Final Robotic Contest: Robotic Carriers in Warehouses

Group Members: Jacob Taylor Cassady, Chase Crutcher, Olalekan Olowo

Robot: Cowboy

December 6, 2019

1 Objective

The Final Robot Contest is centered around designing an autonomous mobile robot to collect colored wooden boxes that have dimensions 1.5”x1.5”x1.5”. The contest play arena can be seen in the appendix, section 6.1, **figure 1**. The robot has two minutes to read a QR code that describes one of three colors, and then collect as many blocks matching the color from the QR code as it can before the time runs out. The robot gets two points for every correctly colored block it collects in its home base and loses a point for every wrong colored block. The arena’s floor is a colored gradient from light to dark with inverse gradients on each side of the center line. Home bases are flat colors of either black or white with values larger or smaller in magnitude than what is found on the gradient.

2 Robot Design

To start designing the robot, two large wheels were placed on sides of the robot chassis attached to DC motors. A castor ball was placed on the front of the robot for balance and two LEGO pieces were screwed on the back to act as pseudo-wheelie bars. The camera was affixed on the front of the robot chassis above the castor ball to be as close to the front as possible. This was done to improve QR code reading. For reference to the hardware and sensor locations on the robot chassis, please refer to **figures 2** and **3** in the appendix, section 5.2. An image with a closer angle of the back IR sensor as well as the pseudo-wheelie bar car be found in **figure 4** of the appendix, section 6.2.

The robot’s hardware design went through many iterations with the idea of a front capture method for acquiring blocks. The first design used an open and close gate, but the additional step was found to be unnecessary. We settled on a front “lasso” the robot uses to surround the object and drag it along to the goal. This lasso is controlled by a servo motor to precisely alternate between two predefined encoder positions. For reference to examples of the lasso’s up and down states please see **figures 5** and **6** in the appendix, section 6.2.

Buttons were affixed to the front and back of the robot to act as touch sensors. Initial designs used LEGOs to attach the buttons, but this was found to be unreliable and brittle. Instead, electrical tape was used. This proved to be a more durable solution.

An IR sensor was attached to the back of the robot using LEGOs and screws. The goal was to have it sit 2-3cm off the ground. Electrical tape was wrapped around the tip of the IR sensor to act as a hood and reduce stray light not being reflected off the ground. This was found to reduce noise in sensor output.

3 Robot Algorithm

The robot's algorithm was designed around a set of threads as well as a set of state machines. After waiting for any button to be pressed to signal the start of the competition, the robot starts three threads and waits 2 minutes to destroy the threads. The functionality of the threads is dependent on the state machines which are formalized using enumerations and passed around using global variables since the built-in thread_create function does not allow for arguments. The Drive Thread and Robot State machine contain the primary logic for the competition.

3.1 State Machines

State machines are used to alter functionality in threads by simply updating a global variable. A more functional description of how these state machines interact with the robot's presentation and decision making can be found in section 3.2 of this document.

3.1.1 Lasso State Machine

There are two valid lasso states. Each corresponds to an encoder position of the servo motor. The robot's lasso state is kept in a global variable, gv_lassoState, that is initialized to LASSO_STATE_UP. A table of the lasso states is shown below.

Table 1 : Lasso State Enumerations

LASSO_STATE	Enum
LASSO_STATE_INVALID	0
LASSO_STATE_DOWN	1
LASSO_STATE_UP	2

3.1.2 Robot State Machine

There are five valid robot states. Each corresponds to a sub-goal the robot is trying to achieve. The initial idea was to further break this down into sub-sub-goals but too much time was spent on building the robot. The robot's state is held in a global variable, gv_robotState, that is initialized to ROBOT_STATE_READING_QR.

Table 2 : Robot State Enumerations

ROBOT_STATE	enum
ROBOT_STATE_INVALID	0
ROBOT_STATE_READING_QR	1
ROBOT_STATE_SEARCHING_FOR_OBJECT	2
ROBOT_STATE_APPROACHING_OBJECT	3
ROBOT_STATE_APPROACHING_GOAL	4
ROBOT_STATE_LEAVING_BASE	5

3.2 THREADS

Threads were used to ensure the robot is responsive to multiple stimuli without over complicating the control flow. The software went through several iterations that included a wide array of threads such as a camera thread and a IR sensor thread. This is because initial implementations attempted to use running average and Kalman filter techniques to try to reduce the noise in sensor responses. This was largely unsuccessful; instead, camera and IR sensor logic were integrated into the Drive Thread.

3.2.1 Lasso Thread

The lasso thread constantly checks the global variable gv_lassoState and updates the lasso's location accordingly. Values for constants LASSO_DOWN_ENCODER and LASSO_UP_ENCODER were found experimentally by using the wallaby's servo GUI in settings. For exact function implementation please see software source file lasso.c. A control flow diagram of the lasso thread can be found in the Appendix, section 6.3, **figure 8**.

3.2.2 Button Thread

The button thread constantly stores the value of the digital inputs assigned to the front two buttons and the back two buttons. If one or both front two buttons have a high value, gv_dealingWithFrontButton is set to 1 for one second. If one or both back two buttons have a high value, gv_dealingWithBackButton is set to 1 for one second. A control flow diagram of the button thread can be found in the Appendix, section 6.3, **figure 9**.

3.2.3 Drive Thread

The drive thread begins by taking a reading from its IR sensor. If the sensor value is greater than 2000, it is assumed to have a white home base. If not, the home base is assumed to be black. This information is stored in a global variable gv_homeBase which is of enumeration type HOME_BASE. This is defined in globals.h along with all other enums and external global variables. Since the rest of the drive thread uses a switch on the variable gv_robotState, subheadings will be used to describe the functionality for each case.

