ENEE408I
Team 4

Tin Nguyen, Emily Shin, Nadiv Panitch

Instructor: Professor G.L. Blankenship
Teaching Assistant: Khoa Pham khoaphamq@gmail.com
Course Designer: Levi Burner lburner@umd.edu

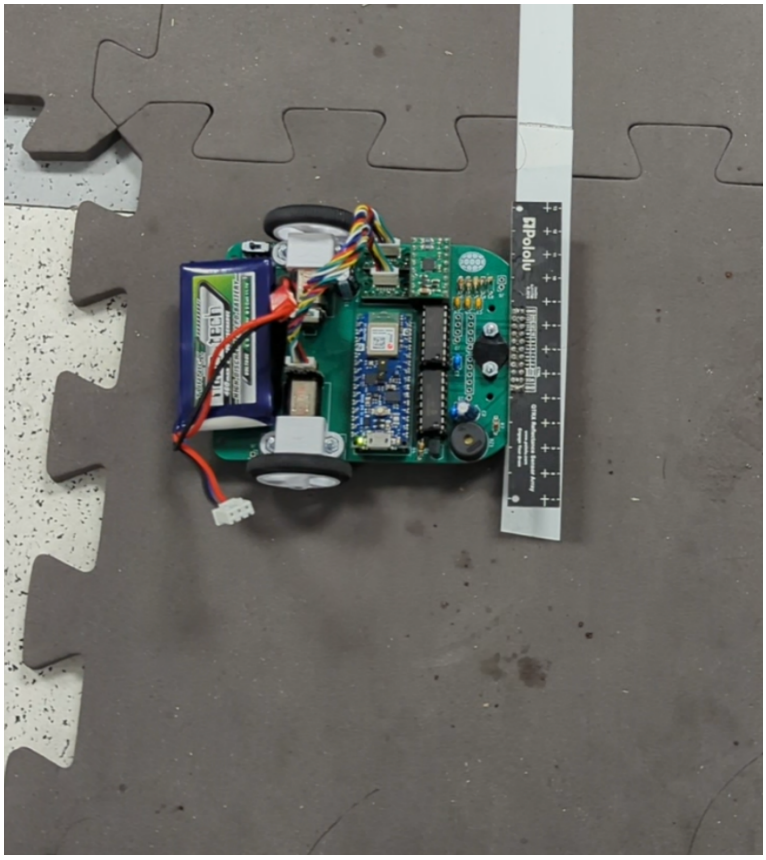


Table Of Contents

Introduction:

Goal:

The primary goal of this project was to create an autonomous robot that could solve a maze. The robots are referred to as “mice” because of their tiny size. One of the extra goals we added was to have three mice working together at the same time. We also intended for the robot to solve the maze and find the shortest possible path.

Overview:

Over the duration of the course, we tried to build an autonomous robot that can solve a maze. The work was split up amongst the three of us. One person worked on the movement and line detection, another person worked on the bluetooth and gyroscope, and the last person worked on the navigation and shortest path algorithm.

We started out working on the infrared sensors because the first priority was to be able to detect the lines of the maze. Once that part was completed, we moved on to getting the movement of the mouse to work. The base idea of our first movement implementation was horrible though. We had to rewrite the entire base code for the movement near the end of the semester, which we will explain later.

First, we want to explain the mistake of our first implementation so you don’t make the same mistake as we did. We had one wheel spinning at constant velocity, while the other wheel copied the rotation of the first wheel. This will ensure that both wheels are spinning with the same rotation. This became a problem when we tried to make a turn because the first wheel, right wheel, will always spin faster than the left wheel at the start of the turn because the left wheel takes time to match up the rotation of the right wheel. This caused the right turn to always offshoot. This was later fixed by rewriting the entire code to control both wheels at the same time, which we will go over in more detail in the troubleshooting section.

While we were working on the movement code, other members were working on the bluetooth and navigation. We don’t want to boggle you down too much in the overview, so we will discuss how we implemented these in more detail in the later sections.

Materials:

- **Arduino BLE Sense 33:**
  This arduino runs at 16MHz and has a built-in gyroscope, accelerometer, and bluetooth module

- **Pre-built Mouse PCB.** The mouse consist of:
  - **Pololu QTRX MD 13A:**
    This is a row of 13 infrared sensors placed on a horizontal bar. It's used to detect the white line since IR reflects off light surfaces (the line) and gets absorbed by dark surfaces.
  - **2 x 3:8 MUX:**
    These are connected the 13 IR sensors to the Arduino since the Arduino has a limited amount of pins. They communicate using SPI protocol, allowing the control of the 13 sensors using only 5 pins from the Arduino.
  - **Buzzer:**
    A low quality mini speaker for playing 8-bit sound from the Arduino
  - **DRV8833 Dual Motor Driver Carrier:**
    Dual H-bridge module for controlling motor's polarity and power
  - **Pololu Metal Gear-box Motors:**
    Motors with built-in encoders for measuring angle of rotation
  - **Wheels:**
    Thin plastic wheels with rubber tires

- **NVIDIA Jetson:**
  This works like a raspberry pi. It's basically a tiny computer that runs on Linux OS. It has wifi and bluetooth capabilities and is used for communicating with the mouse wirelessly.

