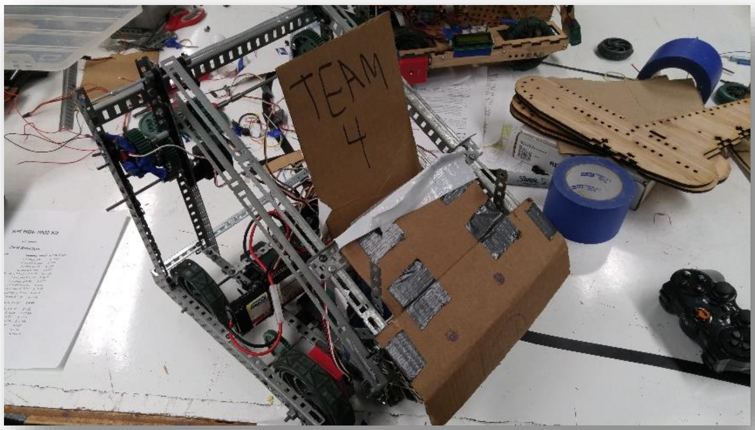
RBE 1001 Final Project Report

B'17, Team 4



Member	Signature	Contribution (%)
[redacted]		

Grading:	Presentation	_____/20
	Design Analysis	_____/30
	Programming	_____/30
	Accomplishment	_____/20
	Total	_____/10

Table of Contents

List of Figures

Introduction

We present a robot designed and built for demonstrating fundamental competencies in the principles of robotics engineered as outlined by the Robotics Engineering (RBE) 1001 class at Worcester Polytechnic Institute (WPI). The robot is intended to execute a strategy for scoring points in a modified version of the 2017 Savage Soccer game. The game has two primary periods, autonomous (auto) and teleoperated (tele-op). During autonomous, the robot must run a preprogrammed set of instructions and have no wireless communication. During tele-op, teams control their robot with a radio controller. In order to demonstrate the aforementioned competencies in auto and tele-op, our robot is equipped with:

I. Sensor based autonomous behavior.
II. A non-trivial power transmission.
III. A lifting device.
IV. A custom electronic circuit that adds functionality to the robot.

This set of equipment allows us to fully show our robotics engineering capabilities and the skills taught in RBE 1001.

Preliminary Discussion

Various strategies were considered during the preliminary stages of design. First, a defensive or passive design was immediately excluded from consideration because such a design does not align with demonstrating the fundamental competencies needed for the project. We then sought to consider the best possible offensive strategies.

For auto, it was determined that scoring ORBs in all three ORBITS would be the optimal strategy for scoring points due to a scoring bonus for scoring in more than one ORBIT during auto. Another important part of strategy in autonomous to consider is how the robot readies itself for the ORB drop at the end of autonomous. Placing the robot directly under the RAMP that drops the ORBs into play gives the team a direct benefit by both having ORBs fall into the robot and, in a competition scenario, prevents other teams from getting the ORBs. To properly execute on both strategies, line following and some method for sensing the ORBITs - i.e. bump or limit switch - might be used. Moreover, a relatively fast drivetrain is needed to accomplish these tasks in under 20 seconds during a competition scenario, but is not necessary in a demonstration scenario where more time may be allowed.

For tele-op, there are two small ball shaped objects to collect and score, the ORBs and ASTEROIDS, and one large ball, the STAR. Scoring into the higher ORBITs score more points with these balls, so the optimal strategy would be to score solely in the highest ORBIT, if possible. However, that means that the highest ORBIT would be prioritized and may fill up completely by the end of the match. Therefore, it is also beneficial to have a strategy that can adapt and also score in the other two ORBITs. Another important point to note is that two separate mechanisms will be needed to collect both the small balls - ORBs and ASTEROIDs - and the larger STAR. Having two such mechanisms may be a burden in the design process and prove to be not worth the opportunity cost of forgoing a second mechanism. Further, because the STAR is a one-time scoring opportunity, and likely harder to score than the smaller balls, scoring the smaller balls is a more useful strategy in general. Finally, teams may score points in tele-op by having their robot be fully supported by the BLACK HOLE. Again, the opportunity cost of driving onto the BLACK HOLE may be too high to justify, depending on final design considerations, as more points could potentially be scored during that time through other means. Also note that the BLACK HOLE isn't large enough to support many robots, which may be an important consideration in competition scenarios.

