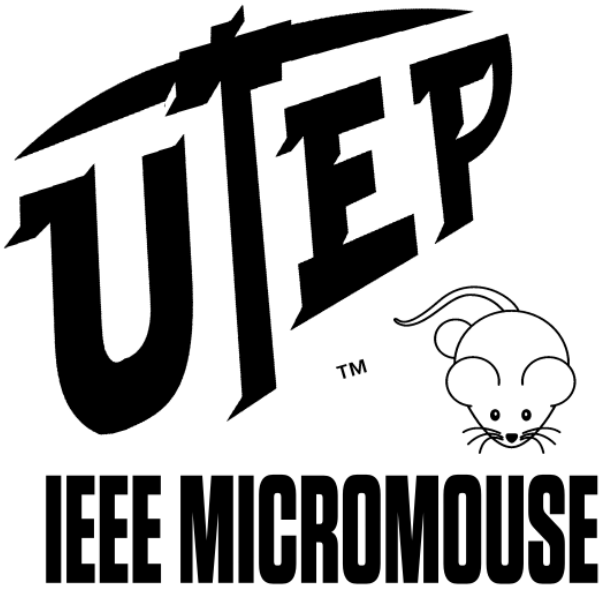

IEEE MICROMOUSE

Workshop Week 8:

Micromouse wheel position mismatch and wall control

Objective:

Enhance the forward control with wall distance and wheel position mismatch compensation. Measure distance from wall and calculate error. Measure encoder wheel and calculate position mismatch. Write the control software to compensate for these errors. Understand the control system design for the mouse.

Background information:

Last week we worked on distance control using the encoder measurements and a proportional controller. This week we will enhance the controller by adding a wheel position mismatch and a wall distance controller.

As explained last week, we can compensate for errors between a desired distance and the actual distance. Our controller will proportionally fix the error distance so that we match it to our desired distance. Using this idea, we can use desired targets and actual sensor measurements to compensate for other disturbances or errors in our mouse.

On the first workshops, some of you noticed that the mouse was steering towards one side even though the speed and direction on both wheels was the same. This is caused by the weight distribution of the mouse and the manufacturing variances between both motors. We want to prevent this behavior from occurring, so we are going to add another proportional controller that will try to minimize the position mismatch between wheels. To design this control, let's first think about the best-case scenario for our mouse motors. If both wheels are always moving at the same speed, then both encoders should read the same value. Now, we know this is not happening because we can see the mouse steering one way or the other, so any deviation from this desired measurement is our error. This can be translated as

$$wheel_{error} = 0 - (right_{encoder} - left_{encoder})$$

$$wheel_{error} = left_{encoder} - right_{encoder}$$

Notice that our desired value in this case is zero since we do not any difference between our wheels.

Another scenario that we are trying to prevent is collision against walls. To do this, we must drive the mouse perfectly at the center of the maze walls. This is of course impossible in practice, and at some point, the mouse will crash against a wall. if you been paying attention, then you know that we can correct for this by adding a proportional controller that compensates for the error on a desired wall distance. In this case the sensor that will measure this difference is the right and left infrared sensors. So, as we did with the wheel position mismatch controller, let's define some equations for our error:

$$Wall_{error-left} = infrared_{left-desired} - infrared_{left-actual}$$
$$Wall_{error-right} = infrared_{right-desired} - infrared_{right-actual}$$
$$Wall_{error-center} = wall_{error-right} - wall_{error-left}$$

If $infrared_{left-desired}$ and $infrared_{right-desired}$ are equal, then the $Wall_{error-center}$ can be reduced to

$$Wall_{error-center} = infrared_{left-actual} - infrared_{right-actual}$$

We want to minimize this off-center error, so again our target should be to always have a measurement of 0 between the left and right infrared sensors (assuming they both measure the same value).

Look at figure 1 and 2 to graphically see how these controllers interact with each other. If you still have aa difficult time understanding the diagrams or any of the concepts, please reach out to any IEEE officer for help.