Github and Video Links:

Most of our codes are uploaded to Github in the link below:
UMD-ENEE408I/ENEE408I_FALL2021_Team4 (github.com)

Video of Navigation Code Presentation:
https://umd.zoom.us/rec/share/QyU6sV4yU0Lv9jNgcKQKd97m1vHPmKoRVhW0nRbfsf9SwcI8LTUwHg7s-Swy8iZW.7wzTUJ_pl8EEWNx4?startTime=1640061513000

Video of our Robot Solving Maze using Left Turn Algorithm:
https://youtu.be/gQOTQK6BXuw

Other Notes:
https://drive.google.com/drive/folders/1eTbA_SJ6lOWi5oeGtkAMrkHcoSO1wVfw?usp=sharing

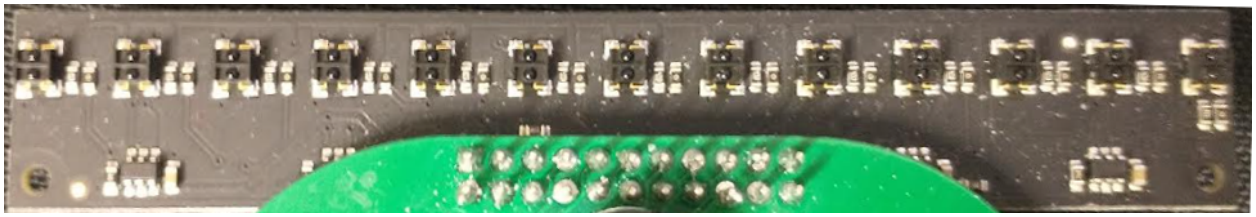
Code and Component Description:

This section will explain the breakdown and fundamentals of our code and how they interact with our hardwares.

IR Sensors and Line Reading (Hardware)

The first function that was worked on was the IR sensors. It's one of the most important functions that the mouse has because no matter how good the movement code is, if the mouse can't see the line it's following then every other movement code is useless.

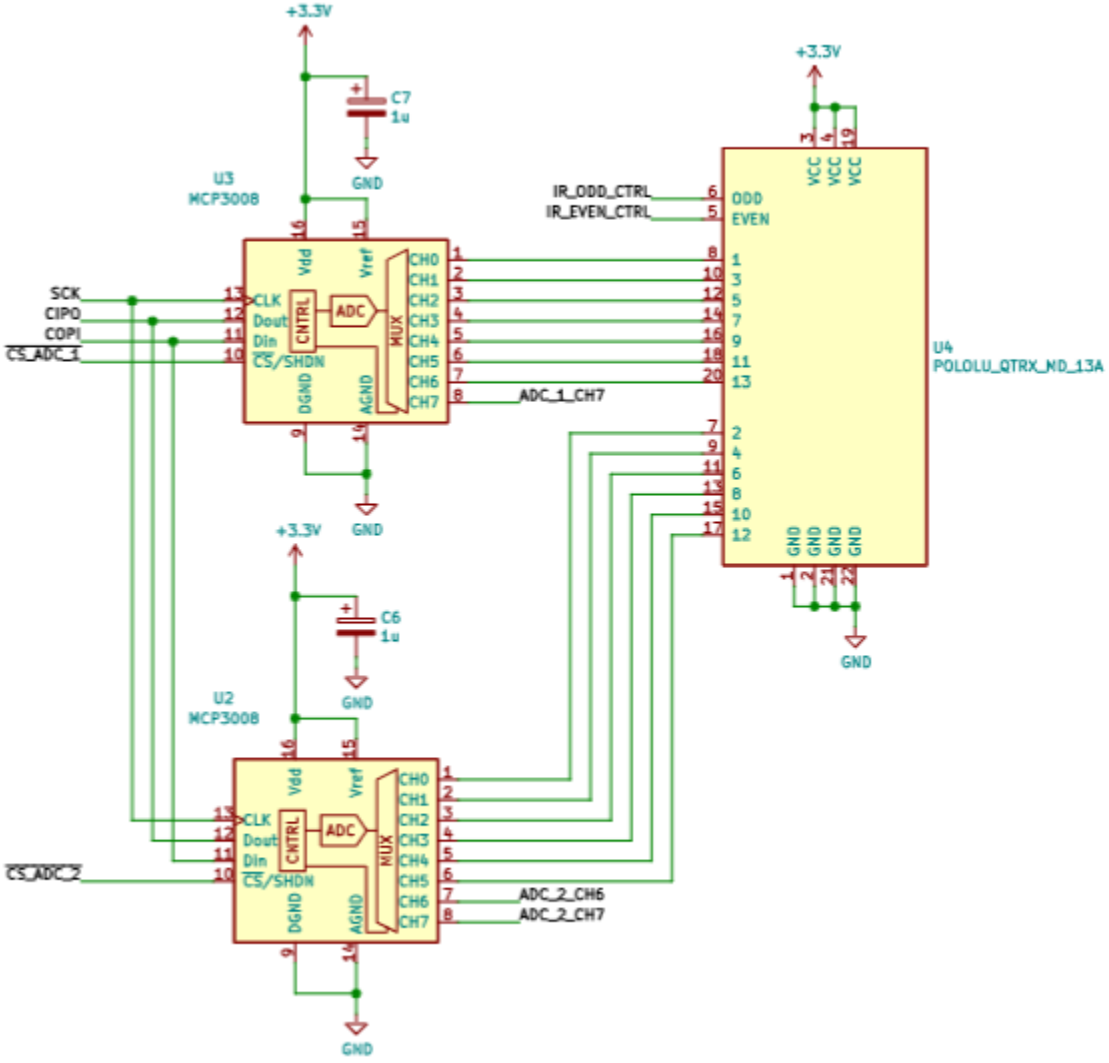
The sensor bar we used was the Pololu QTRX MD 13A. It has 13 individual IR sensors attached to it.



Each sensor has its own led that emits infrared light. These light hits the surface and bounce back to the sensor. The sensor is a phototransistor (PNP), so more light hitting the base will result in a lower emitter-collector current, giving a low ADC value. Black will do the opposite and block the light from reflecting, resulting in a high ADC value. The light emitted by the sensor bar can be seen through your phone's camera in the dark as shown below.

