount)
                future_4[j] = moveList[j + i];
                cout << "Move4: " << future_4[j] << endl;
            }
        }
        tracker = 0;
        for (auto potential : keyPatterns_6)
        {
            bool match = true;
            for (int m = 0; m < 4; m++)
            {
                if (future_4[m] != potential[m])
                {
                    match = false;
                }
            }
            if (match)
            {
                cout << "Matched " << tracker << endl;
                int keyPatternLength = (sizeof(potential) / sizeof(potential[0]));
                // Insert optimized move
                for (int x = 0; x < (sizeof(optimizedPattern_6[tracker]) / sizeof(optimizedPattern_6[tracker][0])); x++)
                {
                    if (optimizedPattern_6[tracker][x] != 3)
                    {
                        optimizedMoves[x] = optimizedPattern_6[tracker][x];
                    }
                }
                i = i+4;
                break;
            }
            tracker = tracker + 1;
        }
    }
}
```

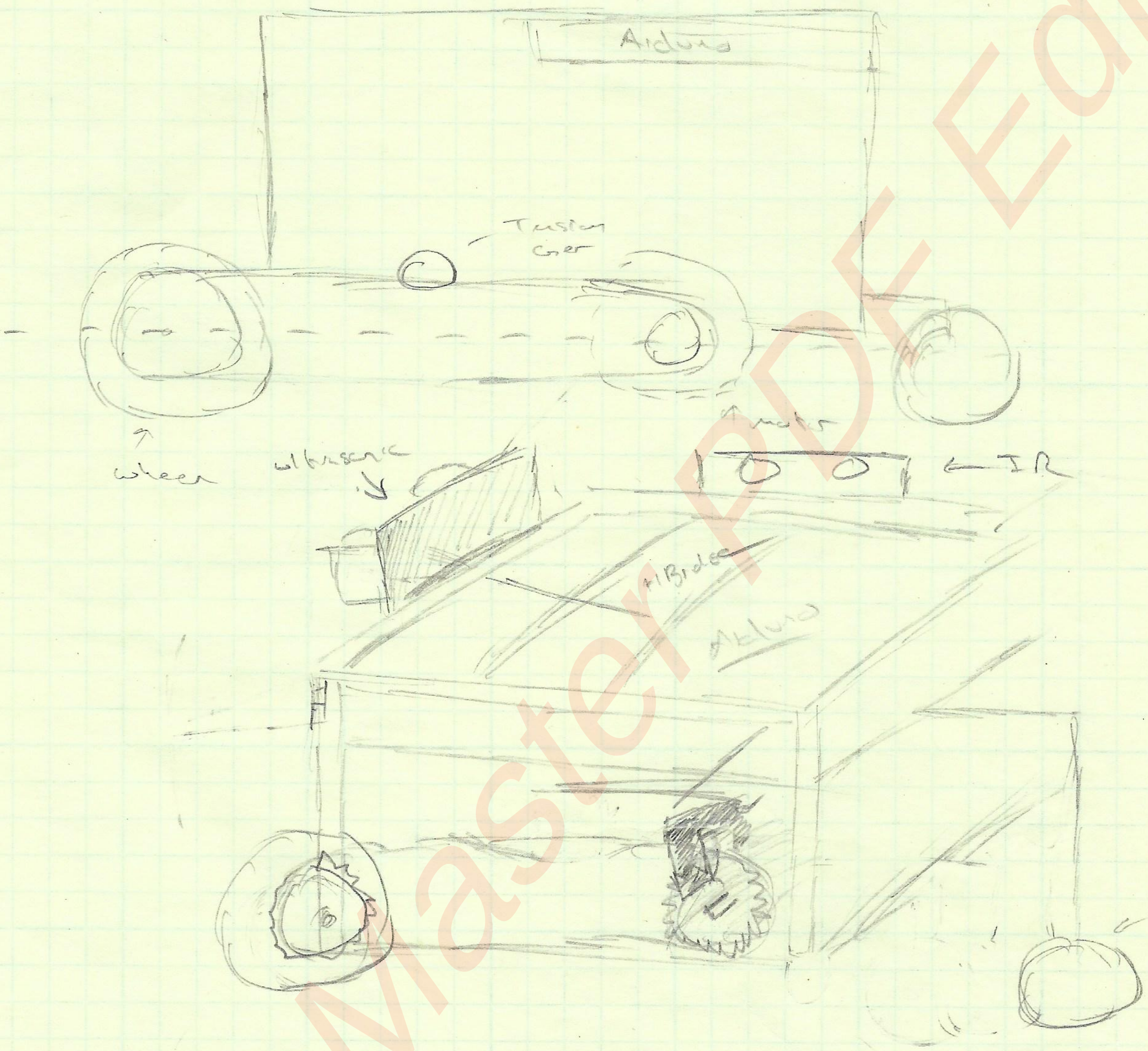