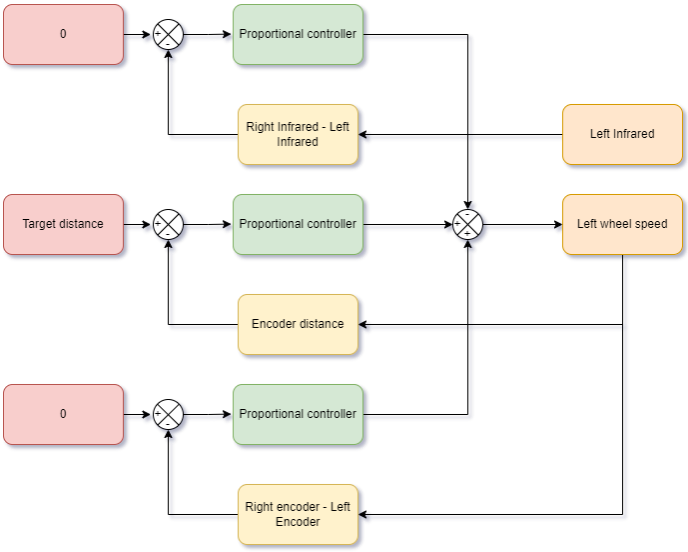

Figure 2 Left wheel controller diagram

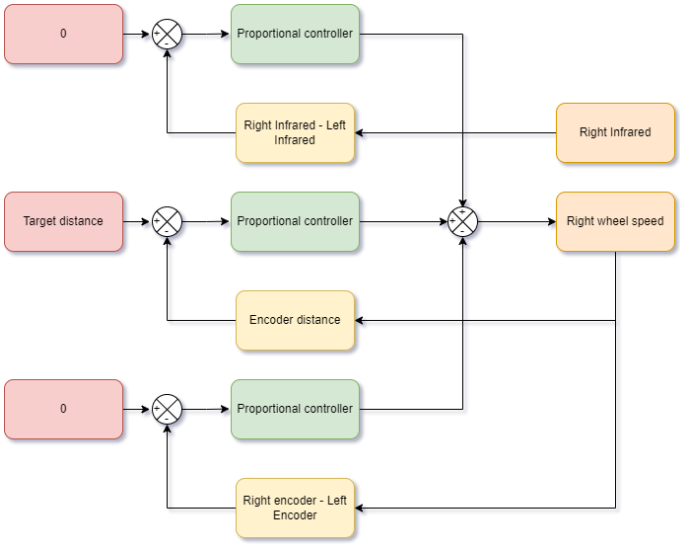

Figure 1 Right wheel controller diagram

Clamp and find_direction functions

Before diving into the control software, we need to organize the code from last week into smaller chunks. Clamping values and finding the direction of wheels can be abstracted into functions as following:

```
1.  float clamp(float value, float max, float min)
2.  {
3.    if(value > max)
4.    {
5.      value = max;
6.    }
7.    else if(value < min)
8.    {
9.      value = min;
10.   }
11.
12.   return value;
13. }
14.
15. bool find_direction(int value)
16. {
17.   bool dir;
18.   if(value > 0)
19.   {
20.     dir = true;
21.   }
22.   else if(value < 0)
23.   {
24.     dir = false;
25.   }
26.
27.   return dir;
28. }
```

We can now clamp the value for the controllers as follows:

```
1.  float distance_output = clamp(distance_kp*distance_error, 150.0, -150.0);
```

And find the direction of a wheel based on its output,

```
1.  bool left_dir = find_direction(left_output);
```

Calculating the errors and outputs

Next, we need to calculate the errors from our different controllers. Following the equations from the background information section and figures 1 and 2 we can write the following code:

```
1.  const long int MIN_ERROR = 50;
2.  const long int target_distance = 1090;
3.
4.  void control_forward(void)
5.  {
6.     /*Keep running the control loop until the error is less that the
7.     * minimum specified error (MIN_ERROR)
8.     */
9.    int distance_error;
10.   do
11.   {
12.     /*Read encoder for left and right motors. Estimate
13.      * mouse traveled distance by taking the average of
14.      * both encoders.
15.      */
16.     long int r_count = right_encoder.read();
17.     long int l_count = -left_encoder.read();
18.     long int encoder_distance = (r_count + l_count) / 2;
19.     int r_ir = ir_1_read(150);
20.     Int l_ir = ir_2_read(150);
21.
22.     /*Compute distance, speed mismatch and distance from wall error*/
23.     distance_error = target_distance - encoder_distance;
24.     int mismatch_error = r_count - l_count;
25.     int ir_error = r_ir - l_ir;
26.
27.     delay(20);
28.   }while(abs(distance_error) > MIN_ERROR);
29.
30.   /*Stop both motors*/
31.   motor_0_speed(0, true);
32.   motor_1_speed(0, true);
33.
34.   /*Reset encoders*/
35.   right_encoder.write(0);
36.   left_encoder.write(0);
37. }
38.
```

Notice that we now have 3 sources of errors that we want to correct,

1. Distance error
2. Wheel position mismatch error
3. Infrared distance from wall error

To minimize this error, we will use the proportional error code from last week. This time we need 3 proportional parameters $distance_{kp}, mismatch_{kp}, IR_{kp}$ to tune the controller output.