Now the problem is that there are 13 sensors to be read in total. The Arduino only has 7 analog pins. How do you read all 13 sensor values? Well, our lab designer added two (3:8)MUX modules to switch between the sensors. The MUX has a built-in SPI protocol controller, which allows the control of all 16 MUX pins using only 3 input pins from the

Arduino: SCK, COPI, CIPO. Those three pins are digital pins, so it saves us some analog pins to be used for other devices.



The CS_ADC pins are the output of the MUX -- it's the pin that has the sensor reading.

In summary, the hardware for the IR sensor is controlled using SPI protocol, which only needs three pins. These three pins allow us to cycle through the 13 sensor diodes on the sensor bar and read them.

IR Sensors and Line Reading (Software)

The code for the IR sensor is quite straightforward. The object **adc1** refers to the reading from the first MUX and **adc2** refers to the reading from the second MUX. We just hard-coded these and stored them into a buffer array of size 13. Something similar to the code below:

```
void read_line(){
    int adc_buf[13];

    adc_buf[12] = adc1.readADC(0);  adc_buf[11] = adc2.readADC(0);
    adc_buf[10] = adc1.readADC(1);  adc_buf[9]  = adc2.readADC(1);
    adc_buf[8]  = adc1.readADC(2);  adc_buf[7]  = adc2.readADC(2);
    adc_buf[6]  = adc1.readADC(3);  adc_buf[5]  = adc2.readADC(3);
    adc_buf[4]  = adc1.readADC(4);  adc_buf[3]  = adc2.readADC(4);
    adc_buf[2]  = adc1.readADC(5);  adc_buf[1]  = adc2.readADC(5);
    adc_buf[0]  = adc1.readADC(6);
}
```

We then print out the readings and try to get an idea of the average value of the white line and average value of the background. The values read by the sensors ranges between 0 to 1024. The value of our white line ranges around 400 to 480. Because of this deviation range, we want to add some kind of threshold to define the white line so that we don't mistake the background, which can get a value of 500, with the white line.

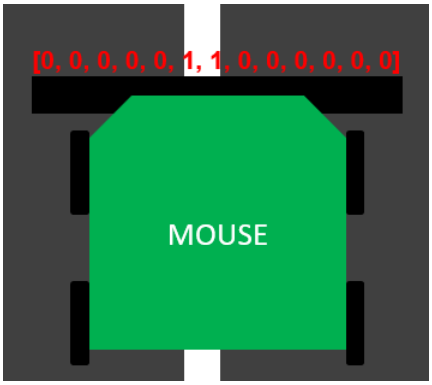
```
void calibrate_adc(int *arr){
    arr[12]= adc1.readADC(0)+ADC_ERROR_MARGIN;
    arr[11]= adc2.readADC(0)+ADC_ERROR_MARGIN;
    arr[10]= adc1.readADC(1)+ADC_ERROR_MARGIN;
    arr[9] = adc2.readADC(1)+ADC_ERROR_MARGIN;
    arr[8] = adc1.readADC(2)+ADC_ERROR_MARGIN;
    arr[7] = adc2.readADC(2)+ADC_ERROR_MARGIN;
    arr[6] = adc1.readADC(3)+ADC_ERROR_MARGIN;
    arr[5] = adc2.readADC(3)+ADC_ERROR_MARGIN;
    arr[4] = adc1.readADC(4)+ADC_ERROR_MARGIN;
    arr[3] = adc2.readADC(4)+ADC_ERROR_MARGIN;
    arr[2] = adc1.readADC(5)+ADC_ERROR_MARGIN;
    arr[1] = adc2.readADC(5)+ADC_ERROR_MARGIN;
    arr[0] = adc1.readADC(6)+ADC_ERROR_MARGIN;
}
```

Now, the problem of looking at an array of 13 values trying to determine whether they're a white line or background is very confusing. For example:

[681, 595, 534, 647, 577, 440, 455, 654, 543, 635, 525, 575]

That is confusing to look at and will be hard to use in other functions later on. So what we did instead was convert the decimal digits into a binary digit instead, where **1 is any value below the threshold (white line)** and **0 and any value above the threshold (black background)**. The array above can be rewritten below as:

[0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0]



In the form above, it's much easier to see where the sensor is detecting the white line. We can see that the mouse is detecting a white line a bit to the left of the center.

This binary data is then stored in a **uint16_t** to save space and processing speed. Having to only calculate a single variable is much faster than calculating 13 separate array data.

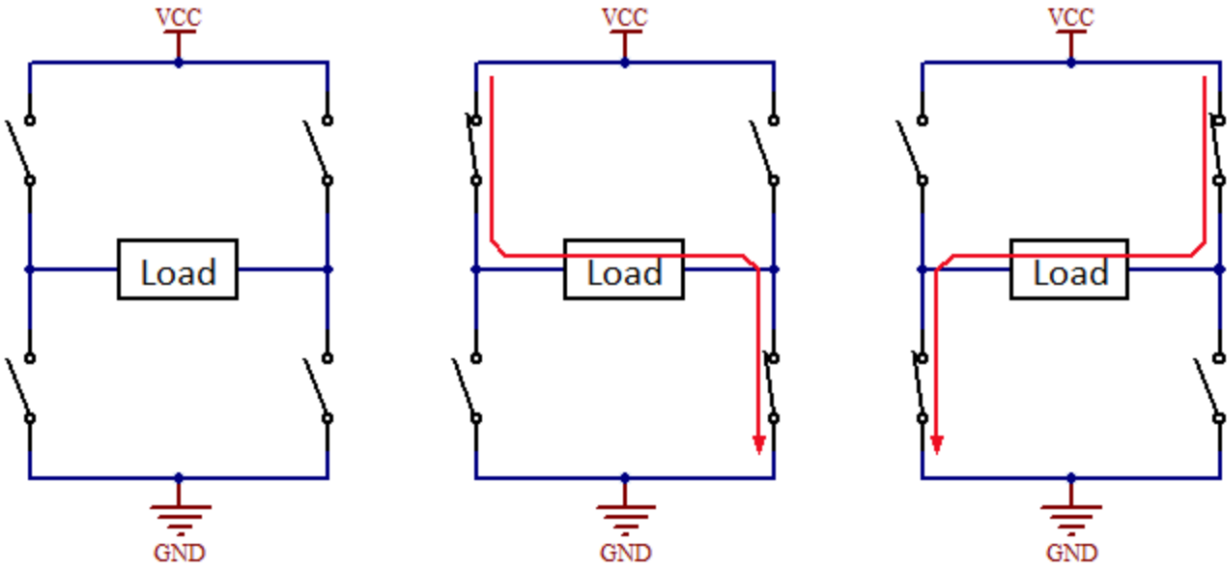
The calculations are done using a bitwise operator. For example, if you want to compare the sensor reading to see if it's a left turn, you AND the sensor reading to isolate the number and compare it with the value you want.

```
if((line_data&0b1111000000000) == 0b1111000000000) //checking for left turn
      //Do something
```