3.2.3.1 case ROBOT_STATE_READING_QR

In the first state, the robot is following a set of scripted commands to ensure it efficiently travels to the QR code. The robot finds the left corner by first taking a left and then driving straight until the variable gv_dealingWithFrontButton becomes 1. The robot then turns left and begins reading the QR code. It slowly pans from left to right, and then right to left until 6 seconds has passed. Then it backs up and moves left and looks again, if it doesn't find the QR code it backs up and looks right, continuing this cycle until it finds it.

Once a valid QR reading is found, the robot updates the global variable gv_objectColor to match the respective OBJECT_COLOR enum. These enums map to the channel numbers in the wallaby's settings. After updating the object color global variable, the robot backs up and turns left. It then updates the variable gv_robotState to be ROBOT_STATE_SEARCHING_FOR_OBJECT.

3.2.3.2 case ROBOT_STATE_SEARCHING_FOR_OBJECT
The get_object_count function is constantly being called in this state. If there is an object, the robot updates the gv_robotState variable to be ROBOT_STATE_APPROACHING_OBJECT. If there is no object in sight, the robot begins to drive in a small circle for 6 seconds while looking for objects. If 6 seconds pass without an object, the robot begins to drive forward until its front buttons are pressed. At the point of the buttons being pressed, the robot backs up 2s and spins 180.

3.2.3.3 case ROBOT_STATE_APPROACHING_OBJECT
To approach the object, the motor speeds are updated at 10 Hz using the object's position in the image to determine the motor speed. If the object's y location in the image is less than OBJECT_LASSO_Y_CAP, then the motor powers are increased by a yPowerComponent. If the object's x location is not within a tolerance of OBJECT_LASSO_X_TOLERANCE both above and below OBJECT_X_GOAL, then one of the motor powers is increased to center the object. A snippet of the relevant code for object tracking can be found in the appendix, section 6.3, **figure 10**.

The NormalizeSensorReading function's arguments are currentValue, minValue, and maxValue. It returns a number between 0 and 100. If the yPowerComponent is 0, the robot sets gv_lassoState to LASSO_STATE_DOWN and sets gv_robotState to ROBOT_STATE_APPROACHING_GOAL. If the robot's get_object_count method ever returns 0 while in this state gv_robotState is set back to ROBOT_STATE_SEARCHING_FOR_OBJECT.

3.2.3.4 case ROBOT_STATE_APPROACHING_GOAL
When approaching the goal, the robot has two main methods. First it will spin 360 degrees taking IR readings and storing them in an array. It then sorts the IR readings using bubble sort and then spins again looking for another IR reading equal to or higher than the second highest IR reading from its first spin. The second highest value is used to mitigate the situation where the robot has an erroneously high value while spinning.

Once facing the direction with the highest value IR, the robot is assumed to be perpendicular to its home base. If its variable gv_homeBase is equal to HOME_BASE_WHITE, it will perform a 180 CW spin to ensure the goal is behind the robot and to its left. It then backs up until one of its back buttons are triggered. When triggered, it turns left, sets gv_lassoState to LASSO_STATE_UP and continues to drive forward until its front buttons are triggered. When they are triggered, the robot is assumed to be in its home and gv_robotState is set to ROBOT_STATE_LEAVING_BASE.

3.2.3.5 case ROBOT_STATE_LEAVING_BASE

To leave the base, the robot drives backwards for 4 seconds, spins 180 degrees clockwise and takes a left turn. It then updates gv_robotState to be ROBOT_STATE_SEARCHING_FOR_OBJECT.

4 Discussion and Conclusion

Cowboy performed better than expected in the competition. It was awarded 3rd place after losing in the semi finals when one of Cowboy's arms was knocked off. During Cowboy's final match to place 3rd, it successfully gathered all three blocks matching the color on its QR code. Improvements had to be made to Cowboy's structural design throughout competition play to improve upon the structural integrity such as the rubber band near the servo motor. This can be seen in **figure 5** of the appendix, section 6.2.

Cowboy's algorithm performed well in some ways but did have a weak spot. The algorithm used for finding the direction of the home base was not always reliable. Sometimes there would be a few erroneously large IR readings during the initial spin due to contact with another robot or a wall. This could cause the robot to want to spin until it gets an even larger IR reading before trying to go home. There were times the robot got snagged by another robot and got confused on the correct home direction.

There are several other additional improvements that could be made. Currently, the robot does not differentiate between a stimulus from its front left or front right buttons. The same is true for a stimulus from its back left or back right buttons. Additional logic could be added to increase or decrease the amount of subsequent turn following a button event. This may improve the robot's likelihood of correcting itself when its approach to home is not optimal. Another way to do this might be to add another IR sensor and use the differential between the two sensors to better inform he robot on its orientation before making a turn up against a wall.

5 Suggestions

A lot of time was spent trying to build things with LEGOs that I later found out could be done with electrical tape instead. In the future, it might help students get off on the right track if examples of when electrical tape and hot glue can be used legally in the competition were shown at the start.

