MAE 3780: Robot Competition Report

Jonathan Distler

May 5, 2026

1 Robot Design and Strategy Overview

The following is our completed robot:

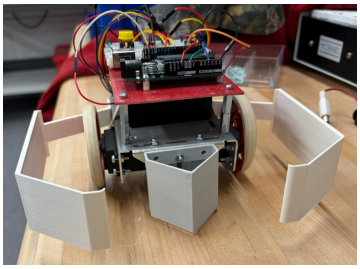


Figure 1: Finalized Robot

Our team was called "Linsanity" because we wanted to personify the feeling of catching lightning-in-a-bottle with our robot. We wanted to be fast. We wanted to be easy to maneuver. And, most importantly, we wanted to win a lot in our short time (05/05/2026 competition day).

The main philosophy of our robot was to create a mechanical system that was both simple and effective, compared with the more complicated solutions that had been explored in semesters past (e.g. revolving arms). Our mechanical system took inspiration from passive mechanical systems like bulldozers to partition the blocks in towards the middle of our robot before being deviated towards one side (or the other) of the robot via the front-sled via an inherent instability. In short, the blocks would collide with our robot head-on, then move from being head-on with the robot to being partitioned next to the wheels with our implemented passive robotic arms.

Ideally, the robot would move forward, detect a color change and then turn left 90-degrees before driving forward until it detects the black border border. Once it detected black, then it would 180 degrees towards the other border. When it detected the opposite border, then ideally, it would orient itself towards the center of its starting box, drive there, then stop.

Electrically, the robot was very simple, it used two h-bridges to drive the right and left motors; it used a button to initiate our strategy sequence, and it used the TCS3200 Color-Sensor to deviate between the two halves of the board (blue and yellow) as well as the borders (black). Although there had been suggestions to use the Bot-Bot Robot QTI sensor, we found that if we adjusted the δ period change between the different colors we

could accomplish the same distinction without as much electrical complexity (with a δ more than 300 µs but less than 1200 µs it would be a change from blue to yellow or vice versa and a δ of more than 1200 µs would be a change from blue or yellow to black).

For our software, our main functionality came from defining drive functions for a specified distance. We would then take the distance, along with some mathematical formalism (provided below) to determine how long (and in what direction) our motors would run. The tricky part was that the delay function native to C takes only took integers as an argument, so we used a "for" loop that runs n-number of times each at 1-millisecond to approximate the true delay necessary for the motors to complete their true revolution amounts. After we had defined the drive functions, we just had to implement them according to our strategy and sensing (Appendix D for strategy, Appendix E for code):

$$D = 2.57 \text{ in}, \ \text{RPM} = 45, \ d = 4.72 \text{ in}$$

$$v = \pi D \cdot \frac{\text{RPM}}{60}$$

$$t_{\text{lin}} = \frac{d}{v}$$

$$t_{\text{ang}} = \frac{\pi \alpha}{2v/d}$$

Design Process Reflection

We quickly determined our general mechanical design principles and strategies for the pitch milestone. Then, we started from small implementations such as one wheel moving, to larger implementations such as both wheels moving in coordination, which worked well with the milestone guidelines. Then, we began implementing more control strategies on-top of our basic drive functionality; mostly opting for a color-sensor. After, we had our basic driving and sensing functionality, we closed the loop and implemented them all at once. Then, we began improving upon our original arm prototypes which were much smaller than we felt was optimal; with longer arms and more arm coverage width wise-wise, our cross-section took up almost the entire 8 in x 8 in bounding box such that the robot could ideally control more blocks.

Our testing process involved designing a circuit in TinkerCAD (if the electrical component existed in the simulation), or referring to previously developed intuition from our lab sections (for the color-sensing in particular). Then, we would translate those circuits onto our physical breadboard along with its respective coding logic. Then, we would test very basic movements, for example, we tested to make sure our drive logic actually turned the right direction and roughly the correct number of degrees, and that our color sensor actually detected different periods across the board. Then, we would integrate the basic functionality into a larger code block and then reintegrate the circuit with the rest of the electrical components.