:

Motors and Movement (Hardware)

The motors are controlled by an H-bridge module. An H-bridge allows the user to change the polarity of the motor by setting specific pins to HIGH and LOW. There are two pins to change. Pin1 controls the top-left and bottom-right gate, and Pin2 controls the top-right and bottom-left gate. When Pin1 is HIGH and Pin2 is LOW, the current will flow in direction and cause the motor to spin clockwise. Inversely, if Pin2 is HIGH and Pin1 is LOW then the current will flow in the opposite direction, so the motor is spin counterclockwise. This is used to turn the wheels forward or backward.



By having two of these H-bridge modules, one for each side of the mouse, we can control the left and right wheels of the mouse individually. To drive forward, we make both sides spin forward. To drive backward, we do the same but spin backward. To turn, we make one side spin forward and the other side spin backward. To stop, we just turn off both sides.

Motors and Movement (Software)

Moving Wheels

As mentioned in the hardware section, the direction of the wheel can be controlled by changing Pin1 and Pin2 of the H-bridge module. How this is implemented is simply shown below:

```
void L_backward(int speed){
    analogWrite(L_Motor_1, speed);
    analogWrite(L_Motor_2, 0);
}

void L_forward(int speed){
    analogWrite(L_Motor_1, 0);
    analogWrite(L_Motor_2, speed);
}


void R_backward(int speed){
    analogWrite(R_Motor_1, speed);
    analogWrite(R_Motor_2, 0);
}

void R_forward(int speed){
    analogWrite(R_Motor_1, 0);
    analogWrite(R_Motor_2, speed);
}

void drive_forward(int speed){
    L_forward(L_speed);
    R_forward(R_speed);
}

void drive_backward(int speed){
    L_backward(speed);
    R_backward(speed);
}

void drive_stop(int speed){
    analogWrite(L_Motor_1, 0);
    analogWrite(L_Motor_2, 0);
    analogWrite(R_Motor_1, 0);
    analogWrite(R_Motor_2, 0);
}
```

Gyroscope Module

In the weeks leading up to the exhibit, we decided to switch tactics when it came to detecting turn completion. Initially, the robot was programmed to turn until it detected lines within the middle of the sensor array. This was an efficient method at the beginning of our mouse's run, but we discovered that as the mouse explored deeper into the maze, small offsets from earlier turns would make the mouse prone to errors. This is due to the mouse starting its turn at an angle. This was fine for turns that were at corners or dead-ends, but at intersections where a forward path was possible, this would throw off the detection and cause the mouse to detect lines far too early.

We decided to mitigate this bug by implementing a gyroscope module. This was done with the IMU sensors on the mouse, which detected angular velocity in three axes. Using the z axis, which the mouse was centered on, we used an integrator to obtain the angular displacement. This made our turns more robust, as we could integrate the gyroscope module with our previous turn completion code so that the mouse only began to detect lines once it was about 70°+ degrees displaced from the start of the turn.

PID Control

We used a PID control system to follow the line. A PID system is a feedback control system that corrects for errors based on what your desired error value is. The P stands for proportional, I stands for integral, and D stands for derivative.

Let's use following the line as an example. In this case, we want our vehicle to always be centered with the line. Since we have 13 sensors, we can give those sensors a value from -6 to 6, with 0 being the sensor at the center.

Since the value of 0 represents the center, that will be our targeted error value. When the vehicle has an error value of 0, it will keep the PWM value of the motor as-is. If the error is not 0, then the PWM value will change to correct the wheels until the middle sensor detects the white line again. However, how does PID control come into play here?

The **proportional** part controls how much to increase the correction value by based on the error value. If the car is moving too far left and has an error=-6, then the correction will increase the PWM value of the left wheel by 6 to make the mouse move right. If the mouse is a bit left of the line with an error=-3, then the correct value will be 3 and is added to the left wheel's PWM to make it move right a little bit.

However, the correction value of 3 is too small for our specific case, so a **gain** was multiplied with the correction value. This gain value is **K**, and it's multiplied to make the correction value bigger or smaller. For example, if the mouse is a bit to the left and has an error of -3, the PID will add 3 to the left wheel to make the left wheel a bit faster. Adding 3 might not be enough though;

therefore, we can multiply the error with a constant K to make the correction value bigger. Using Kp=2 (p for proportional), the correction value for when the mouse is a bit to the left will now be 3*Kp = 3*2 = 6, and the PID will add 6 to the left wheel to make it alot faster. The example is shown below:

```
int Kp = 2;
int center_line_PID(){
    int error = line_reading();     //value from -6 to 6

    int P = error;
    prev_error = error;

    int error_correction = P*Kp;
    return error_correction;
}
```

