Memo

To:	Gary Nave, Chukwuebuka Okwor, and Ramon Chavez
From:	Gordon Dina and Coleson Oliver
Team #:	N-423
Date:	09 March 2023
Re:	Lab #3: Robot Navigation in the Maze

Problem Statement

In this lab, our task was to modify our robot to be capable of navigating a given maze in closed loop control. This involved adding break-beam sensors as motor encoders to provide the data on which the feedback loop would act and adding software capability for the robot to use this new data and intrinsic timing data already available in order to drive precisely (or as precisely as possible) through the maze.

Figure 1: A top-down view of the maze

Methods

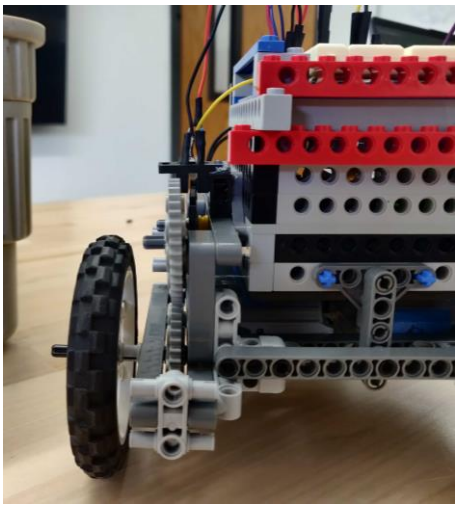


Figure 2: Front view of the motor encoder setup

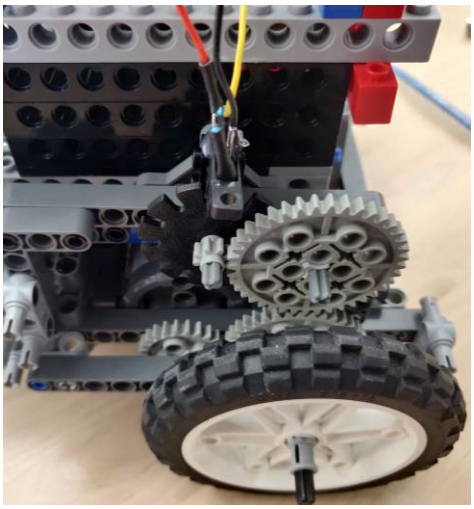


Figure 3: top-side view of the motor encoder setup

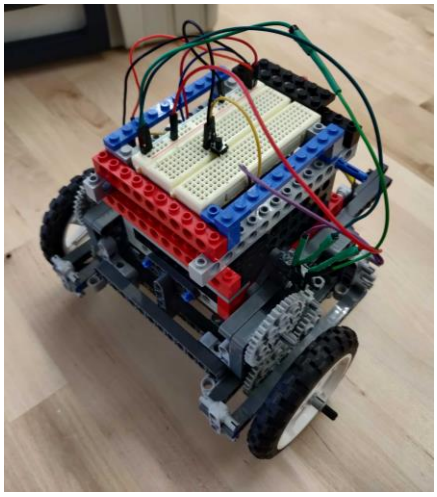


Figure 4: View of the robot

We configured our sensors as shown in Figures 2 and 3, with a 1:5 gear ratio between the wheels and the encoders to increase the spatial resolution for greater accuracy [Nave 2023] (more ticks per revolution). We also used a button as shown in Figure 3 to start each run of the maze.

In terms of the algorithm used to traverse the maze, the path itself was pre-programmed in the robot's memory in terms of drive distances in centimeters and turn angles in degrees. Using the diameter of the wheels and the distance between them, we were able to calculate values for distance per revolution and degrees per revolution. Combining this with our encoder ticks per revolution we were then able to calculate the number of encoder ticks to travel in both motors to actualize a given command. With this error and a derived change in error and change in time updated each loop in the controller we were able to implement a functional PD controller that was mostly able to correct for the difference in performance of the motors to make the robot drive straight. We also utilized data taken in Lab 2 to inform the decision of the maximum and minimum acceptable PWM values so that if the controller saturated to either the high or low extreme, the robot would still be driving relatively straight.

