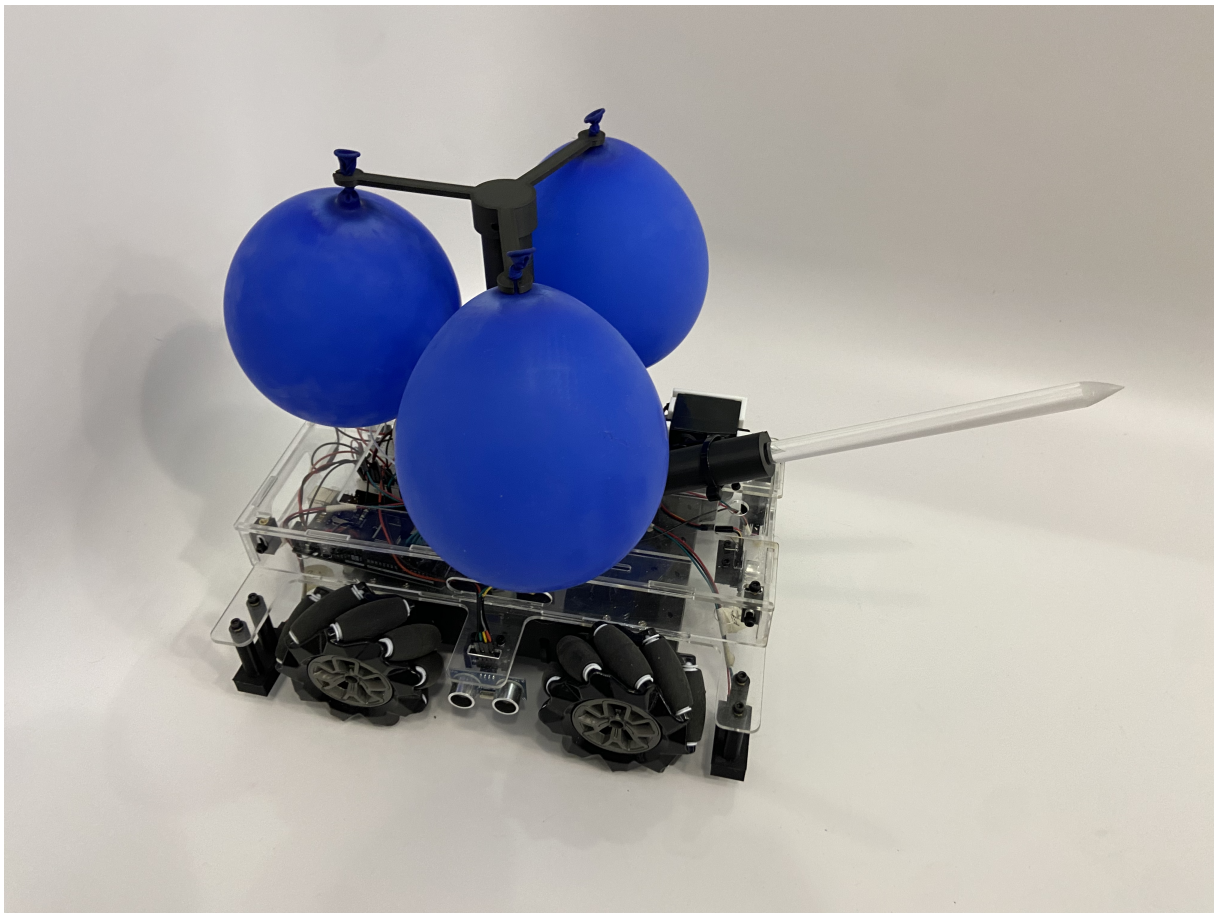
Autonomous Mario Kart Balloon Battle Robot

Grace Farson, Thomas Garcia, Kevin Kim,
Matt Sherman, Vaikunth Keshav Krishnan

Fall 2024 - Team Cobalt



1 Executive Summary

Mario Kart is a competitive multiplayer video game with various game modes. One of those game modes, Mario Kart Balloon Battle, features precise targeting where players attempt to pop three balloons on board opponents Karts in various specialized arenas. This mode emphasizes strategy while also offering a dynamic experience for all players. This game provides a complex platform for a mechatronics and robotics project. The goal of this project is to design an autonomous robot to transform Mario Kart Balloon Battle to a real world scenario and compete against other robots to win a competition.

2 Introduction

In order to successfully build an autonomous robot, integration between vision, movement, and weapons systems is essential. Vision, through the use of a Pixy camera, is imperative for navigating the arena to find and pop opponents balloons. The movement system, involving IR and Ultrasonic sensors, allows the robot to operate within the bounds of the arena and work in tandem with the vision system to find targets and avoid obstacles. The weapons system, a lance, provides the robot with a mechanism to pop balloons and acquire points.

Outlined below is the design report for an autonomous robot to successfully play a live action version of Mario Kart Balloon Battle to compete against other robotic opponents. This battle robot utilizes micro-controllers, sensors, and motors to optimize functionality and precision for competition. The technical approach for this robot encompasses the design requirements outlined in Mario Kart Balloon Battle. These requirements are based on the original game and are intended to create a challenge for a mechatronic system. The full design requirements are outlined in a provided competition document.

3 Technical Requirements

3.1 Robot Framework

There is a specific set of requirements the robots must meet. Each robot must be fully autonomous. Human players may not interact with or send signals to their robot once the round commences. Each robot is allowed a projectile or lance system. The robots must fit within a 14" x 12" footprint and should not be taller than 14" at their highest point. robots will be fit will three latex balloons, specifications are listed in the sections below. The robots cannot take any action percieved by a referee to cause intentional damage to the playing area or an opposing robot (other than balloons). If a robot causes damage to the obstacles or playing area, the team will be held responsible for repair or replacement. The robot must not damage the floor or walls in any way: no studded tires, digging into the floor, use of adhesives, emissions, or extension of obstructing parts, ballistic parts (except projectiles at balloons), electro-magnetic pulse, shocks, flames, chemicals, ect. Each robot must stay in one piece. Only the projectiles may seperate from the vehicle and the robot cannot alter the playing field in any way.