6 APPENDIX

6.1 OBJECTIVE

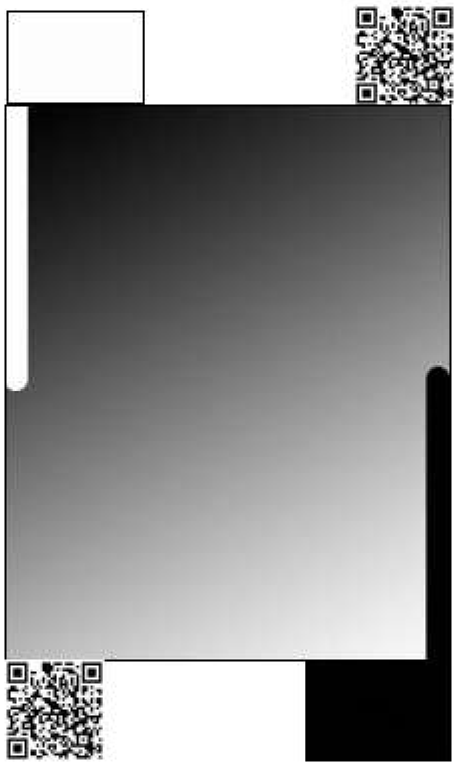


Figure 1 : Robot Competition Play Arena

6.2 Hardware

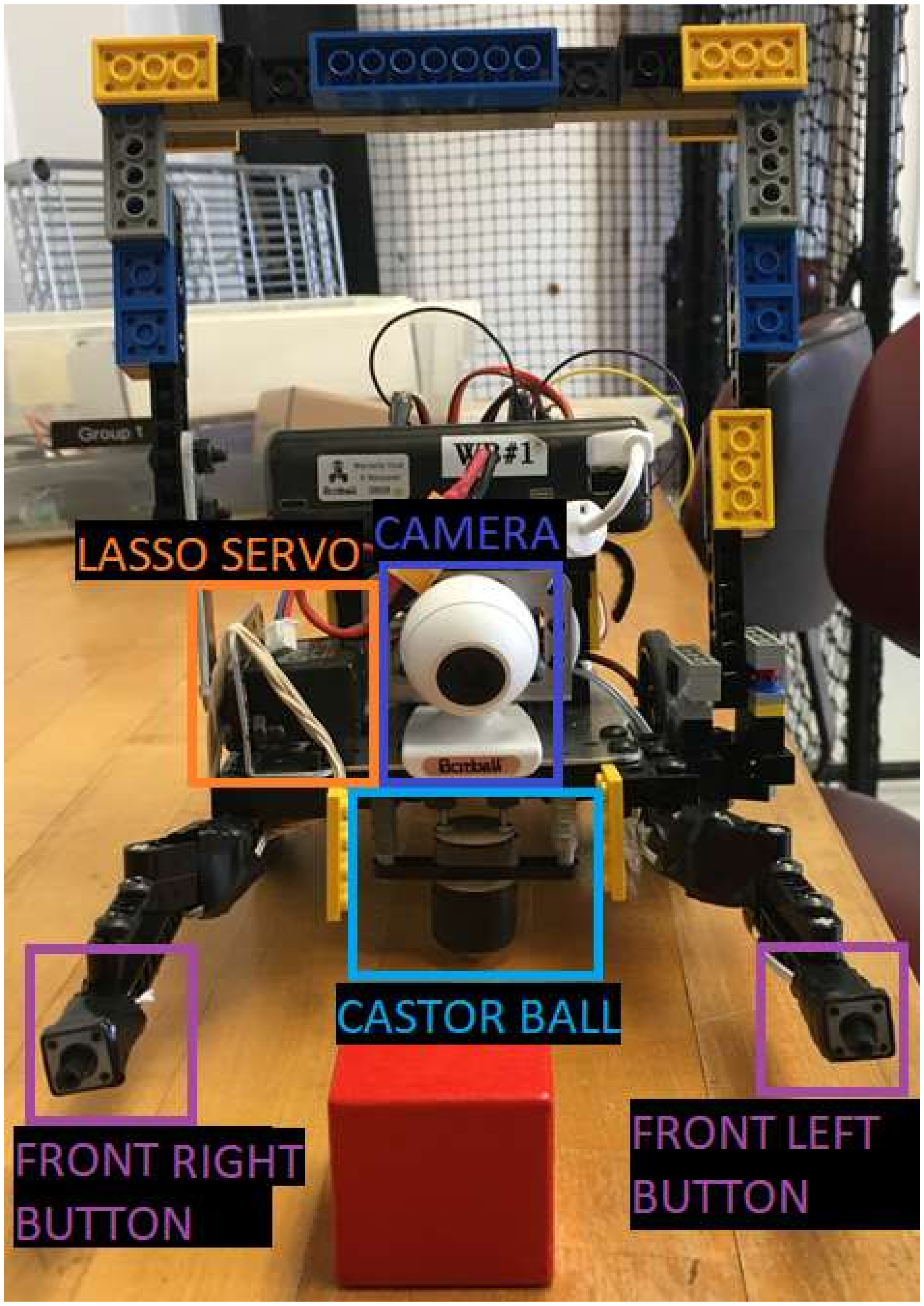