Results

Table 1. Robot Timing and Performance in the Maze

Trial	Maze Tile Reached	Time Elapsed	Manual Interventions
1	10	26.57	1
2	10	27.62	2
3	10	26.76	1

One of the most important things we noticed about the positional performance of the robot in terms of the controller gains was that even though the pure proportional controller would travel the correct forward distance overall, the difference in performance would cause one motor to reach the target first therefore causing a “dog leg” maneuver where the robot would drift sideways in the direction of the underperforming motor. This was very problematic and was the reason we needed to add derivative control. We did not encounter significant steady-state error once we correctly set the minimum PWM for the motors, but various inconsistencies resulted in accumulation of random errors that affected the accuracy more and more as each test progressed. This could be solved with a better overall design, in specific a smaller robot that turned about its center so the risk of running into walls during a turn would be minimized, and perhaps a lower gear ratio between the motors and the wheels/encoders so that the robot would approach set points slower and therefore be less likely to drift under its own momentum when it reaches target points.

Conclusions

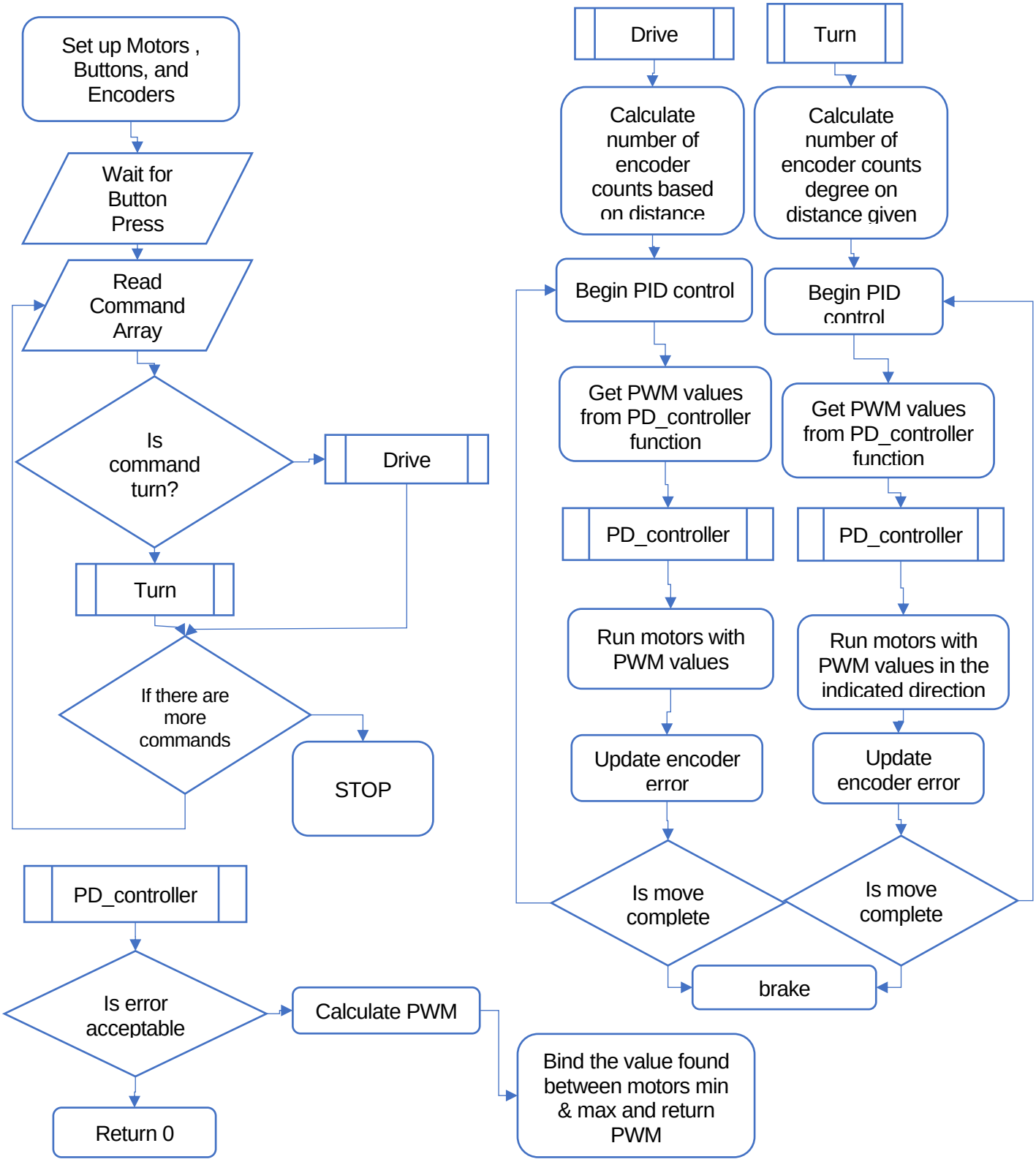
Overall, our robot finally ended up performing somewhat reliably after much parameter tuning. For us this was the biggest surprise as the previous theoretical applications of closed loop control indicate with the proper variables accounted for in the design the robot would drive very precisely. Instead, the process revolved much more around laying the correct ground work and then doing a lot of guess and check to dial in the performance in terms of the various gains and other parameters. In this sense we did well, laying solid software groundwork with plenty of options to tune specific parameters to address specific incorrect robot behaviors, but in the same way it took us so much time to address each individual failure of the robot that we ended up working well outside of lab time just to get it to complete the maze and even still it required outside intervention. For next time, the most meaningful change to make would be to make a smaller robot that turns about its center as discussed previously, since this would greatly reduce the overall precision required to not run into a wall halfway through and ruin an entire test run.