Let's skip over the integral part for now and focus on the derivative part first. The **derivative** part "predicts" what the error value should be based on the rate of change of the error. If the error is changing too fast (i.e. big derivative error value), then the correction value will also be big. The Kd gain value is found using trial and error, but it's typically around 1-10% of the Kp value.

```
int Kp = 2; int Kd = 0.2;
int center_line_PID(){
    int error = line_reading();     //value from -6 to 6

    int P, D;
    P = error;
    D = error - prev_error;
    prev_error = error;

    int error_correction = P*Kp + D*Kd;

    return error_correction;
}
```

Lastly, the **integral** part corrects for all the past errors by summing them up. It corrects for any error over time and will vary depending on the case. This value is not as important as the other two can be omitted in most cases. Its gain value typically range 0-50% of the Kd value.

```
int Kp = 2; int Ki=0.1; int Kd = 0.2;
int center_line_PID(){
    int error = line_reading();     //value from -6 to 6

    int P, I, D;
    P = error;
    I = error + prev_error;
    D = error - prev_error;
```

```
    prev_error = error;

    int error_correction = P*Kp + I*Ki + D*Kd;

    return error_correction;
}
```

What we've shown in this section is the most simple implementation of a PID control. We ended up using a **PID library by Brett Beauregard** instead because it is much more refined and effective than our simple implementation. The simple version is just so you would get a better understanding of what's going on.

Following the Line

As described in the previous section, the line detection uses the center of the line as the error target. Any error value that is shifted from 0 will cause the PID to make corrections until the error value is stable at 0. The values are constrained between 0 and 255 to keep them in bound--the ADC can't exceed a value of 255.

```
void drive_forward(int speed, uint16_t *line_data){
    calculateSensorError(line_data);
    motorPID.Compute();

    int L_speed = constrain(speed - lineCorrection, 0, 255);
    int R_speed = constrain(speed + lineCorrection, 0, 255);

    L_forward(L_speed);
    R_forward(R_speed);
}
```

Driving Straight Without Following Line

We used the difference between the encoder values as our error in case where the white line is not available for the mouse to follow. As a reminder, the encoder returns the angular rotation of the wheels in degrees. To make sure that the mouse is driving straight, the angle between the left wheel and right wheel should be equal--their difference/error should be 0.

```
void drive_straight(int speed){
    encError = enc1.read() - enc2.read();
    encPID.Compute();

    int L_speed = constrain(speed + encCorrection, 0, 255);
    int R_speed = constrain(speed - encCorrection, 0, 255);

    L_forward(L_speed);
    R_forward(R_speed);
}
```

Making a Turn

We used the angular velocity from the gyroscope as our error for the turning. The code for calculating the angular velocity of the gyro is found in our Github IMUTest folder. The PID tries to keep our angular speed to be around 100 deg/s.

```
void drive_right(int speed){
    calculateGyro(&gyroRPS, &gyroAngle);

    //PID to maintain constant linear velocity
    rightTurnError = 100.0 - abs(gyroRPS); //desired "rps"
    rightTurnPID.Compute();


    int L_speed = constrain(speed - rightTurnCorrection, 0, 255);
    int R_speed = constrain(speed - rightTurnCorrection, 0, 255);

    L_forward(L_speed);
    R_backward(R_speed);
}
```

But of course, the gyro isn't super accurate, so we also had a loop that checks whether the mouse is near the center of the line before it continues.

```
void turn_right(int speed, uint16_t *line_data, float targetAngle){
    gyroAngle = 0;
    enc1.write(0); enc2.write(0);

    //Turning the desired angle amount
    while(abs(gyroAngle) < targetAngle){
        drive_right(speed);
    }

    //Making sure it's on a line when it stops
    while(!(*line_data&0b0000011100000)){
        drive_right(speed);
    }

    drive_stop(0);

    enc1.write(0);
    enc2.write(0);
}
```

Tuning the PID

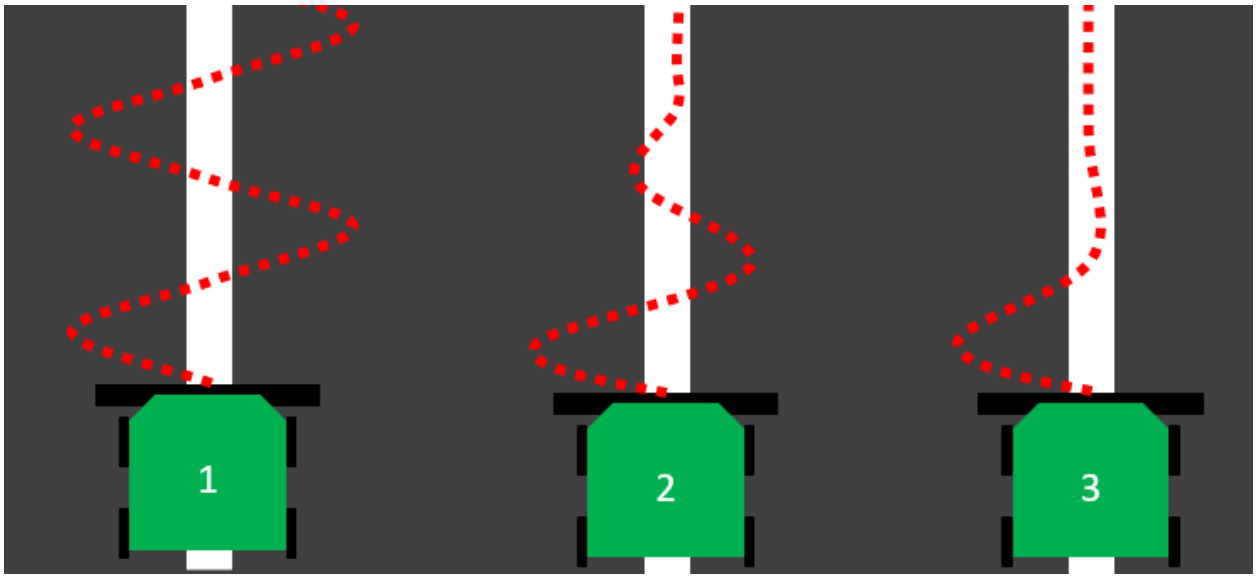
Before we continue, we just want to emphasize that you should use the **PID library by Brett Beauregard** instead of creating your own. His library works much better than what we've created.