The milestones provided really great guide-rails for us to improve our design iteratively without having too much add complexity at any one stage. We didn't run in any major roadblocks throughout the process, aside from milestone 2 because our motors had different implict wear-and-tear, and they didn't move at the same true RPM (even though our code was defined such that they did); as a result we had to tune our turning angles to better match the orthogonality of the track.

Competition Analysis

Our performance was as follows

Wins	Losses	Block Differential
3	4	0

We felt like our robot underachieved compared with how we had performed in milestone 4. We naively believed our new battery would be 9V, and after having submitted our code, our robot couldn't replicate its previous performance; it took a few matches to really deduce the error that the battery was actually at 6.4 V and wasn't providing ample voltage for the color sensor to work properly. As a result we lost a few matches to begin with. Eventually, with a new battery, our edge detection and strategy worked very well and we were able to beat many of our opponents to the center line before turning to collect more blocks along the center line. The main issue was that, as previously mentioned, our motors didn't run at the same rate so our forward movement began to drift, as did our turns, so we didn't truly follow along the center line to optimize the number of blocks. Additionally, with just a sole color sensor we were able to detect both blue to yellow transitions and the borders which was a vital assumption of our robot. Given our robot's respective form factor, short and stocky, we were able to bulldoze a few teams to collect blocks which was not a consideration during our design, but which thankfully worked to benefit our team. Lastly, we felt that the task horizon of our robot was fairly short (one forward movement followed by a two turns), so our robot didn't collect any robots after the 20 s mark and that left a lot of time where our robot wasn't actively contributing towards point scoring.

Conclusions

Considering our performance, we had a few improvements in mind. Firstly, we would implement a pulse width modulation (PWM) protocol for our motors such that they would both move at a designed RPM so there wouldn't be any drift. Secondly, we would implement a new strategy (contingent on not having any drift) where our robot would zig-zag along the center line rather than returning to its start position so that it had more opportunities to collect blocks, especially in the event that another robot moved a block, such that more surface-area would be covered and hopefully more blocks would be picked up.

We thought that our passive design worked very well, so we would probably go forward with something similar in the future. We noticed a lot of other teams added some barrier between the front of their wheels and their arm structure to prevent the wheels from incidentally contacting the blocks and messing up the movement patterns of the robot. Considering the low allotted cost of assembly, we felt like that wasn't a reasonable expectation while staying under budget, but with a larger budget we would certainly opt for a similar partitioning device. Additionally, we would consider using a rotating spindle to increase our robot's effective surface area covered, although that would deviate from our more passive design.

Appendix

Appendix A

The bill of materials is as follows:

Part Name	Part Quantity	Per Part Cost	Total Cost
PLA-Robot Arm (1st iteration)	2	$5.75	$11.50
PLA-Robot Sled	1	$6.52	$6.52
PLA-Robot Arm (final iteration)	2	$8.96	$17.93
Totals	-	-	$35.95

Therefore, the final cost ended up being $35.95.

Appendix B

The circuit diagram is as follows, it uses two L1910 H-Bridges to control the motors, a button to begin the strategy logic, and a color sensor with a blue-filter and 20% output frequency scaling:

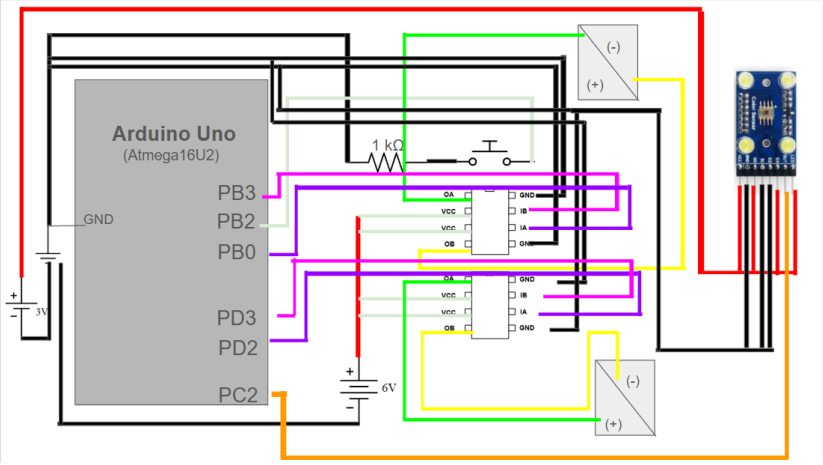


Figure 2: Circuit Diagram for Robot - Notice a common ground shared among all circuit elements and two unique power sources (3V and 6V respectively) providing power to the color sensor, and the motors, with the Atmega16U2's power from an external source

Appendix C

The following outlines the CAD designs of the robot:

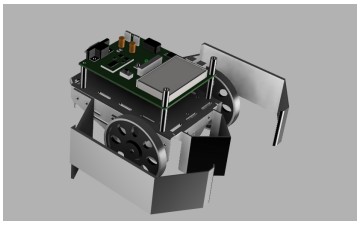


Figure 3: Robot Ortho. View

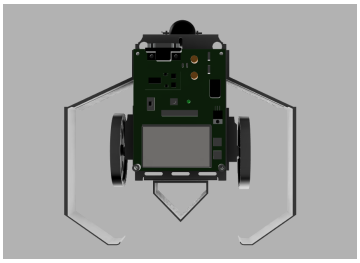


Figure 4: Robot Top View

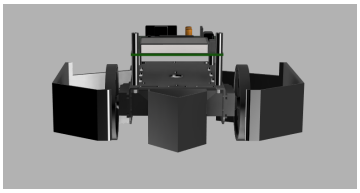


Figure 5: Robot Front View

The following are the iterations of the arms for the robot, with the first iteration testing the proof of concept of the anisotropic strain of the arm in the normal direction, and the second iteration building on that along with an updated CAD model to maximize the allotted space:

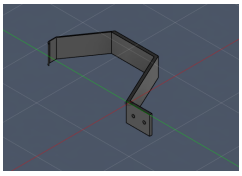


Figure 6: First Iteration Ortho. View



Figure 7: First Iteration Side View

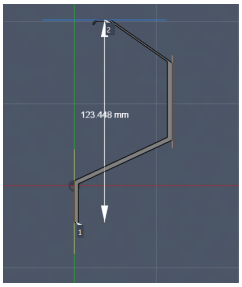


Figure 8: First Iteration Length

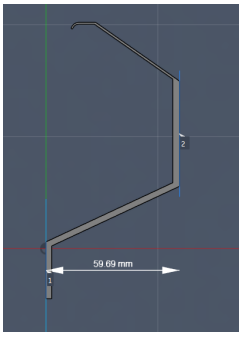


Figure 9: First Iteration Width

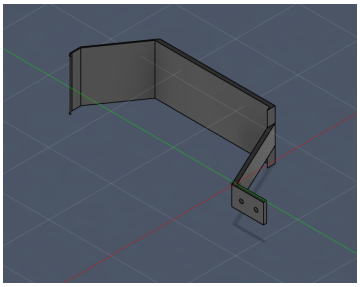


Figure 10: Final Iteration Ortho. View



Figure 11: Final Iteration Side View



Figure 12: Final Iteration Length

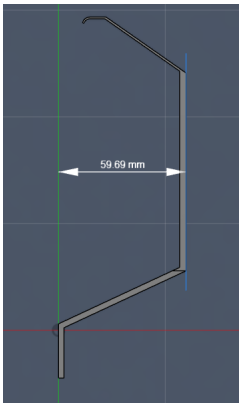


Figure 13: Final Iteration Width

Finally, the following are the CAD views of the "pusher" which would ideally cause the blocks to funnel towards the wheel-base of the robot for collection:

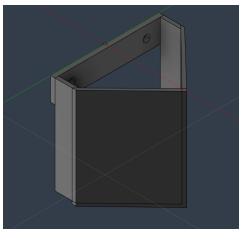


Figure 14: Pusher Ortho. View



Figure 15: Pusher Front View

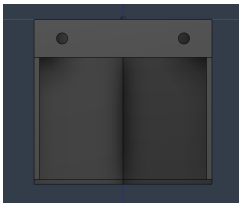


Figure 16: Pusher Back View

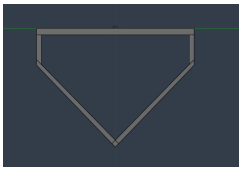


Figure 17: Pusher Top View

Appendix D

The basic strategy was to drive forward until the robot detected a color change, then it would immediately turn left 90 degrees and drive until it were to detect the black border (as mentioned previously we found pretty accurate results just using the color sensor), then it would turn 180 degrees, and upon finding the second black border it would've made an entire sweep of the center line, so it would reorient itself towards the middle of its starting grid, away from the opponent.

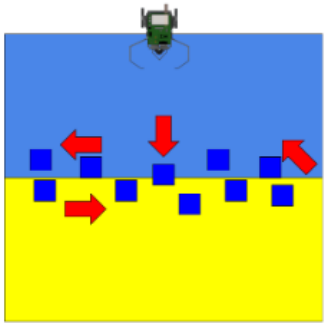


Figure 18: Pictorial Representation of Desired Robot Movement

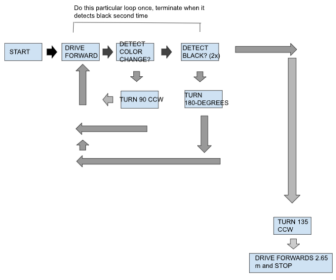


Figure 19: Strategical Flowchart Showcasing Desired Robot Movement

Appendix E

The code is as follows:

Listing 1: Robot Control Code

```
1 #include <avr/io.h>
2 #include <avr/interrupt.h>
```

```
3  #include <util/delay.h>
4  #include <stdint.h>
5  #include <math.h>
6
7  // ---------------- CONSTANTS ----------------
8  // wheel diameter (looks like inches)
9  float wheel_diam = 2.57874016;
10 // motor speed in RPM
11 float rpm = 45;
12 // distance between wheels (used for turning math)
13 float shaft_distance = 4.72440945;
14
15 // ---------------- GLOBALS ----------------
16 // last time the sensor triggered
17 volatile uint16_t lastTime = 0;
18 // time between sensor pulses
19 volatile uint16_t period = 0;
20
21 // baseline period (used to detect changes later)
22 int16_t startPeriod = 0;
23
24 // flag to know if we've started traversing something
25 bool traverse = false;
26 // counts how many traversals happened
27 uint16_t traverseCount = 0;
28
29 // ---------------- RUN STATE (NEW) ----------------
30 // basically "is the robot running or not"
31 uint8_t runActive = 0;
32
33 // ---------------- ISR ----------------
34 // this interrupt fires when the pin on PC2 changes
35 ISR(PCINT1_vect)
36 {
37     // only act on HIGH signal
38     if (PINC & (1 << PC2))
39     {
40         int16_t now = TCNT1;      // grab current timer value
41         period = now - lastTime;  // compute time between pulses
42         lastTime = now;           // update last time
43     }
44 }
45
46 // ---------------- TIMING ----------------
47 // converts a distance into how long the motors should run
48 float time_delay_func(float distance){
49     float wheel_circ = 3.14159265f * wheel_diam; // circumference
50     float v = wheel_circ * (rpm / 60.0f);        // linear speed
51     float t = distance / v;                      // time = d / v
52     return t * 1000.0f;                          // convert to ms
```

```
53 }
54
55 // same idea but for turning a certain angle
56 float time_delay_func_ang(float angle_deg){
57     float angle_rad = angle_deg * 3.14159265f / 180.0f;
58     float wheel_circ = 3.14159265f * wheel_diam;
59
60     float v = wheel_circ * (rpm / 60.0f);
61     float omega = (2.0f * v) / shaft_distance;   // angular velocity
62
63     float t = angle_rad / omega;                 // time needed to turn
64     return t * 1000.0f;                          // ms
65 }
66
67 // safer delay so we don't overflow _delay_ms limits
68 void delay_ms_safe(float ms){
69     while (ms >= 1.0f) {
70         _delay_ms(1);
71         ms -= 1.0f;
72     }
73 }
74
75 // ---------------- MOTOR CONTROL ----------------
76 // shuts everything off
77 void stop_motors(){
78     PORTB &= ~((1 << 0) | (1 << 3));
79     PORTD &= ~((1 << 2) | (1 << 3));
80 }
81
82 // drives forward a given distance
83 void drive_forward(float distance){
84     // set motor direction pins for forward
85     PORTB = (PORTB & 0b11110111) | (1 << 3);
86     PORTD = (PORTD & 0b11110011) | (1 << 2);
87
88     // run motors for calculated time
89     delay_ms_safe(time_delay_func(distance));
90     stop_motors(); // then stop
91 }
92
93 // turns left by some degrees
94 void drive_left(float degrees){
95     // left turn config (one wheel forward, one backward)
96     PORTB = (PORTB & 0b11110111) | (1 << 3);
97     PORTD = (PORTD & 0b11110011) | (1 << 3);
98
99     // run for calculated turn time
100    delay_ms_safe(time_delay_func_ang(degrees));
101    stop_motors();
102 }
```

```
103
104 // ---------------- INIT ----------------
105 void setup()
106 {
107     // set motor pins as outputs
108     DDRB |= (1 << 0) | (1 << 3);
109     DDRD |= (1 << 2) | (1 << 3);
110
111     // sensor pin input
112     DDRC &= ~(1 << PC2);
113     // button input
114     DDRB &= ~(1 << PB2);
115
116     // enable pull-up on button
117     PORTB |= (1 << PB2);
118
119     // enable pin change interrupt for PC2
120     PCICR |= (1 << PCIE1);
121     PCMSK1 |= (1 << PCINT10);
122
123     // setup timer1 (no prescaler)
124     TCCR1A = 0;
125     TCCR1B = (1 << CS10);
126
127     sei(); // enable interrupts
128
129     _delay_ms(500); // wait a bit for things to stabilize
130     startPeriod = period; // capture baseline sensor timing
131 }
132
133 // ---------------- LOOP ----------------
134 void loop()
135 {
136     // ---------------- BUTTON START (TOGGLE RUN) ----------------
137     // if button is pressed (active low)
138     if (!(PINB & (1 << PB2)))
139     {
140         _delay_ms(30); // debounce
141         if (!(PINB & (1 << PB2)))
142         {
143             runActive = 1; // start running
144
145             while (!(PINB & (1 << PB2))); // wait until release
146             _delay_ms(30); // debounce again
147         }
148     }
149
150     // IF NOT RUNNING     DO NOTHING
151     if (!runActive) return;
152
```

```
153     int16_t p;
154
155     // safely copy the period (avoid interrupt conflict)
156     cli();
157     p = period;
158     sei();
159
160     // -------- COLOR CHANGE --------
161     // detects a moderate change in period (likely different surface/color)
162     if (p > 0 && abs(p - startPeriod) > 350 && abs(p - startPeriod) < 800)
163     {
164         drive_forward(1);   // move a bit
165         drive_left(75);     // turn (not quite 90 yet)
166         traverse = true;    // mark that we started traversing
167         drive_forward(1);   // move again
168     }
169
170     // detects a large change (probably a bigger marker)
171     if (p > 0 && abs(p - startPeriod) > 1200)
172     {
173         drive_left(180); // turn around
174         drive_forward(1);
175
176         if (traverse) traverseCount++; // count only if we were traversing
177     }
178
179     // after 2 traversals, do final move
180     if (traverseCount == 2)
181     {
182         drive_left(90);
183         drive_forward(12.0);
184
185         stop_motors();
186         runActive = 0;   // STOP AFTER FINISH
187     }
188
189     // default behavior: keep moving forward slowly
190     drive_forward(1.0);
191 }
```