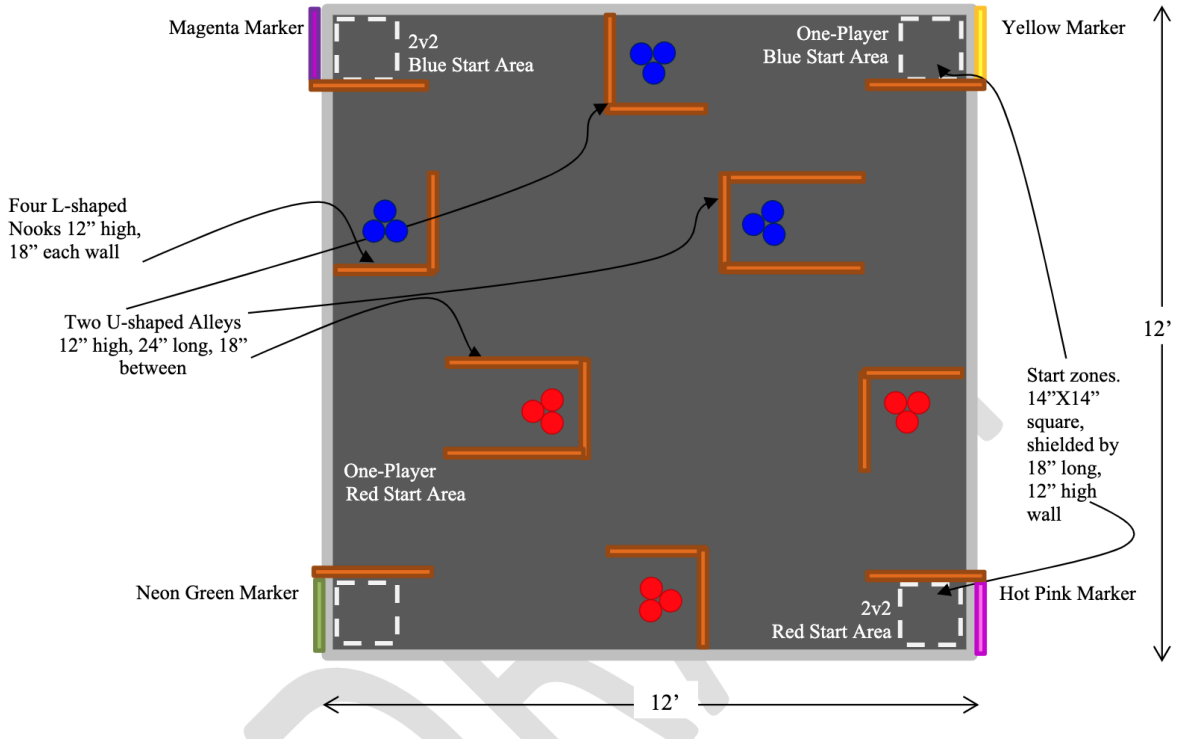
Each robot can have a maximum of three microprocessors, microcontrollers or similar devices on board. A Raspberry Pi, Tennsy, Arduino, mBed, Netburner, Altera DE2, or NI sbRio board would each count as one micro. Simple, small, cheap micros can be counted as ICs if they have limited functionality. A maximum of two cameras are allowed on board. A Pixy, Luxonis, or Pi Camera each count as one. IR sensor pairs do not count.

3.2 Balloons

To begin, the balloons onboard the robot have a select set of requirements that must be met and adhered by. Our robot will navigate a Balloon Battle field and attempt to pop opponent's balloon before its own balloons are popped. Each robot will carry three team balloons (either red or blue). Balloons are nominally 5" diameter latex balloons. Each game round will be 5 minutes with a potential 2 minute overtime if the score at the end of the round is tied between any players. The balloons must be inflated with air, roughly 6" high and 5" wide at its widest point. Each balloon must be firmly mounted on the robot so the nozzle is at the top and must move with the robot. If any balloons fall off the robot, the balloon is considered popped. All balloons must be fully inflated. The base of each balloon should be located exactly 6" from the floor and the balloons must be placed on the exterior of the robot. No parts of the robot can block any of the onboard balloons from sight. Lastly, the three balloons must be spaced evenly around the robot, when looking down from above.

3.3 Arena - Playing Field

The arena must also adhere by a set of requirements. The playing field will be a rectangular 12' x 12' puzzle piece foam exercise mat. The mat is 1/2" thick black modular foam flooring with the textured side up. The diagram below depicts the arena and is not to scale. Each starting area will have a wall that will be 12" high and 18" long. Each will be marked with a vertical colored plane 20" high for Pixy detection. To start, the robot will be placed in one of the four starting areas at a random orientation. Once the robot has been placed and the round commences, no member of the team may touch the robot until game play has ended. Spaced across the arena will be four L-shaped nooks. Each wall of the L will be 12" high and 18" long. Two U-shaped alleys will also be positioned in the arena. They will be 12" high with walls 18" apart and 24" deep. Stationary balloon crowns will be placed in two of the three available locations for each team color to offer extra points during game play. The perimeter of the playing field and all obstacles will be marked off with 2" wide white tape for sensor detection. If a robot crosses the perimeter such that the center point of the robot crosses over the tape, it counts as three popped balloons against the robot and the robot will be returned to its starting area.



3.4 Weapons System

Each robot can utilize a projectile and/or lance weapons system. The requirements for the lance are as follows as that is what our robot will utilize. A single lance can extend up to 8" from the 12" x 14" footprint of the robot to puncture balloons. The maximum cross sectional area of the lance must be 1.0 square inch and the angle of the lance must always be kept within +/- 30 degrees of the horizontal plane at a constant inclination and may not deviate laterally. The lance can be supported by the robot's central tower or elsewhere within the footprint of the robot. The lance support must be vertical and can also have a maximum cross-sectional area of 1.0 square inch and extend no higher than 6" from the ground. The lance can extend and/or retract as long as the above criteria is met. The lance must begin and end the contest within the 12" x 14" robot footprint. Each lance can have a single sharp tip, but a cover or sheath is recommended for safe testing. No other sharp corners, tips, exposed wiring or other potential puncturing objects may be exposed anywhere else on the robot.

3.5 Game Play and Scoring

As mentioned above, the competition will run in rounds of matches. The competition will be double-elimination style and each match will have a 5-minute time limit. Scoring will be based on the number of balloons popped. Each stationary balloon will be worth 30 points, and the on board balloons of opposing robots will be worth 60 points. If all balloons are popped at the end of the round, the robot whose balloon (stationary or onboard) was popped last will be the victor.

3.6 Budget

The budget for this competition, including the robot itself and all weapons systems, must be under 200 dollars. This budget encapsulates the chassis, lance, and any other accessories. Components implemented on board that were obtained without cost are considered free unless the system provides an obvious advantage not otherwise available to other teams.

4 Design Approach

To successfully compete in the Mario Kart Balloon Battle competition, integration between technical systems and design must be achieved. The main goal is to have a functional and precise popping mechanism dependent on the movement system. This would allow our robot to navigate around the arena and line up attacks with the movement of the system. Other systems on board the robot need to work independently and in tandem for optimal functionality. Independent systems makes coding easier, as we are able to isolate each system to its own lines of code rather than trying to code a combination of movement and vision. This approach will reduce the amount of power being supplied by the micro-controller.