There is no single answer to how the PID gain value should be manually tuned, so we'll just describe the one that we've found the easiest and most effective. We will be using the PID for the line follower as the example.

1. **Slowly increase the Kp value until the system is at max oscillation amplitude.**
   In the case of the line follower, you want to increase the Kp value until the mouse oscillates between a straight-white line at a stable maximum error. It's the point where increasing Kp any more would cause it to offshoot from the line.

2. **Slowly increase the Kd value until the oscillation dampens quickly.**
   In the case of the line following, it should take less than 100ms to reach a steady state error of 0.

3. **(Optional) Slowly increase the Ki value until the oscillation is gone. You might want to repeat step 1 and 2 for finer tuning.**



However, there are more accurate ways of tuning the PID control. The PID library has a built-in tuner, but we didn't have the time to research that because what we have right now is already working. If it's not broken, don't fix it.

And if you're hardcore, you can take data of the mouse on a long straight path and solve for the optimal Kp, Ki, and Kd value on MATLAB PID Tuner.

Navigation Code

In this section, I will discuss the navigation code we wrote for our mouse. For our navigation code, we had our mice map out the maze as a Tree structure, with each node of the tree representing an intersection on the map. Ultimately, our mice would solve the maze using a depth first search of the tree. We implemented this structure using object-oriented coding in python, where each ode of the tree was an object that contained data on that intersection. Our goal was to map out the maze as a picture while the mouse solved the maze. There were two major files in our navigation code: Intersection.py and Simulation.py. Intersection.py contained our data structure and maze solving algorithm and simulation.py took a pre-existing tree structure and created an image with the maze and path to the finish.

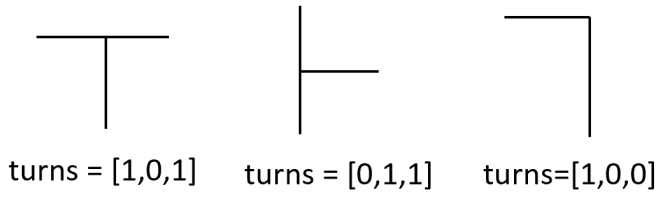
Intersection.py

This file contained the data structures and logic to implement our maze solving algorithm. First we had our intersection object, which held data on each intersection in the maze. Second, we had our mouse object which represented each of our three mice and contained data on which intersection the mouse was currently at, the previous intersection it was at, and which direction the mouse was facing. Lastly was our check_children function and other helper functions which were the main maze-solving algorithm.

Intersection

```
class Intersection(object):
    turns =[0]*3
    distance = 0
    left = None
    straight = None
    right = None
    prev = None
    dead = False
    finished = False
    location = [0]*2
```

Each intersection contains an array of length 3 called turns that indicates whether it has left, straight, or right turns.



It has a value distance, which indicates the distance of this node from the previous node. Each intersection also contains 4 intersections, the intersection down the left path, the straight path, the right path, and the previous intersection. Each intersection also contains Booleans finished

and dead, which mark whether the node is on the path from the start to finish or not. Lastly, each intersection contains an array location, which holds cartesian values for the location of the intersection in the maze, used to draw our graph of the maze later.
For the intersection object we have functions to allow us to set the left straight and right intersections and set the node to finished or dead.

Mouse

In addition to the intersection object, we also have an object called mouse. We initialize all three mice to a mouse object at the beginning of our code and the mouse object allows us to keep information on each individual mouse including its current location on the maze, it's previous node, and the direction it is facing.

```
class Mouse(object):
    mouse_number = 0
    orientation = NORTH
    curr_node = None
    prev_node = None
```

When initializing each mouse, we gave it a mouse number so we could choose which mouse we wanted to send an instruction to. We also had to write functions for the mouse to keep track of its orientation and location of the node it is at.

```
def updateLocation(location, mouse, dist, turnDir):
    if turnDir == LEFT:
        mouse.orientation = (mouse.orientation - 1)%4
    elif turnDir == RIGHT:
        mouse.orientation = (mouse.orientation + 1)%4
    if mouse.orientation == NORTH:
        location[0] += dist
    if mouse.orientation == EAST:
        location[1] += dist
    if mouse.orientation == WEST:
        location[1] -= dist
    if mouse.orientation == SOUTH:
        location[0] -= dist
    return location
```

This function, updateLocation, will take the location of the previous node, the mouse we are calling to turn, the distance from the previous node, and which direction we turned from the previous node and update the orientation and the location of the new intersection. We defined North, East, South, and West to equal 1, 2, 3, and 4, respectively, so we could update the mouse orientation on a right turn by incrementing the orientation mod 4 and a left turn by decrementing the orientation by 1 mod 4. If we are moving north, the Y value of the location should increase by the distance, if we are moving east, the X value of the location should increase, if the mouse is moving west, the X value should decrease, and if the mouse is moving south, the Y value of the location should decrease by the distance.