Figure 2 : Front Hardware and Sensors

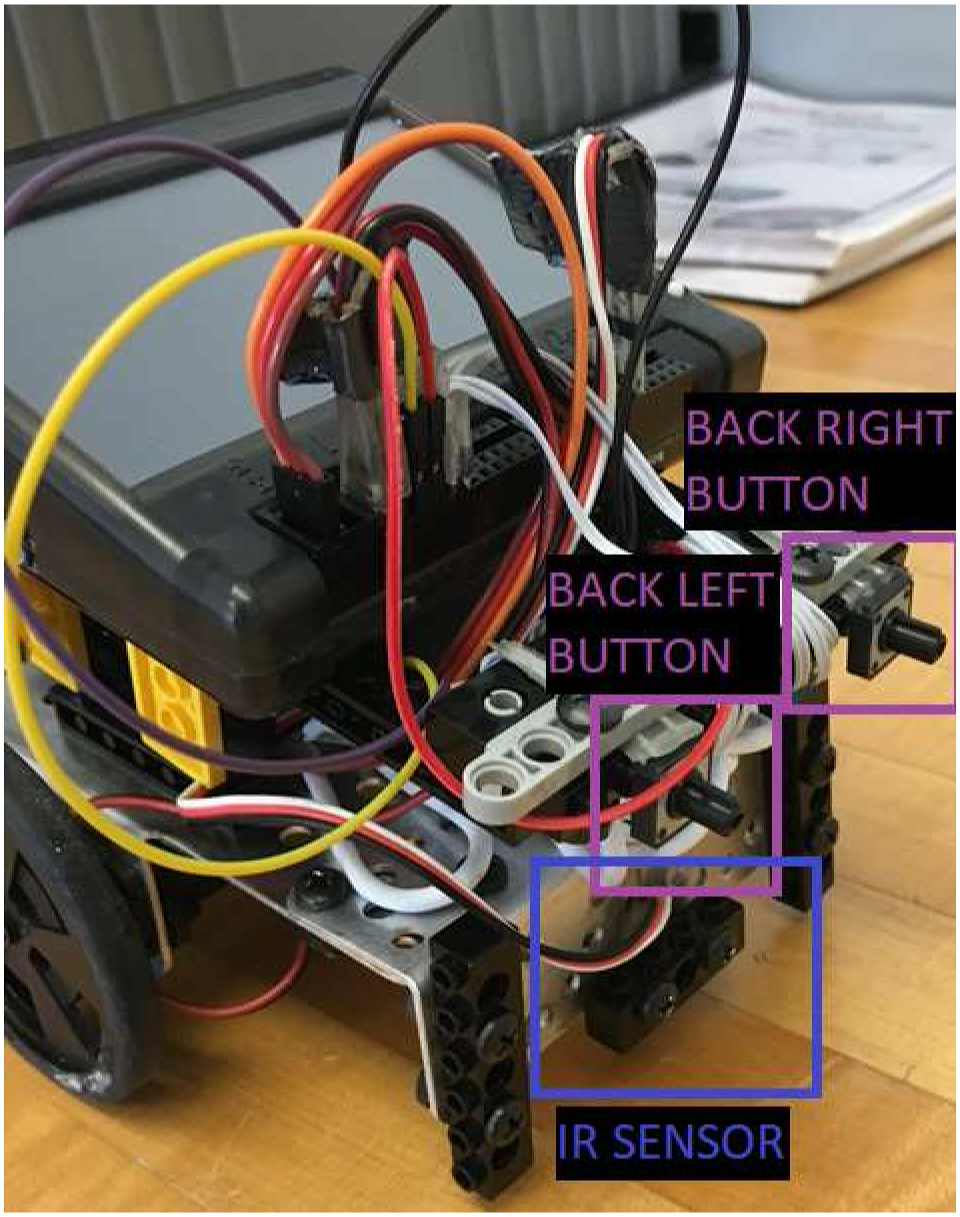


Figure 3 : Back Hardware and Sensors

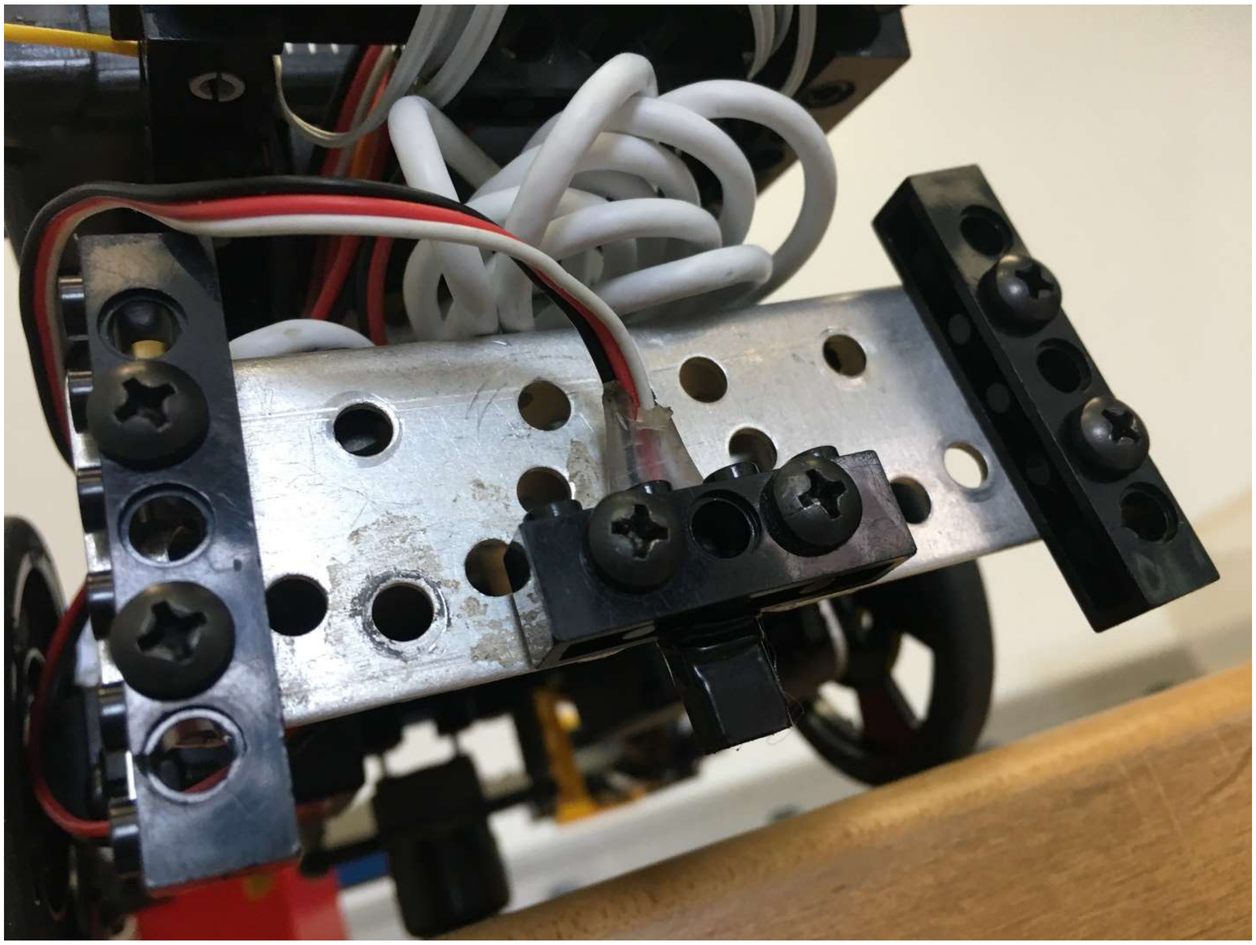