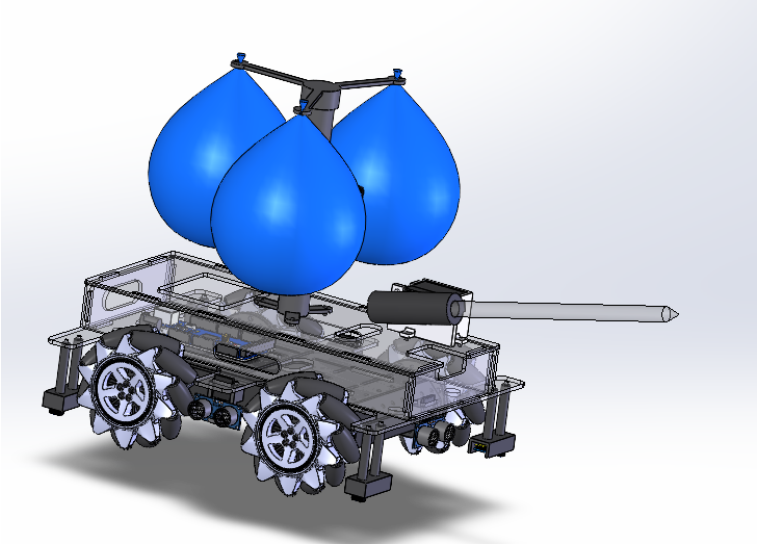
Another design approach was the type of wheels our robot would utilize to move around the playing field. We determined mecanum wheels would be optimal for this specific competition. Unlike traditional wheels, mecanum wheels are highly maneuverable and flexible. These features are ideal for navigating dynamic environments and tight spaces. Mecanum wheels feature rollers mounted at 45 degree angles around the circumference of the wheel to provide omnidirectional movement. This allows the robot to move both laterally, diagonally, and in a rotational motion with zero turn radius. According to the playing field, robots will begin in a tight corner of the arena in any orientation. Zero-turn radius allows the robot to navigate these tight conditions. The ability of the robot to move diagonally and seamlessly makes path planning and navigation simple. Specifically, mecanum wheels are beneficial for maneuvering around obstacles fluidly without having to reorient the robots position. The playing field requires the robot to make frequent changes in direction with minimal space availability, making mecanum wheels an optimal choice for a compact design. Lastly, system integration and control is fairly simple with the appropriate algorithms and controllers. Overall, mecanum wheels provide a competitive edge when compared to traditional methods because of their agility, precision, and increased mobility.

Choice of weapons system was a crucial design consideration for the robot as well. Given the choice between a projectile system or a lance, our team decided to simplify our design and minimize risk while still effectively popping balloons through the use of a lance. Unlike a projectile system, we didn't need to worry about reloading, jamming mechanism, or complex system integration. A lance provides us with a sound weapon to knock out our competition while maintaining simplicity in both integration and design.

Determining which electronics we would have on board our robot was the last major design consideration. Based on constraints like availability, design, and budget, our team decided to use an Arduino for the micro controller system. Arduino's have been readily used by many, making reference code easily accessible. They also provide simple integration between sensors and cameras. For our vision system, we choose to use the Pixy Cam due to

easy integration with the Arduino and simple controls for coding. Both of these systems did not impact our budget as they were available for use through the school. As for our sensors, we choose to integrate four IR sensors along with three Ultrasonic sensors for navigation.

5 Design



5.1 Electronics

This section encompasses electronics design including items such as Arduino Mega, infrared (IR) sensors, ultrasonic distance sensors, camera, and motors/drivers. In order to utilize the Arduino, code must be written using Arduino IDE. The sensors are integrated and controlled with the Arduino using the same coding language. The four motors on board the robot are directly connected to the power source and use a motor driver controlled by the Arduino, based on voltage requirements. The camera is controlled though SPI pins on the Arduino.

5.1.1 Micro-controller

Our robot uses one Arduino Mega 2560. The Arduino Mega 2560 is a micro-controller board featuring 54 digital input/output pins, 16 analong inputs, and 4 harware serial ports (UART). The micro-controller reads signals from sensors, processes the data, and controls various devices like motors and servos. Code is uploaded onto the Arduino Mega using a USB connection. With a multitude of pins and high memory capacity, the Arduino Mega is perfect for complex projects requiring the integration between various systems and sensors, such as our Mario Kart Balloon Battle project.

5.1.2 Sensors

The sensors were designed with the goal of navigation and obstacle avoidance in mind. They will provide autonomous navigation for the entirety of the game play. Our robot utilizes 3 ultrasonic sensors at the front and sides and four IR sensors at each corner of the chassis. Ultrasonic sensors transmit sound waves and assess the time the sound wave takes to return to measure distance. We used this sensor for obstacle avoidance when navigating the playing field. Specifically, we utilized three HC-SR04 sensors on three sides of our robot due to their easy integration with the Arduino and how cost effective they are. Since the robot never moves backwards, we did not feel the need for an ultrasonic sensor on the back of the chassis. Therefore, if an obstacle or opponent is within a set distance from our robot, we will turn away from the object to avoid collision. The HC-SR04 wiring to the Arduino Mega is shown in the table below.

Arduino Pin	HC-SR04 1 Pin	HC-SR04 2 Pin	HC-SR04 3 Pin
5V	Vcc	Vcc	Vcc
38, 40, 42	Trigger	Trigger	Trigger
39, 41, 43	Echo	Echo	Echo
GND	GND	GND	GND

We utilized four IR sensors for line detection during game play. IR sensors detect infrared radiation emitted by reflective surfaces. Their ability to detect objects, measure distances, and sense changes made them highly desirable on board our robot. For our project, the white tape and the foam of the arena reflect differently, allowing our robot to detect the boundary and obstacles placed throughout the arena. We specifically used four TCRT5000 sensors because of their obtain-ability and simplicity. Using four sensors will help our robot maneuver out of tight spaces and corners. The location of the sensors was picked to make sure all parts of the robot stay in bounds during game play. The TCRT5000 wiring to the Arduino Mega is shown in the table below.

Arduino Pin	TCRT5000 Sensors	TCRT5000 Pin
5V	Front Left, Front Right, Back Left, Back Right	Vcc
GND	Front Left, Front Right, Back Left, Back Right	GND
A11	Front Left	A0
A10	Front Right	A0
A9	Back Left	A0
A8	Back Right	A0

5.1.3 Camera

A camera is necessary for detecting balloons during the competition. We needed an advanced vision system to identify either red or blue colors as the sensors above do not have that capability. We choose to integrate a PixyCam into our design. The Pixy offers easy integration with the Arduino, using SPI or I2C, and is relatively small and easy to mount. The graphical user interface (GUI) provided by the PixyMon application allowed us to train and monitor colored objects in real time. Live video feed is displayed with detected objected highlighted. To train the camera, we were able to select and assign various objects and colors (in our case balloons) to detection signatures through the interface. Pixy cameras are sensitive to changes in light throughout the day and the GUI allowed us to set the exposure, brightness , and color thresholds for the time of day we would be competing. Simple integration between the Arduino and Pixy made this camera the optimal choice for our project.

