```
/**
   @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;
```

```
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
    provided
142    based on the bot intrinsics
143 
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
    provided
160  *
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 
```

```
   @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
        }
    }
```

```
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 unitl 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
```

```
    // 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()
{
```

```
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 optimation. 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         // Check 5 out first
447         int future_5[5];
448         for (int j = 0; j < 5; j++)
449         {
450             if ((j + i) < movesCount)
451             {
452                 future_5[j] = moveList[j + i];
453             }
454         }
455         tracker = 0;
456         for (auto potential : keyPatterns_6)
457         {
458             bool match = true;
459             for (int m = 0; m < 5; m++)
460             {
461                 if (future_5[m] != potential[m])
462                 {
463                     match = false;
464                 }
```

```
            }
            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()
{
```

```
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 
```