```cpp

int main()
{
    optimize();
    cout << "Optimized" << endl;
    for (auto move : optimizedMoves)
    {
        cout << move << ", ";
    }
}
```

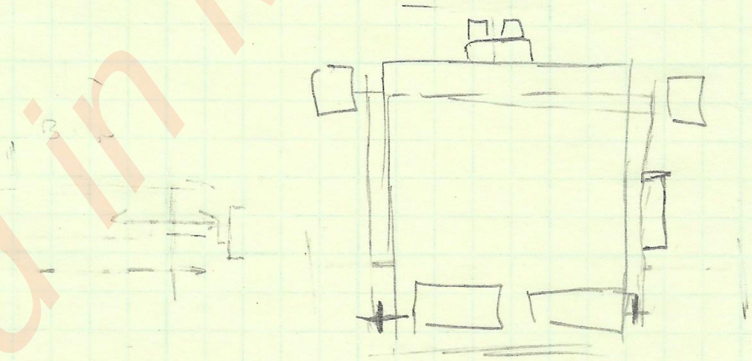


Author: Christian Prather

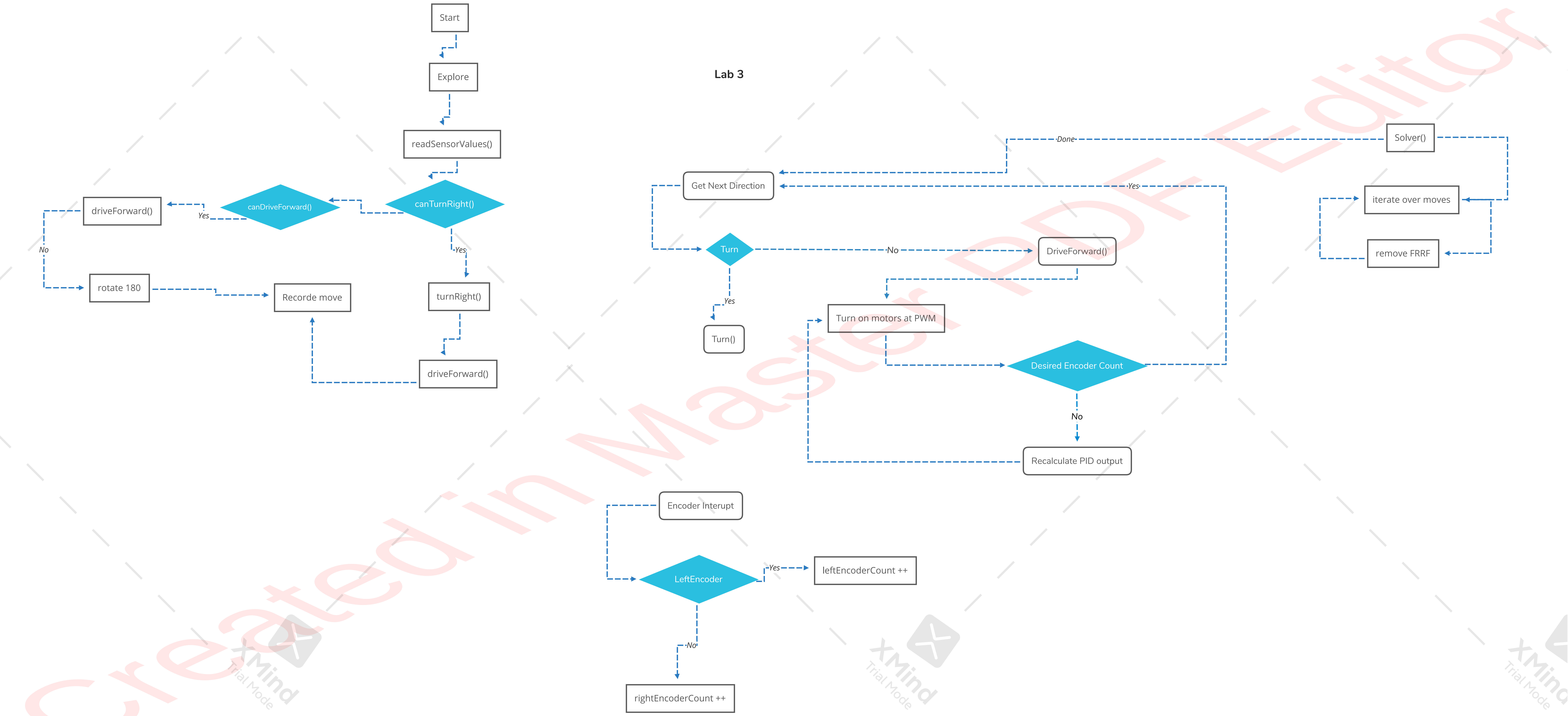
Sheet: /
File: robot_lab4.sch
Title: ~~Lab 3 Robot~~ LAb4
Size: A4 Date: 2020-10-19 **Rev: Version 3**
KiCad E.D.A. eeschema 5.1.5+dfsg1-2build2 Id: 1/1

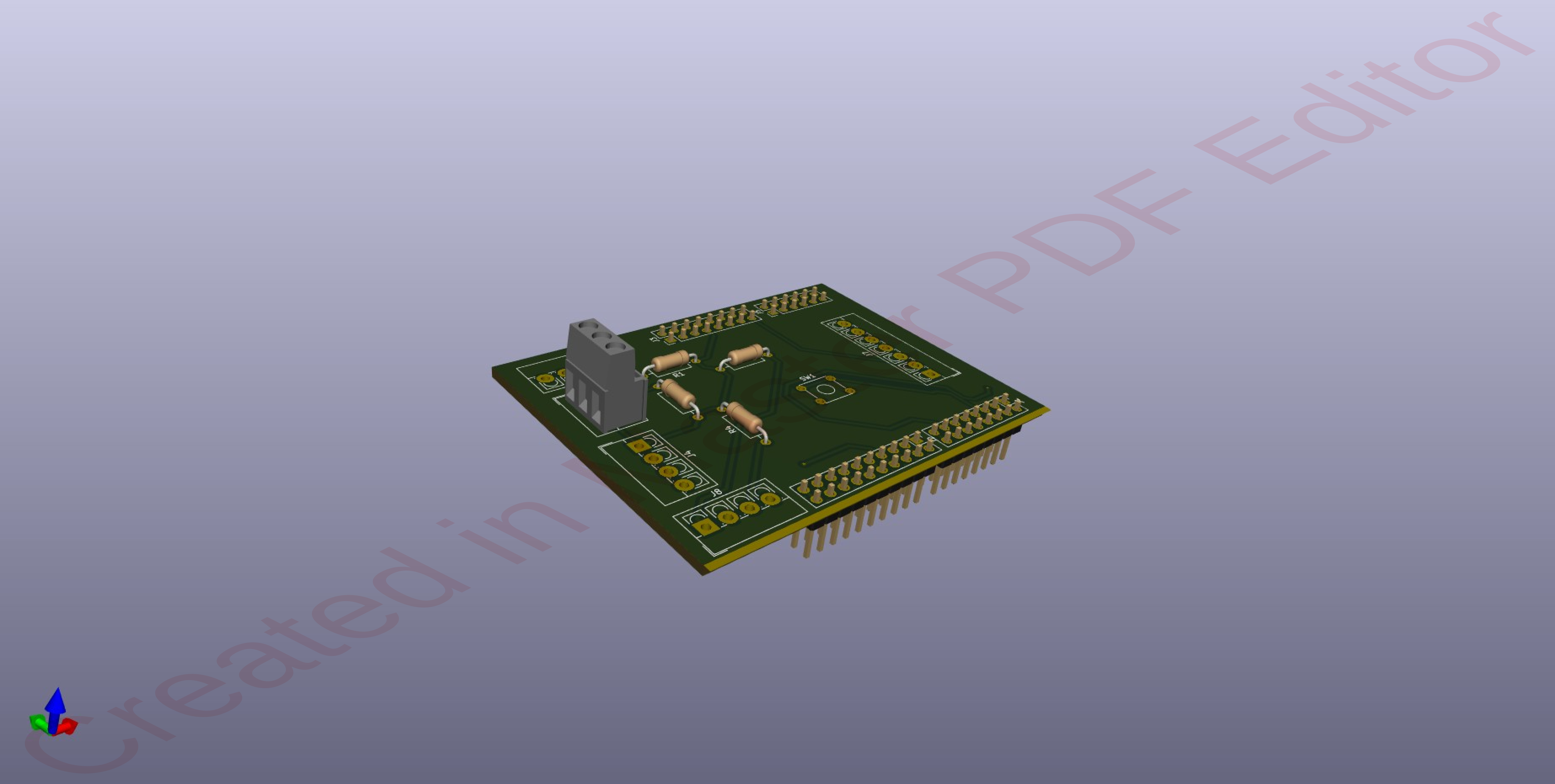
Mechanical

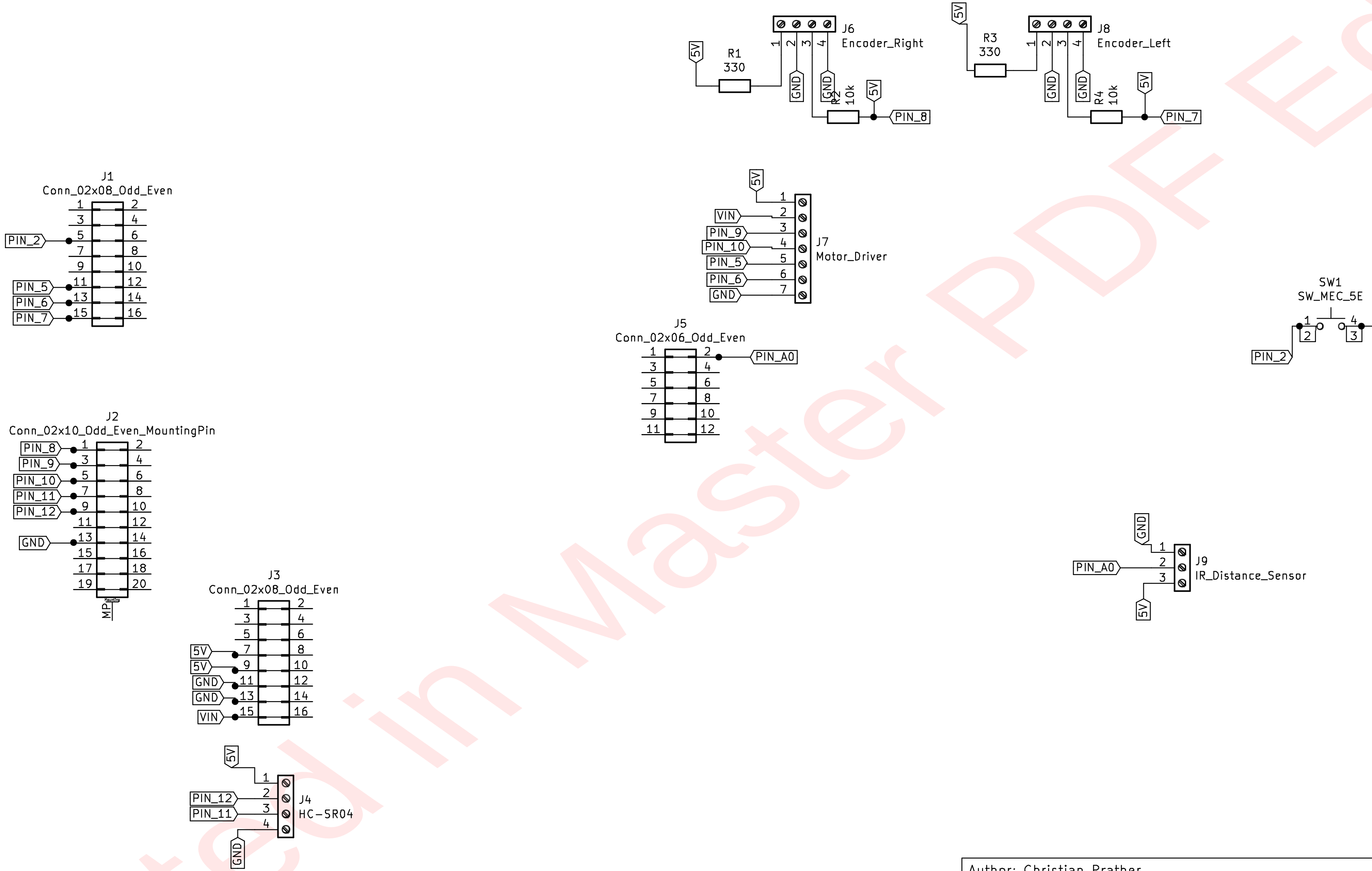




MS


Lab 3
Start
Explore
readSensorValues()
canTurnRight()
Yes
turnRight()
driveForward()
canDriveForward()
Yes
driveForward()
No
rotate 180
Recorde move
Solver()
Done
iterate over moves
remove FRRF
Get Next Direction
Yes
Turn
No
DriveForward()
Yes
Turn()
Turn on motors at PWM
Desired Encoder Count
No
Recalculate PID output
Encoder Interupt
LeftEncoder
Yes
leftEncoderCount ++
No
rightEncoderCount ++


SW1
R1



Author: Christian Prather

Sheet: /
File: robot_lab4.sch

Title: Lab 4 Robot

Size: A4	Date: 2020-10-19	**Rev: Version 3.1**
KiCad E.D.A. kicad 5.1.8-db983349187ubuntu20.04.1		Id: 1/1