In terms of closed loop control overall, the advantage is that it is demonstrably more accurate than closed loop control. However, only relying on data intrinsic to the robot itself still presents opportunities for error, where the robot can drift or just simply impact an obstacle without ever knowing. In addition, closed loop control is much more difficult to implement and tune than open-loop control but the drastic increase in accuracy is worth that cost for most applications.

References

Gary Nave. "PID Control and Encoders". Colorado School of Mines. Retrieved 08 March 2023

Appendices:


Set up Motors , Buttons, and Encoders
Wait for Button Press
Read Command Array
Is command turn?
Drive
Turn
If there are more commands
STOP
PD_controller
Is error acceptable
Calculate PWM
Return 0
Bind the value found between motors min & max and return PWM
Drive
Turn
Calculate number of encoder counts based on distance
Calculate number of encoder counts degree on distance given
Begin PID control
Begin PID control
Get PWM values from PD_controller function
Get PWM values from PD_controller function
PD_controller
PD_controller
Run motors with PWM values
Run motors with PWM values in the indicated direction
Update encoder error
Update encoder error
Is move complete
Is move complete
brake

```
/* a closed loop proportional control
   gkn 20230209
*/

// Include Libraries
// This is the PinChangeInterrupt package by NicoHood, available in the library search
#include <PinChangeInterrupt.h>

// Driver definitions

// If you have a kit with the moto shield, set this to true
// If you have the Dual H-Bridge controller w/o the shield, set to false
#define SHIELD true

//SHIELD Pin varables - cannot be changed
#define motorApwm 3
#define motorAdir 12
#define motorBpwm 11
#define motorBdir 13

//Driver Pin variable - any 4 analog pins (marked with ~ on your board)
#define IN1 9
#define IN2 10
#define IN3 5
#define IN4 6

// Lab Specific definitions

// Defining these allows us to use letters in place of binary when
// controlling our motors
#define A 0
#define B 1
#define pushButton 10 // install a Pullup button with its output into Pin 2

// You may want to define pins for buttons as bump sensors. Pay attention to other
used pins.


// The digital pins we'll use for the encoders
#define EncoderMotorLeft  8
#define EncoderMotorRight 7

// Initialize encoder counts to 0
// These are volatile because they change during interrupt functions
```

```
volatile unsigned int leftEncoderCount = 0;
volatile unsigned int rightEncoderCount = 0;

//These are to build your moves array, a la Lab 2
#define FORWARD             30.48
#define LEFT                1
#define RIGHT              -1

// CONSTANTS TO MODIFY IN THIS LAB

// Drive constants - dependent on robot configuration
#define EncoderCountsPerRev 120.0
#define   DistancePerRev          25.5   //   TODO:   Measure   wheel   diameter   (8.1
cm)         (Calculated Value 25.44)
#define DegreesPerRev        165 // TODO: Measure distance between wheels (18.8 cm)
(Calculated Value 155.58)

// Proportional Control constants
// what are your ratios of PWM:Encoder Count error?
#define kP_A 1.8
#define kP_B 2.0

// Derivative Control constants
#define kD_A 5.0
#define kD_B 1.0

// how many encoder counts from your goal are acceptable?
#define distTolerance 3

// Deadband power settings
// The min PWM required for your robot's wheels to still move
// May be different for each motor
#define minPWM_A 75
#define minPWM_B 75

#define maxPWM_A 202
#define maxPWM_B 230

// Lab specific variables (in cm)