Also, we do not want one our controllers to saturate the output, as this will leave the other controllers without any control output to correct its error. Using the clamp function can we

limit the output of the controllers to a percentage of the maximum speed (+-255). In this case we assigned the following values of the maximum speed:

$$distance_{output} = \pm 150$$

$$mismatch_{output} = \pm 50$$

$$IR_{output} = \pm 50$$

Which corresponds to the 58%, 19% and 19% of the max speed value.

The code for the output with clamping can be written as follows,

```
1.  const long int MIN_ERROR = 50;
2.  const long int target_distance = 1090;
3.  const float distance_kp = 1.0;
4.  const float speed_kp = 2.0;
5.  const float ir_kp = 2.0;
6.
7.  void control_forward(void)
8.  {
9.      /*Keep running the control loop until the error is less that the
10.    * minimum specified error (MIN_ERROR)
11.    */
12.   int distance_error;
13.   do
14.   {
15.     /*Read encoder for left and right motors. Estimate
16.      * mouse traveled distance by taking the average of
17.      * both encoders.
18.      */
19.     long int r_count = right_encoder.read();
20.     long int l_count = -left_encoder.read();
21.     long int encoder_distance = (r_count + l_count) / 2;
22.     int r_ir = ir_1_read(150);
23.     int l_ir = ir_2_read(150);
24.
25.     /*Compute distance, speed mismatch and distance from wall error*/
26.     distance_error = target_distance - encoder_distance;
27.     int mismatch_error = r_count - l_count;
28.     int ir_error = r_ir - l_ir;
29.
30.     /*Proportional error for all of our errors. Clamp values for all outputs
31.     Check the function clamp*/
32.     float distance_output = clamp(distance_kp*distance_error, 150.0, -150.0);
33.     float mismatch_output = clamp(speed_kp*mismatch_error, 50.0, -50.0);
34.     float ir_output = clamp(ir_kp*ir_error, 50.0, -50.0);
35.
36.     delay(20);
37.   }while(abs(distance_error) > MIN_ERROR);
38.
39.   /*Stop both motors*/
40.   motor_0_speed(0, true);
41.   motor_1_speed(0, true);
42.
43.   /*Reset encoders*/
44.   right_encoder.write(0);
45.   left_encoder.write(0);
46. }
47.
```

Wall missing from one side

Our controllers will work fine if we have both walls at each side of the mouse. Unfortunately, the maze has sections that have only one wall. This will cause the mouse to steer towards the empty space. To solve this, we need to detect whenever there is not a wall on one of the sides of the mouse, and artificially set the measurement as if there was a wall there.

Add this code to the previous section,

```
1.  const int RIGHT_IR_THRESHOLD = 200;
2.  const int LEFT_IR_THRESHOLD = 200;
3.  const long int MIN_ERROR = 50;
4.  const long int target_distance = 1090;
5.  const float distance_kp = 1.0;
6.  const float speed_kp = 2.0;
7.  const float ir_kp = 2.0;
8.
9.  void control_forward(void)
10. {
11.     /*Keep running the control loop until the error is less that the
12.    * minimum specified error (MIN_ERROR)
13.    */
14.   int distance_error;
15.   do
16.   {
17.     /*Read encoder for left and right motors. Estimate
18.      * mouse traveled distance by taking the average of
19.      * both encoders.
20.      */
21.     long int r_count = right_encoder.read();
22.     long int l_count = -left_encoder.read();
23.     long int encoder_distance = (r_count + l_count) / 2;
24.
25.     /*Read infrared sensors. Ideally both infrared sensors
26.      * should read the same value form the wall, so for our error
27.      * we can do r_infrared - l_infrared. Check if there is no wall
28.      * one side or the other since this will cause the mouse go towards one wall
29.      */
30.     int r_ir = ir_1_read(150);
31.     int l_ir = ir_2_read(150);
32.
33.     if(r_ir < RIGHT_IR_THRESHOLD)
34.     {
35.       /*force "good measurement*/
36.       r_ir = 700;
37.     }
38.     if(l_ir < LEFT_IR_THRESHOLD)
39.     {
40.       /*force "good measurement*/
41.       l_ir = 700;
42.     }
43.
44.     /*Compute distance, speed mismatch and distance from wall error*/
45.     distance_error = target_distance - encoder_distance;
46.     int mismatch_error = r_count - l_count;
47.     int ir_error = r_ir - l_ir;
48.
49.     /*Proportional error for all of our errors. Clamp values for all outputs
```

```
    Check the function clamp*/
    float distance_output = clamp(distance_kp*distance_error, 150.0, -150.0);
    float mismatch_output = clamp(speed_kp*mismatch_error, 50.0, -50.0);
    float ir_output = clamp(ir_kp*ir_error, 50.0, -50.0);

    delay(20);
  }while(abs(distance_error) > MIN_ERROR);

  /*Stop both motors*/
  motor_0_speed(0, true);
  motor_1_speed(0, true);

  /*Reset encoders*/
  right_encoder.write(0);
  left_encoder.write(0);
}