5.1.4 Motors and Drivers

Motors and Drivers are an integral part of the design project for this robot. We chose to implement four general DC motors for the movement of our system. We also utilized a Servo motor for our weapons system. The motors were driven by the Arduino through the use of two motor drivers. Two 8V DC motors were placed on the right side of the chassis and Two on the left side. These motors were in charge of the movement of the mecanum wheels. The specific motor drivers we used were two SparkFun ROB-14451 drivers. We were able to control these motors through the motor driver by programming forwards and rotational movement dictated by the sensors on board the robot. The ROB-14451 wiring to the Arduino Mega is shown in the table below.

SparkFun ROB-14451 1 Pin	SparkFun ROB-14451 2 Pin	Arduino Connection
Vm	Vm	11.1 V battery
Vcc	Vcc	5V
GND	GND	GND
A01	A01	Motor 1/3 - positive
A02	A02	Motor 1/3 - negative
B02	B02	Motor 2/4 - positive
B01	B01	Motor 2/4 - negative
PWMA	PWMA	Pin: 4 & 2
AI2	AI2	Pin: 33 & 25
AI1	AI1	Pin: 32 & 24
STBY	STBY	-
BI1	BI1	Pin: 34 & 26
BI2	BI2	Pin: 35 & 27
PWMB	PWMB	Pin: 5 & 2

5.1.5 Weapons System - Servo Motor

To fix our lance at 30 degrees once the game commences, we choose to utilize a Servo motor. Specifically, we used the Parallax Standard Servo (900-00005). This electromechanical device allowed us to rotate our lance to a specified angle within 0 to 180 degrees. The servo consists of a motor, potentiometer, and control circuit. To control its position, we adjust the duty cycle of the pulse-width modulation (PWM) to obtain a precise angle. The Parallax Standard Servo (900-00005) wiring to the Arduino Mega is shown in the table below.

Parallax Standard Servo (900-00005) Pin	Arduino Pin Connection
GND	GND
Vcc	5V
PWM	Pin 8

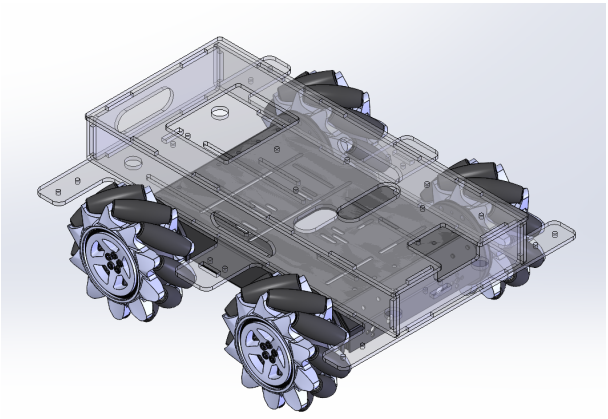
5.2 Power System

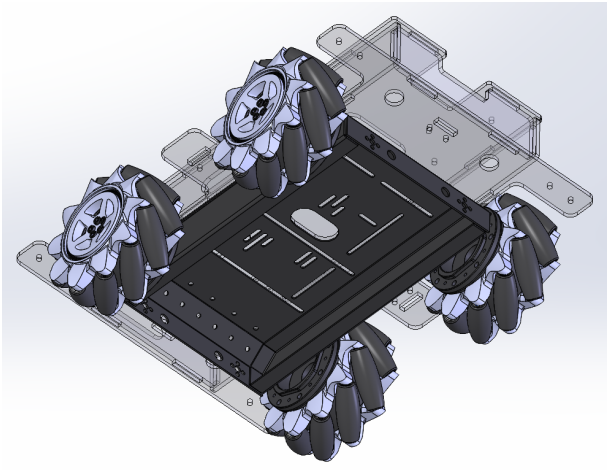
In order to power the system, we used a 11.1V LiPo (Lithium-Polymer) battery. The LiPo battery is not only lightweight but has a high capacity which makes it optimal for being the main power source on board our system. The battery's high-capacity energy source allows us to power multiple components. For this project, the battery supplies power to four motors

driven by two motor drivers and an Arduino that processes data from IR and ultrasonic sensors. Lastly, the LiPo powers a servo motor to control the lance mechanism for precise balloon popping. The voltage of the LiPo battery is sufficient to reliably and effectively power all parts of the system.

5.3 Robot Chassis Design

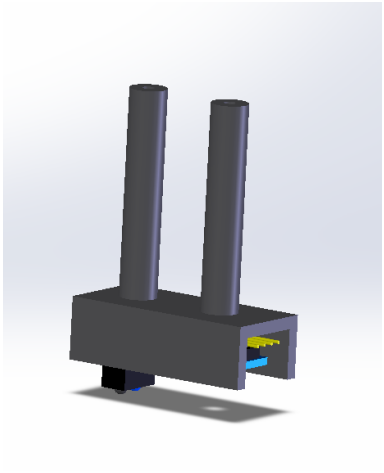
The chassis is made out of acrylic and forms a 14" x 12" rectangular box. The box sits on top of a metal frame with four shafts for wheels. The frame is supported by four mecanum wheels which are each driven by DC motors and controlled by two motor drivers and an Arduino Mega. The inside of the chassis was designed to safely and securely hold all of the components of the robot without interference from other opponents. Each mecanum wheel is mounted by a motor shaft and then attached to the metal frame in an even fashion to distribute weight.

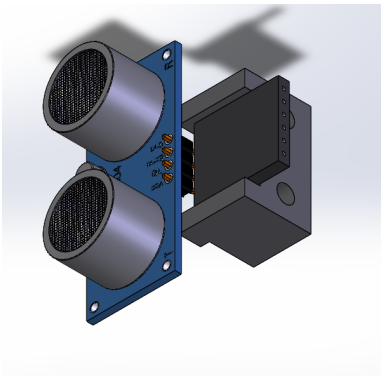




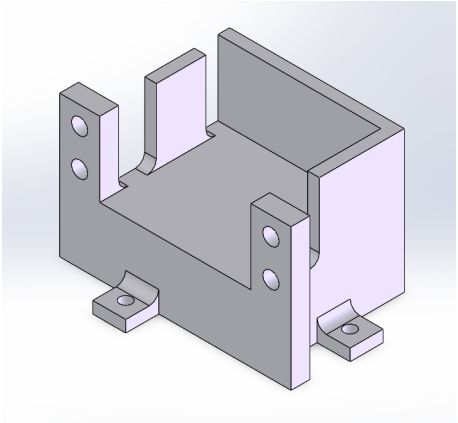
The sensors are mounted to the chassis in order to achieve optimal sensing and maneuverability while keeping the design as simple as possible. The four IR sensors are mounted at each corner of the chassis in front of a respective wheel. This allows the sensor to detect the boundary lines and obstacle lines before the robot crosses the boundary. Each IR sensor is mounted to a 3D printed mount for secure and precise sensing.