// Fill in this array will forward distances and turn directions in the maze (a la Lab
2)
int  moves[]  =  {FORWARD,  LEFT,  FORWARD,  LEFT,  FORWARD,  RIGHT,  FORWARD*3,  RIGHT,
FORWARD*2, RIGHT, FORWARD};
```

```
void setup() {
  // set stuff up
  Serial.begin(9600);
  motor_setup();
  pinMode(pushButton, INPUT_PULLUP);

  // add additional pinMode statements for any bump sensors


  // Attaching Wheel Encoder Interrupts
  Serial.print("Encoder Testing Program\n");
  Serial.print("Now setting up the Left Encoder: Pin ");
  Serial.print(EncoderMotorLeft);
  Serial.println();
  pinMode(EncoderMotorLeft, INPUT_PULLUP); //set the pin to input

  // The following code sets up the PinChange Interrupt
  // Valid interrupt modes are: RISING, FALLING or CHANGE
          attachPinChangeInterrupt(digitalPinToPinChangeInterrupt(EncoderMotorLeft),
indexLeftEncoderCount, CHANGE);
  // if you "really" want to know what's going on read the PinChange.h file :)
  ////////////////////////////////////////////////
  Serial.print("Now setting up the Right Encoder: Pin ");
  Serial.print(EncoderMotorRight);
  Serial.println();
  pinMode(EncoderMotorRight, INPUT_PULLUP);     //set the pin to input
         attachPinChangeInterrupt(digitalPinToPinChangeInterrupt(EncoderMotorRight),
indexRightEncoderCount, CHANGE);
} /////////////// end of setup ////////////////////////////////////

//////////////////////// loop() ////////////////////////////////////
void loop()
{

  while (digitalRead(pushButton) == 1); // wait for button push
  while (digitalRead(pushButton) == 0); // wait for button release

  Serial.println("Button Pressed");

  for (int i = 0; i < sizeof(moves)/2; i++) { // Loop through entire moves list
    if(moves[i]==LEFT){
      // Fill with code to turn left
      turn(LEFT, 90);
    }
```

```
    else if(moves[i]==RIGHT){
      // Fill with code to turn right
      turn(RIGHT, 90);
    }
    else{
      // Fill with code to drive forward
      drive(moves[i]);
    }
    // TODO: use SW break from Lab 2
    run_motor(A, 0);
    run_motor(B, 0);
    delay(1000);
  }
}
////////////////////////////////// end of loop() ///////////////////////////////////


//////////////////////////////////////////////////////////////////////////////////
int drive(float distance)
{
  // create variables needed for this function
  int countsDesired, cmdLeft, cmdRight, xerr_L, xerr_R, dxerr_L, dxerr_R;
  float dt;

  // Find the number of encoder counts based on the distance given, and the
  // configuration of your encoders and wheels
  countsDesired = distance / DistancePerRev * EncoderCountsPerRev;

  // reset the current encoder counts
  leftEncoderCount = 0;
  rightEncoderCount = 0;

  // we make the errors greater than our tolerance so our first test gets us into the
loop
  xerr_L = distTolerance + 1;
  xerr_R = distTolerance + 1;

  dxerr_L = 0;
  dxerr_R = 0;

  unsigned long t0 = millis();

  // Begin PID control until move is complete
  while (xerr_L > distTolerance || xerr_R > distTolerance)
```

```
  {
    dt = float(millis() - t0);
    t0 = millis();

    // Get PWM values from the controller function
    cmdLeft  = PD_Controller(kP_A, kD_A, xerr_L, dxerr_L, dt, minPWM_A, maxPWM_A);
    cmdRight = PD_Controller(kP_B, kD_B, xerr_R, dxerr_L, dt, minPWM_B, maxPWM_B);

    // Run motors with calculated PWM values
    run_motor(A, cmdLeft);
    run_motor(B, cmdRight);

    // Update encoder error
    // Error is the number of encoder counts between here and the destination
    dxerr_L = (countsDesired - leftEncoderCount ) - xerr_L;