```

Output functions for wheels

Finally, we can calculate the output speed and direction of the wheels. Remember that we need to mix the output of the proportional controllers into a single output. The following equation shows how to combine the outputs for the left and right wheel speed.

$$left_{output} = distance_{output} + mismatch_{output} - IR_{output}$$
$$right_{output} = distance_{output} - mismatch_{output} + IR_{output}$$

For the direction, we can use the find_direction function we wrote earlier,

```
bool left_dir = find_direction(left_output);
bool right_dir = find_direction(right_output);
```

Combining the code from the previous sections will yield this final code

```
const int RIGHT_IR_THRESHOLD = 200;
const int LEFT_IR_THRESHOLD = 200;
const long int MIN_ERROR = 50;
const long int target_distance = 1090;
const float distance_kp = 1.0;
const float speed_kp = 2.0;
const float ir_kp = 2.0;

void control_forward(void)
{
    /*Keep running the control loop until the error is less that the
   * minimum specified error (MIN_ERROR)
   */
  int distance_error;
  do
  {
    /*Read encoder for left and right motors. Estimate
     * mouse traveled distance by taking the average of
     * both encoders.
```

```
20.       */
21.     long int r_count = right_encoder.read();
22.     long int l_count = -left_encoder.read();
23.     long int encoder_distance = (r_count + l_count) / 2;
24.
25.     /*Read infrared sensors. Ideally both infrared sensors
26.      * should read the same value form the wall, so for our error
27.      * we can do r_infrared - l_infrared. Check if there is no wall
28.      * one side or the other since this will cause the mouse go towards one wall
29.      */
30.     int r_ir = ir_1_read(150);
31.     int l_ir = ir_2_read(150);
32.
33.     if(r_ir < RIGHT_IR_THRESHOLD)
34.     {
35.       /*force "good measurement*/
36.       r_ir = 700;
37.     }
38.     if(l_ir < LEFT_IR_THRESHOLD)
39.     {
40.       /*force "good measurement*/
41.       l_ir = 700;
42.     }
43.
44.     /*Compute distance, speed mismatch and distance from wall error*/
45.     distance_error = target_distance - encoder_distance;
46.     int mismatch_error = r_count - l_count;
47.     int ir_error = r_ir - l_ir;
48.
49.     /*Proportional error for all of our errors. Clamp values for all outputs
50.     Check the function clamp*/
51.     float distance_output = clamp(distance_kp*distance_error, 150.0, -150.0);
52.     float mismatch_output = clamp(speed_kp*mismatch_error, 50.0, -50.0);
53.     float ir_output = clamp(ir_kp*ir_error, 50.0, -50.0);
54.
55.     /*Calculate final output speed for each wheel*/
56.     int left_output = ((int)distance_output + (int)(mismatch_output) - (int)(ir_output));
57.     int right_output = ((int)distance_output - (int)(mismatch_output) + (int)(ir_output));
58.
59.     /*Find the turn direction of each wheel. Check the find_direction function*/
60.     bool left_dir = find_direction(left_output);
61.     bool right_dir = find_direction(right_output);
62.
63.     motor_0_speed(abs(right_output), right_dir);
64.     motor_1_speed(abs(left_output), left_dir);
65.
66.     delay(20);
67.   }while(abs(distance_error) > MIN_ERROR);
68.
69.   /*Stop both motors*/
70.   motor_0_speed(0, true);
71.   motor_1_speed(0, true);
72.
73.   /*Reset encoders*/
74.   right_encoder.write(0);
75.   left_encoder.write(0);
76. }
77.
```

Exercise:

Place the previous code into your micromouse.cpp and micromouse.h code. Call your code on the sketch file as follows.

```
1.  #include "micromouse.h"
2.
3.  void setup()
4.  {
5.    /*Initialize motors*/
6.    motors_init();
7.    ir_init();
8.    /*Initialize serial console*/
9.    Serial.begin(9600);
10. }
11.
12. void loop()
13. {
14.   /*Wait for button press*/
15.   while(digitalRead(BUTTON) != HIGH);
16.   delay(1500);
17.
18.   control_forward();
19. }
20.
```

Use the maze to see that the mouse is staying in between the walls of the mouse, is not steering and it moves the desired distance each time.

If you prefer, write all of the code into the sketch file and test it this way