Figure 4 : IR Sensor Close Up

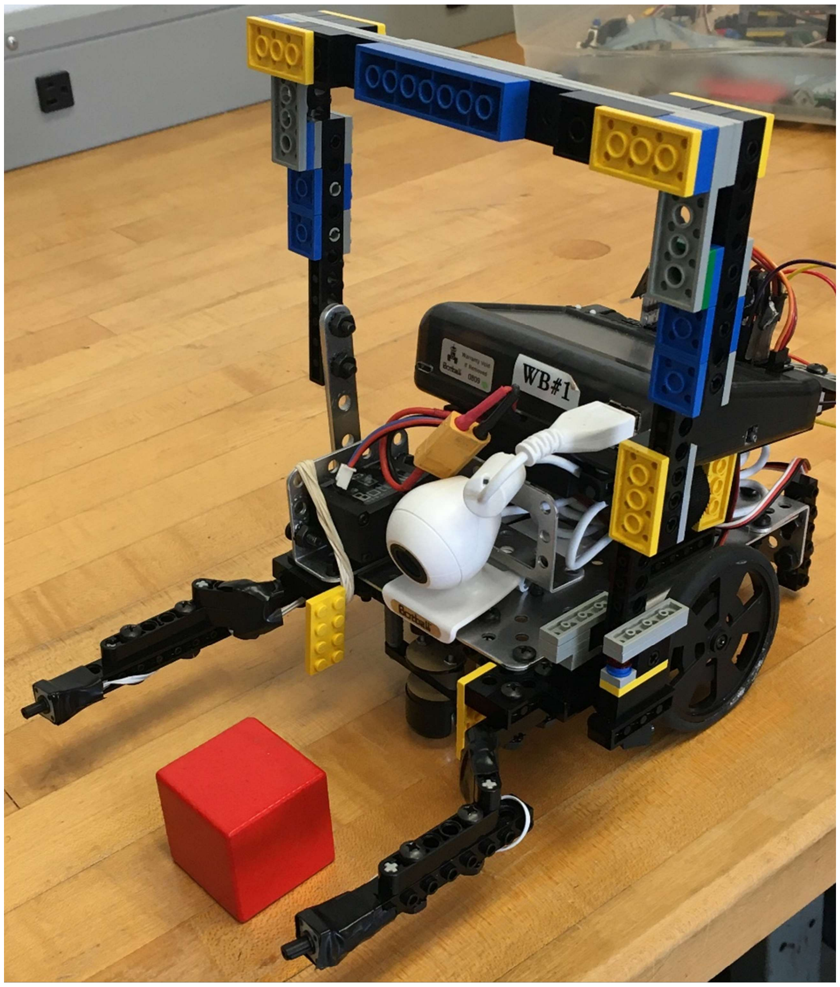


Figure 5 : LASSO_STATE_UP Encoder Position

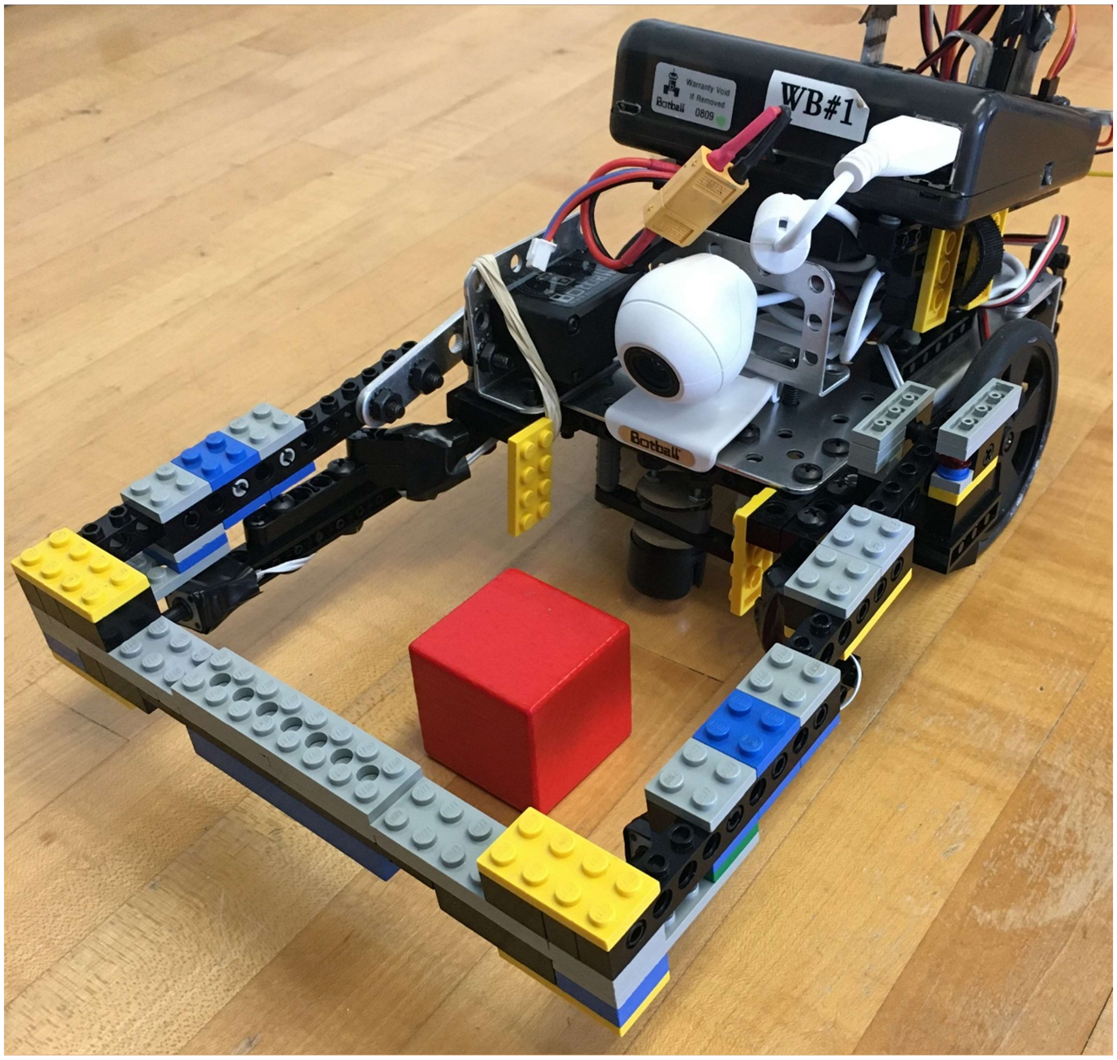