The second helper function we wrote for the mouse was return_to_alive, which will take a mouse and its current node and return it to the most recent alive node.

```
def return_to_alive(node, mouse):
    if node.dead == False:
        return
    else:
        if node == node.prev.left:
            #turn right and drive to next node
            comm.send_instr(RIGHT)
            mouse.update_node(node.prev)
            return_to_alive(node.prev, mouse)
        elif node == node.prev.straight:
            #drive straight to next node
            comm.send_instr(STRAIGHT)
            mouse.update_node(node.prev)
            return_to_alive(node.prev, mouse)
        else:
            #turn left and drive to next node
            comm.send(LEFT)
            mouse.update_node(node.prev)
            return_to_alive(node.prev, mouse)
```

First the function will check if the current node is alive. If it's not, it will check to see which direction the mouse originally turned at this node and follow its path back to the next previous node and recursively call return_to_alive until it finds an alive node.

Check_Children

This function is the meat of our navigation code. It is a recursive function that performs a depth-first search of the maze until it finds the finish line.

```
def check_children(curr, mouse):

    if curr.turns(1)==1 & curr.left.dead == False:

        #turn_left and drive to next node(mouse_number)
        [dir, dist, k] = mouse.turnLeft()

        finished = dir & 0b1000
        left = dir & 0b0100
        straight = dir & 0b0010
        right = dir & 0b0001

        location = updateLocation(curr.location, mouse, dist, LEFT)
        #acquire data from mouse and create new intersection
        intersect = Intersection(left, straight, right, location, finished)
        curr.setLeft(intersect)

        mouse.updateNode(curr.left)
        check_children(curr.left)

        if curr.finished == True:
            curr.prev.finished = True
            return
```

The function takes an input of the current node and mouse. As of now the code is only implemented for one mouse, but ultimately we would have wanted to optimize it to drive the closest mouse to the node we want to check.
First the method checks to see if the current node has a left turn and that the left turn is still alive, if so, it will turn the mouse left and drive forward to the next node using mouse.turnLeft(). mouse.turnLeft() returns an array with dir, which is a 4 bit number that gives data on if we reached the finish line, if there is a left turn, if there's a straight, and if there's a right turn and the distance from the previous node. Using this data, we can update the location on the mouse and find the location of the new node using the function we discussed earlier called updateLocation. Now that we have the data on the intersection, we can create a new node called intersect and set the current node.left to intersect. We then update the location of the mouse and recursively call check children on the new node. After that recursive call finishes, we can check to see if our current node is now finished, and if it is we set the previous node to *finished* as well.

The code repeats for the straight and right turns to see if they exist and perform depth-first searches of both those paths. If we traverse all existing paths and none of them lead to the finish line, we set the current node to dead and send the mouse back to the most previous alive node.

```
else:
    set_node_dead(curr)
comm.send_instr(UTURN)
return_to_alive(curr, mouse)
```