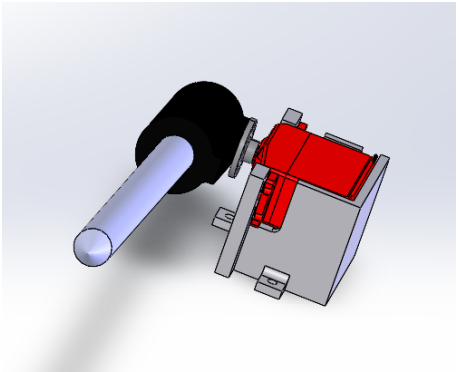
Three ultrasonic sensors are also mounted on the chassis. One is located at the front, and the two others are on the sides of the chassis. The mounts for the ultrasonic sensors were designed to securely fasten the sensor to the side of the robot while maintaining protection from other projectiles and opponents.



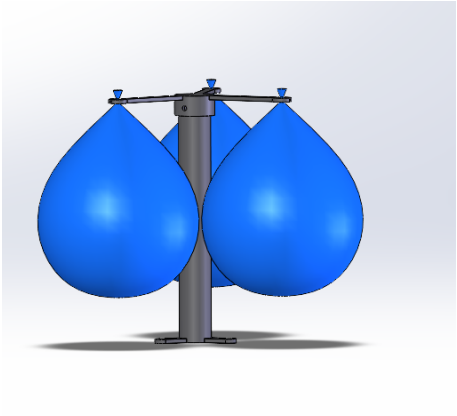


The weapons system we chose for our project is a lance. We constructed the lance out of acrylic in the form of a long cylinder with a precise point at the end for puncturing balloons. The lance is attached to a servo motor on top of the chassis by a 3D printed mount. This mount allowed us to secure the lance to the servo while also securing the servo to the top of the chassis. The servo motor allows the lance to begin in an upwards position (90 degrees) and then rotate to 30 degrees during the competition. This mechanism is an essential part of the competition as it provides our robot with the ability to puncture balloons. Acrylic was used because of its lightweight and durable material.





A balloon mount was added to the center of the chassis with the goal of holding three balloons and providing a mount for the Pixy camera. The mount features a 3D printed design to hold the nozzle of each balloon securely so that they are equidistant from each other when looking down at the chassis and don't fall off during game play. The Pixy camera is mounted on the balloon mount to accurately detect balloons without obstructions while maintaining stability while the robot is in motion.



Inside the acrylic box of the chassis is the Arduino Mega held in place using Velcro and zip ties. The power supply is mounted on the bottom of the metal frame with zip ties to ensure no opponent can run into it. Overall, the chassis fits the requirement of the 14" x 12" footprint while securing and protecting all system components. The goal of the design was to maintain functionality while emphasizing protection and aesthetics. The lightweight materials used, such as acrylic and PLA, minimize the weight of the robot while maintaining strength and performance.

5.4 Code

The code is designed using a class hierarchy in Arduino, prioritizing efficiency and simplifying the debugging process. Each utility associated with the robot is implemented in its dedicated C++ file, while the primary logic integrating these utilities resides in the robotCode.ino file. This modular approach significantly streamlines both debugging and maintenance. Because of the class structure, the number of lines written was decreased by simply creating objects for the sensors and motors rather than defining functionality for all components explicitly.

5.4.1 IR Sensors

The IR sensors' functionality is handled through the IRsensor class (Appendix 1 Fig. 4). This class includes a constructor to create an IRsensor object which sets the corresponding pin mode on the Arduino to enable receiving of the sensor data. The two methods implemented in this class are read() and tooClose(). The read() function returns the analog value received from the Arduino. The tooClose() function returns a boolean variable signaling whether the sensor is detecting black or white, which was determined through testing. The tooClose() method improved the main robot code because the threshold value only had to be changed in one place during calibration in the grid.

5.4.2 Ultrasonic Sensors

The ultrasonic sensors' functionality is implemented in the Ultrasonic class (Appendix 1 Fig. 3). This class includes a constructor to set the pins and pin modes for each ultrasonic sensor. The key method that was implemented in this class is the getDistance() function, which returns a float variable of the distance detected by the sensor in inches. This works

by sending a pulse from the sensor and measuring the time the signal takes to return. The getDistance() function was used throughout the main robot code to evaluate what the behavior of the robot should be at different points in the grid.

5.4.3 Motors

The motors' functionality is implemented through the Motor class (Appendix 1 Fig. 5). This class contains a constructor that sets the three pins used to control the direction and speed of the motors. It also contains three functions forward(), backward(), and stop() to move the motors in a certain direction at a certain speed or to stop the motors. These functions were included in this class rather than directly implemented into the main code to simplify the more complex methods implemented in the main robot code.

5.4.4 Robot

The code comes together to define the robot's functionality in the Robot class. This class includes objects of both sensor classes, the Motor class, the Servo class, and the Pixy class to define all the components on the robot and to provide access to their functionality (Appendix 1 Fig. 6). It also includes functions to set all of these objects, as well as other crucial variables such as the color signatures on the Pixy, from the main .ino file. After defining the functionality of these functions, more functions to implement the general movement patterns of the robot are defined. These include moveForward(), moveBackward(), moveLeft(), moveRight(), rotateCW(), and rotateCCW(). The movement functions build on the motor classes to integrate the four motors together to move the robot. These functions were important to define separately from the other functions to simplify the code.

The three other functions implemented in the Robot class are attack(), startup(), and getActive(). The attack() function is called when an opponent's balloon is detected (Appendix 2). It then executes until one of the IR sensors senses a line of tape, the front sensor detects an object within 2 inches, or the opponent is no longer visible. The logic in this function runs a loop the repeatedly checks the aforementioned sensors. Inside the loop, the position of the opponent is retrieved from the Pixy. The position is then normalized between -1 and 1. If the position is negative, then the opponent is left of center of the robot, and vice versa. A tolerance of 0.2 is used to determine whether the robot should turn towards the opponent or move forward towards the opponent.

The startup() function is used to get the robot out of the starting position in the grid at the beginning of the round (Appendix 3). The robot can be set in any orientation in the starting area. Based on the sensor readings, the function determines which orientation the robot is in, and adjusts the movement accordingly. This allowed us to start the robot at different orientations in different rounds depending on the starting location in the grid, and prevented the robot from getting stuck in the small start box. Although this function only executes once rather than in a continuous loop, there are several loops inside it to avoid having delays where the robot is moving without receiving sensor input.