Figure 6 : LASSO_STATE_DOWN Encoder Position

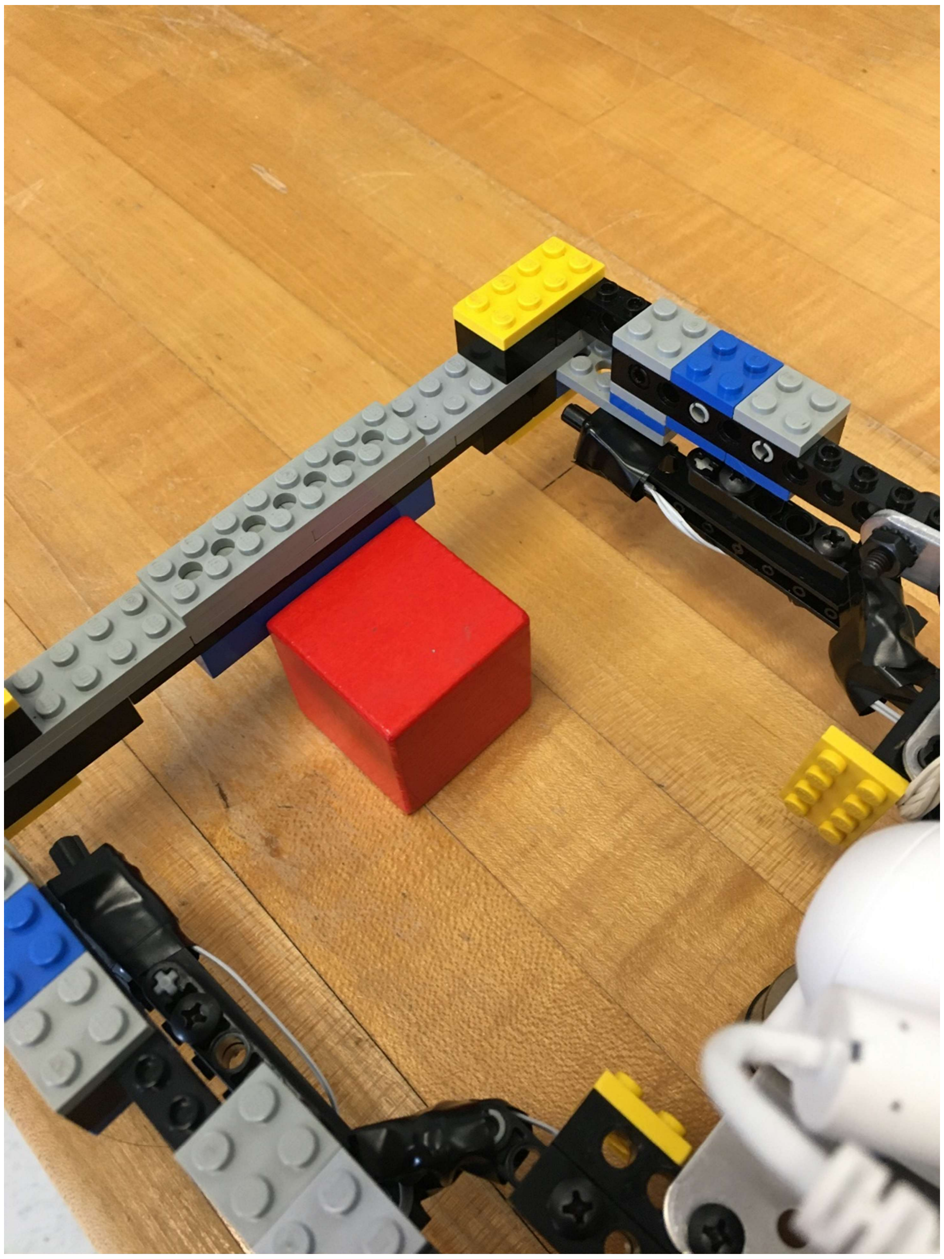


Figure 7 : LASSO_DOWN_STATE Clearing

6.3 SOFTWARE

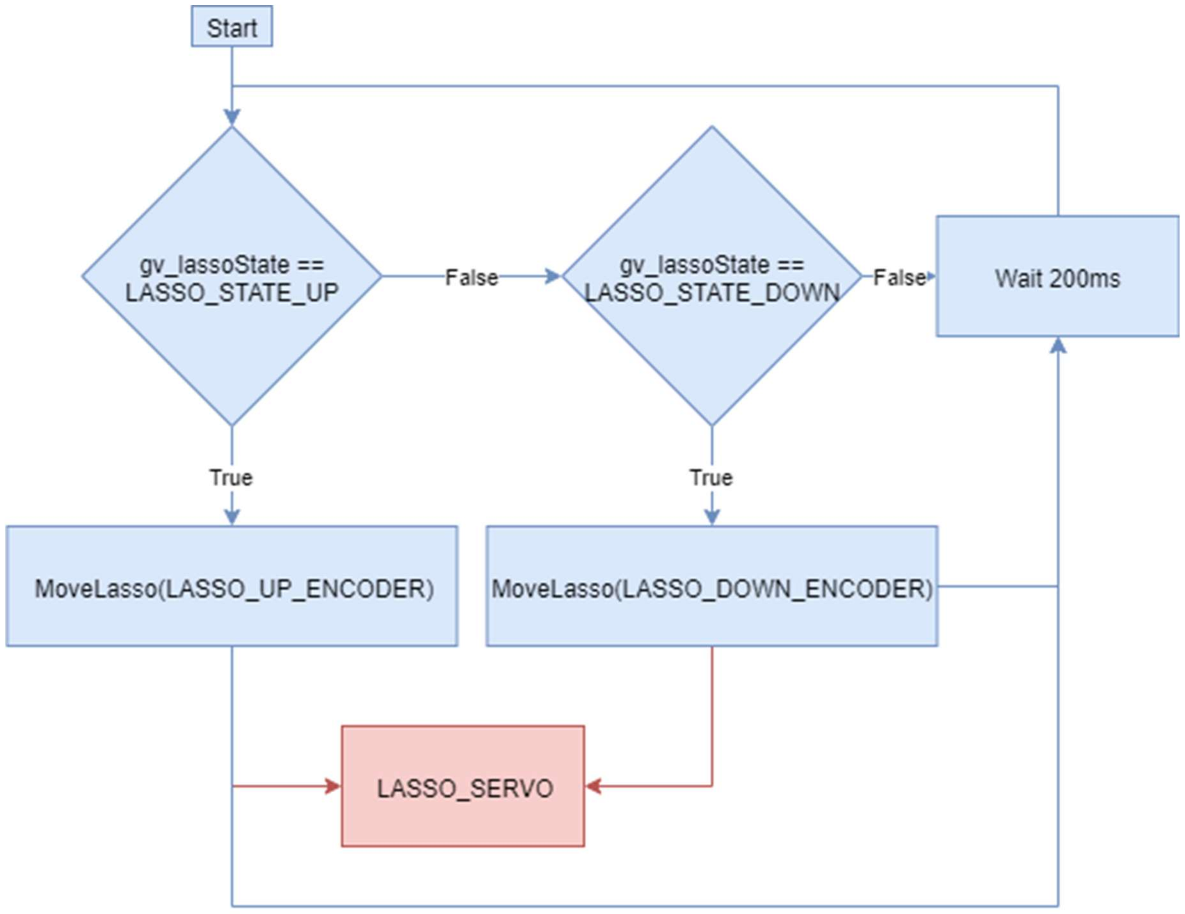

Start
gv_lassoState == LASSO_STATE_UP
False
gv_lassoState == LASSO_STATE_DOWN
False
Wait 200ms
True
True
MoveLasso(LASSO_UP_ENCODER)
MoveLasso(LASSO_DOWN_ENCODER)
LASSO_SERVO