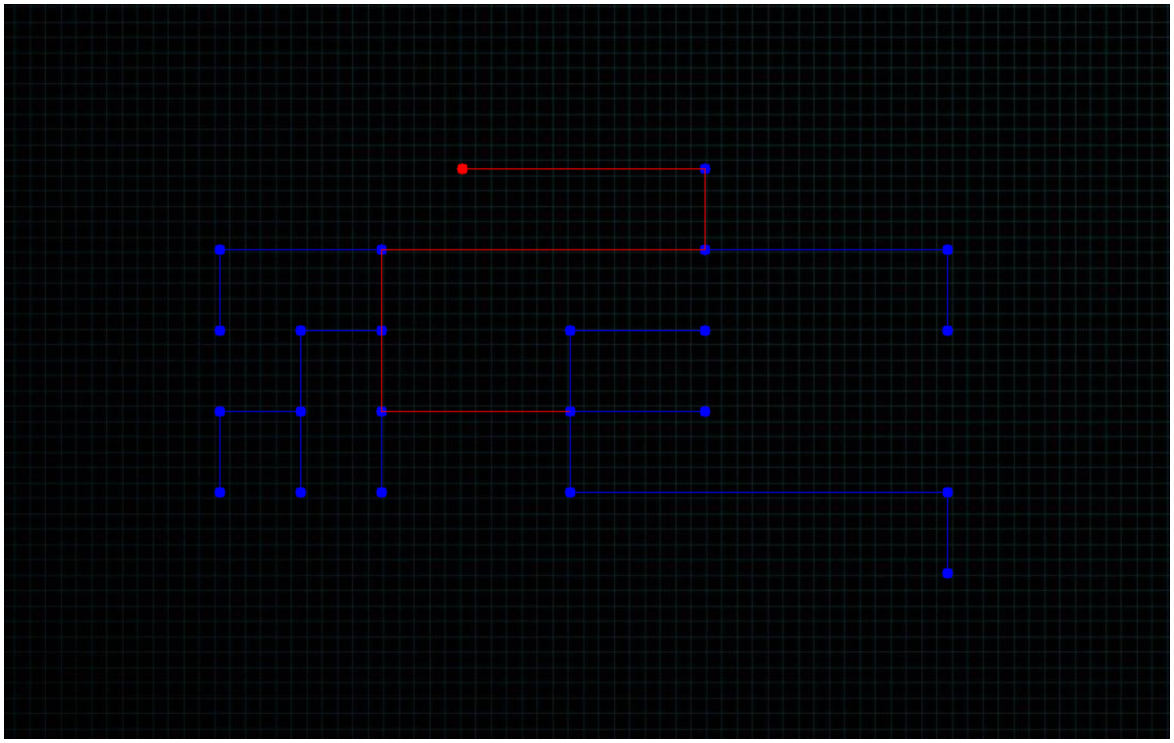
Simulation.py

This file takes a pre-existing intersection tree and graphs it out as an image. In the current file, I manually created the tree structure for two of the mazes on the final presentation to graph out. We used CV2 to create the image using its circle and line functions. The program works by taking a blank black picture and adding circles in the locations of the intersections and lines along the paths of the maze. This function also does a depth-first search, similar to our navigation code and can be seen in our video presentation of our navigation code.

```python
def add_node_to_plot(node, img):
    if node.finished:
        img = cv2.circle(img, node.location, 5, (0, 0, 255), -1)
        if node.prev:
            img = cv2.line(img, node.location, node.prev.location, (0,0,255), 1)
        cv2.imshow('image', img)
        cv2.waitKey(800)
    else:
        img = cv2.circle(img, node.location, 5, (255, 0, 0), -1)
        if node.prev:
            img = cv2.line(img, node.location, node.prev.location, (255,0,0), 1)
        cv2.imshow('image', img)
        cv2.waitKey(800)
    if node.left:
        add_node_to_plot(node.left, img)
    if node.straight:
        add_node_to_plot(node.straight, img)
    if node.right:
        add_node_to_plot(node.right, img)

    if node.finished:
        if node.prev:
            img = cv2.line(img, node.location, node.prev.location, (0,0,255), 1)
            cv2.imshow('image', img)
            cv2.waitKey(800)
        node.prev.finished = True

    return img
```



Bluetooth Communication

There are different types of bluetooth. The one we used was the Bluetooth Low Energy (BLE) because it was built into the Arduino BLE 33 that we're using.

The bluetooth works by advertising services of a certain ID value. You can think of the service as a function that is shared between devices, and the ID is a pointer to that function. So if you have a code on your Arduino, and one on your linux system, you have to tell them the ID of which function to look at. We call this ID the UUID. The UUID can be any number as long as it's unique. We generated our UUID by googling "UUID generator". Again, the UUID can be any random generic number as long as it's a unique number; this is so that it doesn't confuse with other ID numbers from other devices in the room.

On the arduino side, you have to give your advertised service a name. This is the name that will show up when you try to pair. The name will be attributed to the device address, which is a unique identifier to every device. The rest of the code is just setting up the service on both the arduino code and python code.

We created two services, one for writing data and one for reading data. If the service for writing data updates, then the bluetooth code on the Arduino will execute. Otherwise, it'll act like no data has been received. For the reading data, we had a flag bit on the MSB. If that bit is a 1 then it means that the arduino has written some value to the linux system. If it's a 0 then no value is written.

BLE is one of those things that is difficult to understand and you'll just have to stare at the code for a while to figure it out. The useful simplified bluetooth code we have is in the bluetooth_test3 folder:

ENEE408I_FALL2021_Team4/bluetooth_test3 at master · UMD-ENEE408I/ENEE408I_FALL2021_Team4 (github.com)

RTOS and Multithreading

RTOS stands for Real-Time Operating System. It's a set of code that manages multiple tasks in your code so that they can run in parallel. It's pretty much multithreading. We didn't have much use for it except for using it to play music in a second thread while our main code is running on the arduino. However, it is nice to know how to use it in case you need it.

The RTOS library we use comes from the **mbed** library. It is extremely simple to use and only takes up a few lines of code to use it. You first define the library. Then you define the object. Then you create the function that you want to run in parallel and have it run an infinite while-loop.

```
//RTOS stuff
#include <mbed.h>
using namespace mbed;
using namespace rtos;

//Threads. Create a new one for each new function you want to run in
parallel
Thread t1;
Thread t2;

void setup() {
    t1.start(t1_func);
    t2.start(t2_func);
}

void loop(){
}

void t1_func(){
    //setup stuff here
    while(1){
          //loop stuff in here
    }
}

void t2_func(){
    //setup stuff here
    while(1){
          //loop stuff in here
    }
}
```

Challenges and Solutions:

Turning While Stationary

We had trouble making our vehicle turn in one spot. We initially tried making it so that the left wheel rotates at the same but opposite angle from the right wheel. However, for some reason, even when the wheel turns at the same angle, the car would turn in a curved path.

We end up fixing this problem by using the gyro. Having the mouse turn with a constant angular speed ended up making it turn in a stationary area. The mouse did this by making small pulses of high speed spin, which gave it enough torque to move both wheels at the same time. We also had separate PID function for the left and right turn so that they don't share the same error values.

Driving Straight After A Turn

We had a problem where the mouse would not follow the line after it made a turn. This problem ended up being we used the same PID function for the turn as the function for making the car drive forward. So when the car turned, it would keep the old error value from the turn and that messes up the correction value for when the mouse starts driving forward. The big gap in error value causes the correction value to be big and offshoot the mouse from the center line. This problem actually originated from our first implementation of the line following code. We had one wheel spinning at constant speed and the other wheel copying the rotation of the first wheel. So we were actually using two separate PID systems to control the drive forward function initially.

This was fixed by scrapping the idea of keeping constant speed. We realized we didn't need it to be perfect, we discarded the function for maintaining constant speed since it made every other function much more complicated to implement. We ended up controlling both wheels using only one PID control, and used a PID library instead of the one we wrote. What we learned is that don't complicate stuff if you don't need to. **If there's a library for it, use it!** Don't write stuff from scratch because it won't be as good as the one the experts wrote.

Not Detecting Line Randomly

Sometimes the sensors would not detect the line at random moment, most likely due to noises. When this happens, the mouse thinks it's a dead end and starts to do a 180 degree turn. This was fixed by having a counter to count how many times the mouse sees the empty background to confirm that it's not random noise that's causing the misreading.
Also, do not use the RTOS to read the lines. Because the Arduino only has a 1-core processor, it does multithreading by alternating between processes, which means it could switch back to the main loop's code before finishing reading all the ADC values from the sensors. This will give the wrong reading.

Conclusion:

This class was a great learning experience. Even though we mentioned using available libraries instead of writing our own to make life easier, we learned a lot by writing the functions from scratch and making mistakes. We've also learned that we should avoid building our code upon a broken base code. We spent so much time building upon the broken movement code. Once we realized there's no other way to fix the turning problem, we had to rewrite the whole entire base code for the movement in a single day. Had we fixed this at the beginning, we would have had more time to refine the movement so that it'll be more accurate and consistent.