The getActive() function implements the logic that the robot runs on for the duration of each round after the startup() function executes (Appendix 4). It starts by checking that the robot is still on. If not, it shuts the robot off and stops execution. The current

distance reading of all the sensors is retrieved to start. A series of if-else statements are then executed in order of importance. The most important sensor readings are from the front two IR sensors so that the robot does not go out of the grid and lose the round. These two cases are handled, with several other conditions ingrained in them. Next, the case where an object is too close to the front is handled so that the robot does not run into walls or other opponents. Next, the left and right distance are handled. Finally, if none of the other sensors detect a hazardous condition, the robot checks if an opponent is detected, and if so, it calls the attack function to attempt to puncture the balloon. If none of these conditions are true, the default behavior is to move forward. Finally, there is a check to ensure that the robot does not get stuck in the same spot (against a wall or in a corner) where the sensors cannot detect any useful information. This is handled with a condition that says if the sensor readings have remained the same for more than five seconds, the robot will move backward.

The getActive() function avoids using delays in as many places as possible. Instead, it is called in a repeated fashion during the robot's exectuion and executes quickly, telling the robot the behavior is should take for the next 40 ms. By taking this approach, the sensor readings are always updated so the robot can always decide what the best action to take is at any time. This also helped the robot move smoothly around the grid.

5.4.5 Main .ino Code

To upload the code to the Arduino, the functionality must be integrated into a .ino file. This file uses all the classes described above to inherit the functionality of all components and of the robot. However, this code is very bare bone since all the robot's functionality is defined in the classes. The only functionality implemented in this file is for the button that controls whether the robot is on or off. The button uses an interrupt subroutine (ISR) in order to always be detecting for a button press. This avoids the button sensing from being blocked by the other functions since the Arduino is a single-thread processor. Other than the button, this code is only responsible for holding all the variables for the button pin numbers and creating instances of the other classes (Appendix 1 Fig. 1 and 2).

6 Summary and Lessons Learned

Not only did our robot successfully navigate and perceive the arena but also successfully beat out all of our competitors to win the competition. The robot was able to detect obstacles, navigate around obstacles, stay in bounds, engage the lance, attack opponents, and pop balloons. There were a few moments the robot was stuck in a corner on the playing field but in most cases the logic we fed through the algorithm on board the Arduino was able to successfully maneuver the robot out of these situations. Varying light conditions and clothing worn by people around the arena sometimes influenced what the robot was detecting. In the future, these discrepancies could be resolved through more expensive equipment, changes to the coding logic, and better camera training and calibration to improve sensing.

Another challenge was other opponents ability to knock our robot out of bounds when it was actively attacking or stuck at a boundary for a significant period of time. A potential solution could involve adding another weapons system, changing the distance sensing and

logic for the ultrasonic sensors, and improving the logic to make sure our robot can navigate out of complex areas such as tight spaces and corners.

Overall, the robot demonstrated superior performance when compared to opponents. Wiring and connection was sound and secure, navigation and perception was successful, and our sensing and lance mechanism was able to detect balloons and accurately pop them to gain points during the competition. Throughout the project, we learned valuable lessons about the importance of calibrating sensors, coding logic, and maintaining secure connections and integration between all systems. Future work can use these lessons to improve robustness and overall performance.

We have two main suggestions for future students in the class. The first is to structure the code in a way that makes it easy to write and keep track of as the code base grows and more and more functionality is implemented. By using classes, we were able to make the code significantly easier to understand, write, and debug. This may be one of the key reasons we were able to win the run off. Secondly, the most important takeaway, is to not underestimate the importance of testing. Because we started testing the robot's functionality early, we were able to catch many errors that the other teams did not. Even more testing would have helped us, as our robot still experienced some edge cases during the run off that we had not seen before.

References

[1] Reamon, D. (2024). *MCEN 4115/5115 Mechatronics: Mario Kart Balloon Battle Project Guidelines*. University of Colorado Boulder.

7 Appendix 1

```
#include "motor.h"
#include "IRsensor.h"
#include "robot.h"
#include "Ultrasonic.h"

// motor pins
const int frontLeftMotorPin1 = 27, frontLeftMotorPin2 = 26, frontLeftMotorSpeedPin = 3;
const int frontRightMotorPin1 = 25, frontRightMotorPin2 = 24, frontRightMotorSpeedPin = 2;
const int rearLeftMotorPin1 = 35, rearLeftMotorPin2 = 34, rearLeftMotorSpeedPin = 5;
const int rearRightMotorPin1 = 33, rearRightMotorPin2 = 32, rearRightMotorSpeedPin = 4;
const int lancePin = 8;
// sensor pins
const int frontLeftIRPin = A11, frontRightIRPin = A10, rearLeftIRPin = A9, rearRightIRPin = A8;
const int frontUltrasonicTrigPin = 38, frontUltrasonicEchoPin = 39;
const int leftUltrasonicTrigPin = 42, leftUltrasonicEchoPin = 43;
const int rightUltrasonicTrigPin = 40, rightUltrasonicEchoPin = 41;
// button pin
const int buttonPin = 18;
// team color signatures
const int teamSig = 1, opponentSig = 2;
// sensor thresholds
const int IR_tapeThreshold = 600, ultrasonic_distanceThreshold = 200;

/*******************************************************
  this section is for the actual robot code
*******************************************************/

// create the robot object
Robot robot;

// ISR for handling the button press
void buttonISR() {
  robot.turnOnOff();
}
```

Figure 1: Main Code

```ino
void setup() {
  // set up the robot
  robot.setLance(lancePin);
  robot.setStartButton(buttonPin);
  robot.setFrontLeftMotor(frontLeftMotorPin1, frontLeftMotorPin2, frontLeftMotorSpeedPin);
  robot.setFrontRightMotor(frontRightMotorPin1, frontRightMotorPin2, frontRightMotorSpeedPin);
  robot.setRearLeftMotor(rearLeftMotorPin1, rearLeftMotorPin2, rearLeftMotorSpeedPin);
  robot.setRearRightMotor(rearRightMotorPin1, rearRightMotorPin2, rearRightMotorSpeedPin);
  robot.setFrontLeftIR(frontLeftIRPin);
  robot.setFrontRightIR(frontRightIRPin);
  robot.setRearLeftIR(rearLeftIRPin);
  robot.setRearRightIR(rearRightIRPin);
  robot.setFrontUltrasonic(frontUltrasonicTrigPin, frontUltrasonicEchoPin);
  robot.setLeftUltrasonic(leftUltrasonicTrigPin, leftUltrasonicEchoPin);
  robot.setRightUltrasonic(rightUltrasonicTrigPin, rightUltrasonicEchoPin);
  robot.setPixy();
  robot.setSignatures(teamSig, opponentSig);
  // robot.buttonCounter = 0;
  robot.on = false;

  // use an interrupt for the button pin
  attachInterrupt(digitalPinToInterrupt(buttonPin), buttonISR, RISING);
  Serial.begin(9600);


void loop() {
  if (robot.on) {
    robot.startup();
  }
}
```