Figure 8 : Lasso Thread Control Flow Diagram

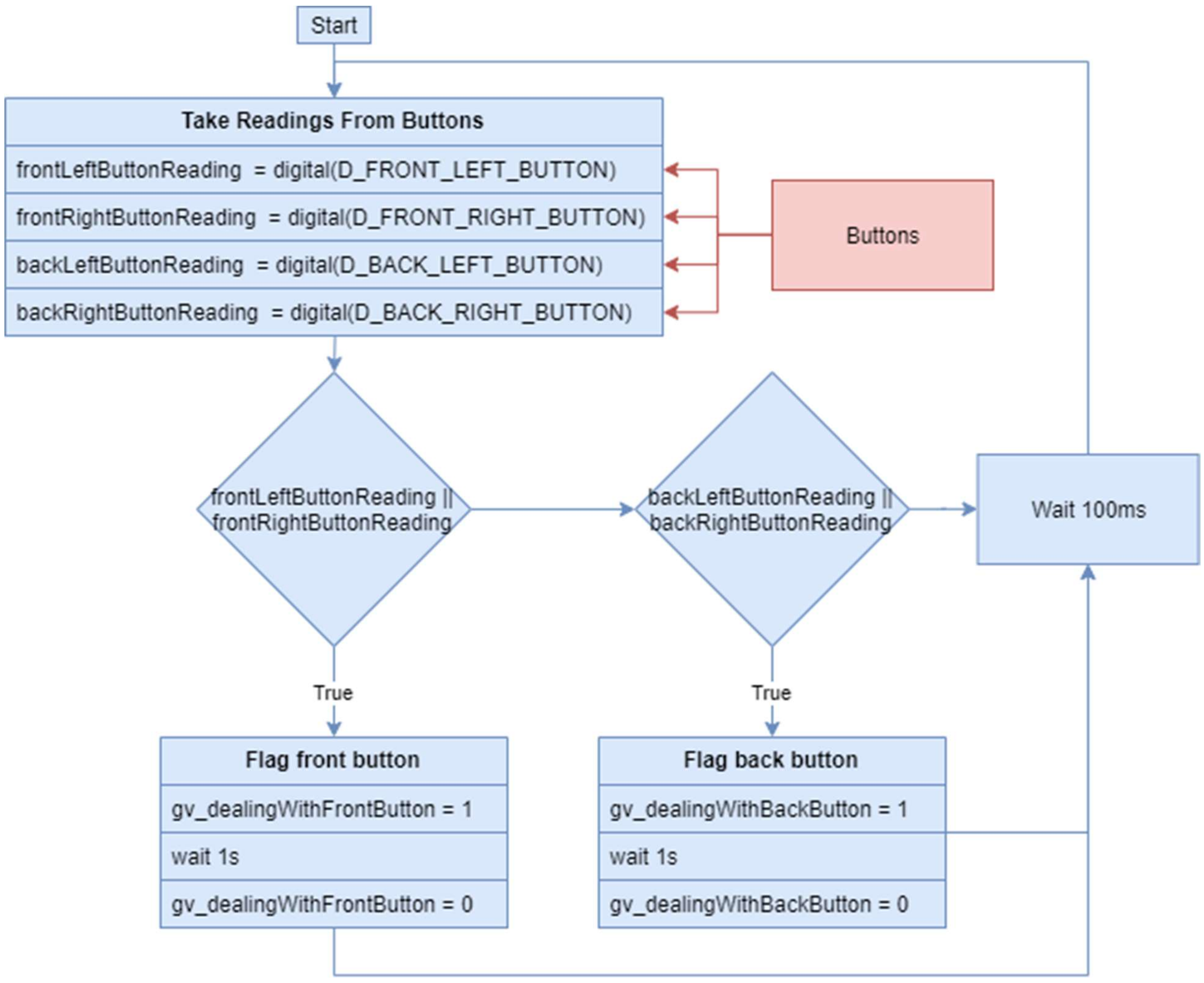
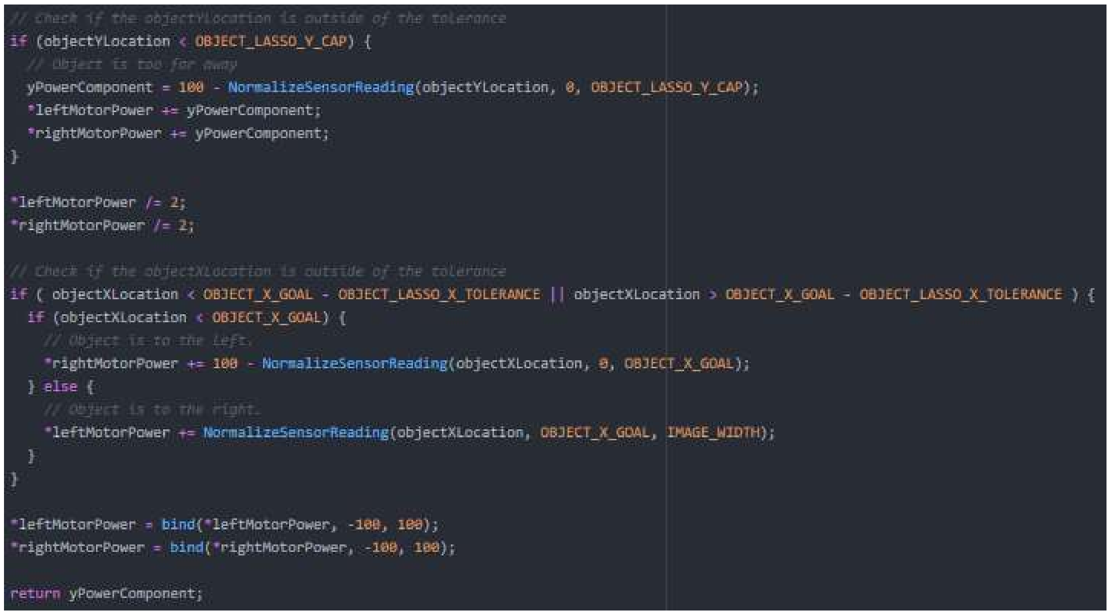

Start
Take Readings From Buttons
frontLeftButtonReading = digital(D_FRONT_LEFT_BUTTON)
frontRightButtonReading = digital(D_FRONT_RIGHT_BUTTON)
backLeftButtonReading = digital(D_BACK_LEFT_BUTTON)
backRightButtonReading = digital(D_BACK_RIGHT_BUTTON)
Buttons
frontLeftButtonReading || frontRightButtonReading
backLeftButtonReading || backRightButtonReading
Wait 100ms
True
True
Flag front button
gv_dealingWithFrontButton = 1
wait 1s
gv_dealingWithFrontButton = 0
Flag back button
gv_dealingWithBackButton = 1
wait 1s
gv_dealingWithBackButton = 0

Figure 9 : Button Thread Control Flow Diagram



```
// Check if the objectYLocation is outside of the tolerance
if (objectYLocation < OBJECT_LASSO_Y_CAP) {
  // Object is too far away
  yPowerComponent = 100 - NormalizeSensorReading(objectYLocation, 0, OBJECT_LASSO_Y_CAP);
  *leftMotorPower += yPowerComponent;
  *rightMotorPower += yPowerComponent;
}

*leftMotorPower /= 2;
*rightMotorPower /= 2;

// Check if the objectXLocation is outside of the tolerance
if ( objectXLocation < OBJECT_X_GOAL - OBJECT_LASSO_X_TOLERANCE || objectXLocation > OBJECT_X_GOAL - OBJECT_LASSO_X_TOLERANCE ) {
  if (objectXLocation < OBJECT_X_GOAL) {
    // Object is to the left.
    *rightMotorPower += 100 - NormalizeSensorReading(objectXLocation, 0, OBJECT_X_GOAL);
  } else {
    // Object is to the right.
    *leftMotorPower += NormalizeSensorReading(objectXLocation, OBJECT_X_GOAL, IMAGE_WIDTH);
  }
}

*leftMotorPower = bind(*leftMotorPower, -100, 100);
*rightMotorPower = bind(*rightMotorPower, -100, 100);

return yPowerComponent;
```

Figure 10 : Relevant Object Tracking Section