    dxerr_R = (countsDesired - rightEncoderCount) - xerr_R;

    xerr_L = countsDesired - leftEncoderCount;
    xerr_R = countsDesired - rightEncoderCount;

    // Some print statements, for debugging
    Serial.print(xerr_L);
    Serial.print(" ");
    Serial.print(dxerr_L);
    Serial.print(" ");
    Serial.print(cmdLeft);
    Serial.print("\t");
    Serial.print(xerr_R);
    Serial.print(" ");
    Serial.print(dxerr_R);
    Serial.print(" ");
    Serial.println(cmdRight);
  }

}
////////////////////////////////////////////////////////////////////////////////


// Write a function for turning with PID control, similar to the drive function
int turn(int direction, float degrees)
{
  // create variables needed for this function
  int countsDesired, cmdLeft, cmdRight, xerr_L, xerr_R, dxerr_L, dxerr_R;
  float dt;
```

```
  // Find the number of encoder counts based on the distance given, and the
  // configuration of your encoders and wheels
  countsDesired = degrees / DegreesPerRev * EncoderCountsPerRev;

  // reset the current encoder counts
  leftEncoderCount = 0;
  rightEncoderCount = 0;

  // we make the errors greater than our tolerance so our first test gets us into the
loop
  xerr_L = distTolerance + 1;
  xerr_R = distTolerance + 1;

  dxerr_L = 0;
  dxerr_R = 0;

  unsigned long t0 = millis();

  // Begin PID control until move is complete
  while (xerr_L > distTolerance || xerr_R > distTolerance)
  {
    dt = float(millis() - t0);
    t0 = millis();

    // Get PWM values from controller function
    cmdLeft  = PD_Controller(kP_A, kD_A, xerr_L, dxerr_L, dt, minPWM_A, maxPWM_A);
    cmdRight = PD_Controller(kP_B, kD_B, xerr_R, dxerr_L, dt, minPWM_B, maxPWM_B);

    // Run motors with calculated PWM values in the indicated direction
    run_motor(A, direction * cmdLeft);
    run_motor(B, -direction * cmdRight);

    // Update encoder error
    // Error is the number of encoder counts between here and the destination
    dxerr_L = (countsDesired - leftEncoderCount ) - xerr_L;
    dxerr_R = (countsDesired - rightEncoderCount) - xerr_R;

    xerr_L = countsDesired - leftEncoderCount;
    xerr_R = countsDesired - rightEncoderCount;

    // Some print statements, for debugging
    Serial.print(xerr_L);
    Serial.print(" ");
```

```
    Serial.print(dxerr_L);
    Serial.print(" ");
    Serial.print(cmdLeft);
    Serial.print("\t");
    Serial.print(xerr_R);
    Serial.print(" ");
    Serial.print(dxerr_R);
    Serial.print(" ");
    Serial.println(cmdRight);
  }

}

//////////////////////////////////////////////////////////


// If you want, write a function to correct your path due
// to an active bump sensor. You will want to compensate
// for any wheel encoder counts that happend during this maneuver


//////////////////////////////////////////////////////////
int PD_Controller(float kP, float kD, int xerr, int dxerr, float dt, int minPWM, int
maxPWM)
//  gain, deadband, and error, both are integer values
{
  if (xerr <= distTolerance) { // if error is acceptable, PWM = 0
    return (0);
  }

  // Proportional and Derivative control
  int pwm = int((kP * xerr) + (kD * (dxerr/dt)));
  pwm = constrain(pwm,minPWM,maxPWM); // Bind value between motor's min and max
  return(pwm);
}


//////////////////////////////////////////////////////////

// These are the encoder interupt funcitons, they should NOT be edited

void indexLeftEncoderCount()
{
  leftEncoderCount++;
```

```
}
//////////////////////////////////////////////////////
void indexRightEncoderCount()
{
  rightEncoderCount++;
}
```