Figure 2: Main Code

```ino
#include "Ultrasonic.h"
#include <Arduino.h>

Ultrasonic::Ultrasonic() : trigPin(-1), echoPin(-2) {}

Ultrasonic::Ultrasonic(int trigPinNo, int echoPinNo) : trigPin(trigPinNo), echoPin(echoPinNo) {
  pinMode(trigPin, OUTPUT);
  pinMode(echoPin, INPUT);
}

float Ultrasonic::getDistance() {
  float echoTime;                 //variable to store the time it takes for a ping to bounce off an object
  float calculatedDistance;       //variable to store the distance calculated from the echo time

  //send out an ultrasonic pulse that's 10ms long
  digitalWrite(trigPin, HIGH);
  delayMicroseconds(10);
  digitalWrite(trigPin, LOW);

  echoTime = pulseIn(echoPin, HIGH);      //use the pulsein command to see how long it takes for the
                                          //pulse to bounce back to the sensor

  calculatedDistance = echoTime / 148.0;  //calculate the distance of the object that reflected the pulse (half the bounce time multiplied by the speed of sound)

  return calculatedDistance;
}
```

Figure 3: Ultrasonic Sensor Code

```cpp
#include <Arduino.h>
#include "IRsensor.h"

IRsensor::IRsensor() : pin(-1) {}

IRsensor::IRsensor(int pinNo) : pin(pinNo) {
  pinMode(pin, INPUT);
}

int IRsensor::read() {
  return analogRead(pin);
}

bool IRsensor::isOnTape() {
  return analogRead(pin) < 300;
}
```

Figure 4: IR Sensor Code

```cpp
#include <Arduino.h>
#include "motor.h"

Motor::Motor() : dir1(-1), dir2(-2), speedPin(-3) {}

Motor::Motor(int pin1, int pin2, int pin3) : dir1(pin1), dir2(pin2), speedPin(pin3) {
  pinMode(dir1, OUTPUT);
  pinMode(dir2, OUTPUT);
  pinMode(speedPin, OUTPUT);
}

void Motor::forward(int speed) {
  digitalWrite(dir1, HIGH);
  digitalWrite(dir2, LOW);
  analogWrite(speedPin, speed);
}

void Motor::backward(int speed) {
  digitalWrite(dir1, LOW);
  digitalWrite(dir2, HIGH);
  analogWrite(speedPin, speed);
}

void Motor::stop() {
  digitalWrite(dir1, LOW);
  digitalWrite(dir2, LOW);
  analogWrite(speedPin, 0);
}
```

Figure 5: Motor Code

```
#include "motor.h"
#include "IRsensor.h"
#include "Ultrasonic.h"
#include <Pixy2.h>
#include <Servo.h>

class Robot {
  private:
    // color signature of robot's balloons
    int teamSig;
    // color signature of opponent's balloons
    int opponentSig;
    // the value of the threshold for the IR sensor detecting tape color
    const int tapeThreshold = 500;
    // the value of the ultrasonic sensor reading when an object is too close
    const int distanceThreshold = 2;
    // the value of the ultrasonic sensor reading when a front wall is detected
    const int frontWallDistanceThreshold = 9;
    // pin # of lance servo
    int lancePin;
    // pin # of the start button
    int buttonPin;
    // Motor object for each wheel's motor (see Motor.h for functionality)
    Motor frontLeftMotor;
    Motor frontRightMotor;
    Motor rearLeftMotor;
    Motor rearRightMotor;
    // IRsensor object for each IRsensor (see IRsensor.h for functionality)
    IRsensor frontLeftIR;
    IRsensor frontRightIR;
    IRsensor rearLeftIR;
    IRsensor rearRightIR;
    // Ultrasonic sensor objects
    Ultrasonic frontUltrasonic;
    Ultrasonic leftUltrasonic;
    Ultrasonic rightUltrasonic;
    // Pixy camera object
    Pixy2 pixy;
    // Servo object for the lance
    Servo lance;
    // function to normalize the position of balloons detected by Pixy cam
    double mapPosition(int x);
```

Figure 6: Robot class header file

```
public:
  // default constructor
  Robot();
  // set the lance pin #
  void setLance(int pin);
  // set the start button pin #
  void setStartButton(int pin);
  // set the Motors
  void setFrontLeftMotor(int pin1, int pin2, int speedPin);
  void setFrontRightMotor(int pin1, int pin2, int speedPin);
  void setRearLeftMotor(int pin1, int pin2, int speedPin);
  void setRearRightMotor(int pin1, int pin2, int speedPin);
  // set the sensors
  void setFrontLeftIR(int pin);
  void setFrontRightIR(int pin);
  void setRearLeftIR(int pin);
  void setRearRightIR(int pin);
  void setFrontUltrasonic(int trigPin, int echoPin);
  void setLeftUltrasonic(int trigPin, int echoPin);
  void setRightUltrasonic(int trigPin, int echoPin);
  // set Pixy camera
  void setPixy();
  // set team color signatures
  void setSignatures(int team, int opponent);

  volatile bool on;
  // handles the robot state (on or off)
  void turnOnOff();
  // general movement
  void stop();
  void moveForward(int speed);
  void moveBackward(int speed);
  void moveLeft(int speed);
  void moveRight(int speed);
  void rotateCW(int speed);
  void rotateCCW(int speed);
  // fighting
  // void findOpponent();
  void attack();
  // robot initialization in grid
  void startup();
  // handle sensor readings
  void getActive();

  // test functions
  void lanceAttack();
  void testMotors();
  void testMovement();
  void testIRSensors();
  void testUltrasonicSensors();
  void testPixy();
```

Figure 7: Robot class header file cont.

8 Appendix 2

```
void Robot::attack() {
  float frontDistance = frontUltrasonic.getDistance();
  while (!frontLeftIR.isOnTape() && !frontRightIR.isOnTape() && frontDistance > 2) {
    if (!on) {
      stop();
      lance.write(90);
      return;
    }
    pixy.ccc.getBlocks(true, opponentSig);
    if (!pixy.ccc.numBlocks) return;
    double location = mapPosition(pixy.ccc.blocks[0].m_x);
    if (location < -0.20) {
      if (frontDistance < 6) {
        moveBackward(75);
        delay(100);
      }
      rotateCCW(50);
      delay(5);
    }
    else if (location > 0.20) {
      if (frontDistance < 9) {
        moveBackward(75);
        delay(100);
      }
      rotateCW(50);
      delay(5);
    }
    else {
      moveForward(75);
      delay(5);
    }
    // stop();
    delay(5);
    frontDistance = frontUltrasonic.getDistance();
  }
}
```

9 Appendix 3

```
void Robot::startup() {
  float frontDistance = frontUltrasonic.getDistance();
  float leftDistance = leftUltrasonic.getDistance();
  float rightDistance = rightUltrasonic.getDistance();

  if (frontDistance > 10) {
    while (rightDistance < 10) {
      if (!on) {
        stop();
        return;
      }
      if (frontLeftIR.isOnTape()) {
        rotateCW(50);
      }
      else if (frontRightIR.isOnTape()) {
        rotateCCW(50);
      }
      else {
        moveForward(50);
      }
      delay(50);
      rightDistance = rightUltrasonic.getDistance();
    }
      moveForward(50);
      delay(1000);
  }
  else if (leftDistance > 10) {
    while (frontDistance < 10) {
      if (!on) {
        stop();
        return;
      }
      if (frontLeftIR.isOnTape()) {
        rotateCCW(50);
      }
      else if (rearLeftIR.isOnTape()) {
        rotateCW(50);
      }
      else {
        moveLeft(75);
      }
      delay(50);
      frontDistance = frontUltrasonic.getDistance();
    }
      moveLeft(75);
      delay(1000);
  }
```

```
else if (rightDistance > 10) {
  while (frontDistance < 10) {
    if (!on) {
      stop();
      return;
    }
    if (frontRightIR.isOnTape()) {
      rotateCW(50);
    }
    else if (rearRightIR.isOnTape()) {
      rotateCCW(50);
    }
    else {
      moveRight(75);
    }
    delay(50);
    frontDistance = frontUltrasonic.getDistance();
  }
    moveRight(75);
    delay(1000);
}
else {
  while (leftDistance < 10) {
    if (!on) {
      stop();
      return;
    }
    if (rearLeftIR.isOnTape()) {
      rotateCCW(50);
    }
    else if (rearRightIR.isOnTape()) {
      rotateCW(50);
    }
    else {
      moveBackward(50);
    }
    delay(50);
    leftDistance = leftUltrasonic.getDistance();
  }
    moveBackward(50);
    delay(1000);
    rotateCCW(50);
    delay(1000);
}
```

10 Appendix 4

```
void Robot::getActive() {
  if (!on) {
    stop();
    lance.write(90);
    return;
  }
  pixy.ccc.getBlocks(true, opponentSig);
  int opponentLocation = -1;
  bool opponentAhead = false;
  if (pixy.ccc.numBlocks) {
    opponentLocation = pixy.ccc.blocks[0].m_x;
    if (opponentLocation > 100 && opponentLocation < 210) {
      opponentAhead = true;
    }
  }

  float frontDistance = frontUltrasonic.getDistance();
  float leftDistance = leftUltrasonic.getDistance();
  float rightDistance = rightUltrasonic.getDistance();

  static int index = 1;
  bool FR_IR = frontRightIR.isOnTape();
  bool FL_IR = frontLeftIR.isOnTape();
  bool RR_IR = rearRightIR.isOnTape();
  bool RL_IR = rearLeftIR.isOnTape();

  const int speed = 70;

  static bool dirFront = true;

  if (frontRightIR.isOnTape()) {
    if (frontLeftIR.isOnTape()) {
      moveBackward(speed);
      delay(700);
      rotateCW(speed);
      delay(300);
    }

    else if (rearRightIR.isOnTape()) {
      moveLeft(speed);
      delay(200);
      rotateCCW(speed);
      delay(40);
    }

    else if (rearLeftIR.isOnTape()) {
      lance.write(90);
      rotateCCW(speed);
      delay(300);
      moveBackward(speed);
      delay(1800);
      rotateCCW(speed);
      delay(750);
      lance.write(18);
    }
    else {
      if (frontDistance > frontWallDistanceThreshold) {
        rotateCCW(speed);
      }
      else {
        lance.write(90);
        while (frontDistance < frontWallDistanceThreshold) {
          if (leftDistance < 5 || rightDistance < 5) break;
          rotateCCW(speed);
          delay(15);
          frontDistance = frontUltrasonic.getDistance();
          leftDistance = leftUltrasonic.getDistance();
          rightDistance = rightUltrasonic.getDistance();
        }
        lance.write(18);
      }
    }
    dirFront = !dirFront;
  }
  else if (frontLeftIR.isOnTape()) {
    if (rearLeftIR.isOnTape()) {
      lance.write(90);
      moveRight(speed);
      delay(700);
      rotateCW(speed);
      delay(300);
      lance.write(18);
    }
    else if (rearRightIR.isOnTape()) {
      lance.write(90);
      rotateCCW(speed);
      delay(300);
      moveBackward(speed);
      delay(1800);
      rotateCCW(speed);
      delay(750);
      lance.write(18);
    }
    else {
      if (frontDistance > frontWallDistanceThreshold) {
        rotateCW(speed);
      }
      else {
        lance.write(90);
        while (frontDistance < frontWallDistanceThreshold) {
          if (leftDistance < 5 || rightDistance < 5) break;
          rotateCW(speed);
          delay(15);
          frontDistance = frontUltrasonic.getDistance();
          leftDistance = leftUltrasonic.getDistance();
          rightDistance = rightUltrasonic.getDistance();
        }
        lance.write(18);
      }
    }
    dirFront = !dirFront;
  }
```

```
else if (frontDistance < frontWallDistanceThreshold) {
    if (leftDistance < 9) {
      if (rightDistance < 9) {
        lance.write(90);
        unsigned long t = millis();
        while (frontDistance < 9 || rightDistance < 9) {
          if (frontRightIR.isOnTape() || frontLeftIR.isOnTape()) {
            moveBackward(speed);
          }
          else {
            rotateCW(speed);
          }
          delay(50);
          frontDistance = frontUltrasonic.getDistance();
          rightDistance = rightUltrasonic.getDistance();
          if (millis() - t > 4000) {
            while (frontDistance < 9 || rightDistance < 9) {
              if (rearLeftIR.isOnTape()) {
                rotateCCW(speed);
              }
              else if (rearRightIR.isOnTape()) {
                rotateCW(speed);
              }
              else {
                moveBackward(speed);
              }
              delay(40);
              frontDistance = frontUltrasonic.getDistance();
              rightDistance = rightUltrasonic.getDistance();
            }
          }
        }
        lance.write(18);
      }
    }
    else {
      if (!opponentAhead) {
        if (dirFront) rotateCW(speed);
        else rotateCCW(speed);
      }
    }
}
else if (leftDistance < distanceThreshold) {
  moveRight(speed);
}
else if (rightDistance < distanceThreshold) {
  moveLeft(speed);
}
else if (opponentLocation > 0) {
  attack();
}
else {
  moveForward(speed);
}
```

```
delay(30);
if (frontRightIR.isOnTape() == FR_IR && rearRightIR.isOnTape() == RR_IR && rearLeftIR.isOnTape() == RL_IR && frontLeftIR.
isOnTape() == FL_IR){
  index = index + 1;
  if (index > 50){
    lance.write(90);
    index = 0;
    moveBackward(speed);
    delay(500);
    rotateCW(speed);
    delay(500);
    lance.write(18);
  }
}
else {
  index = 0;
}
```