Problem Statement

With these strategies in mind, we decided that our primary designs goals are scoring in multiple ORBITs with small balls and accurately navigating the field. Secondary design goals include a fast drivetrain and ability to climb the BLACK HOLE. Tertiary goals include scoring the STAR and being able to push other robots. Scoring in multiple ORBITs achieves the autonomous multiplier and accurately navigating the field is necessary to do so, so both were put as highest priority goals. A fast drivetrain allows the robot to traverse the field quickly to score more points. In addition, a fast drivetrain allows us to reach multiple ORBITS in autonomous under a time limit. Climbing the BLACK HOLE provides us with a large scoring objective that we determined would be worth the opportunity cost because the highest ORBIT will likely be full close to the end of the match, as evidenced by the high school 2017 Savage Soccer game. Finally, scoring the STAR would provide us with additional points, but not as many as the small balls, so it was deemed of tertiary importance. Pushing other robots is also not necessary for demonstration purposes so it was deemed of tertiary importance, although it would be useful for competition scenarios.

Preliminary Designs

In order to achieve those design goals, we first brainstormed a U-shaped drivetrain with a V-shaped guide rail in the front to easily line up the ORBITs with our robot. From there, we decided that 4'' wheels and 4x 393 motors with a speed ratio of 1 would allow us to move quickly and climb the BLACK HOLE (see Final Design Analysis for derivation). A limit switch was included in our design on the tip of the V-shaped rail to detect when our robot contacts with the ORBIT, useful for autonomous operations.

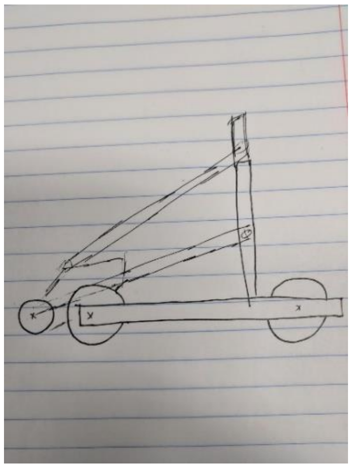


Figure 1. A side view of our four-bar linkage mechanism.

Figure 1 above demonstrates a first idea of a four-bar linkage that allows for lifting capabilities. Rubber band intake rollers would be on the front of the linkage, allowing for collection of the small balls. Behind the intake subsystem would be a storage mechanism. Reversing the intake direction would allow for outputting the small balls into the ORBITs. This four-bar linkage would also need to reach a height of 18'' to reach the highest ORBIT.

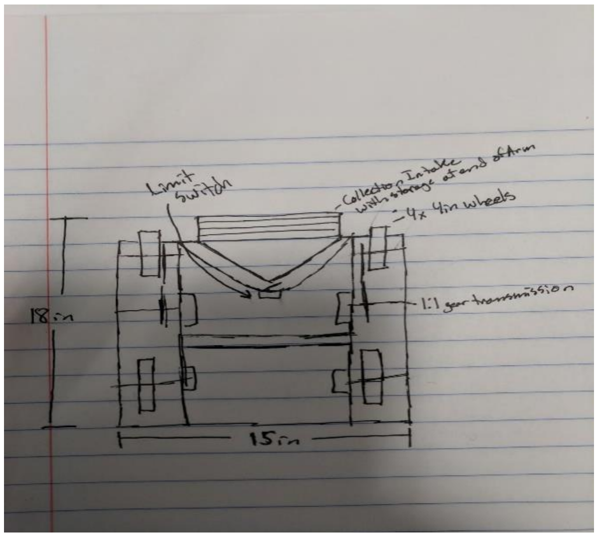
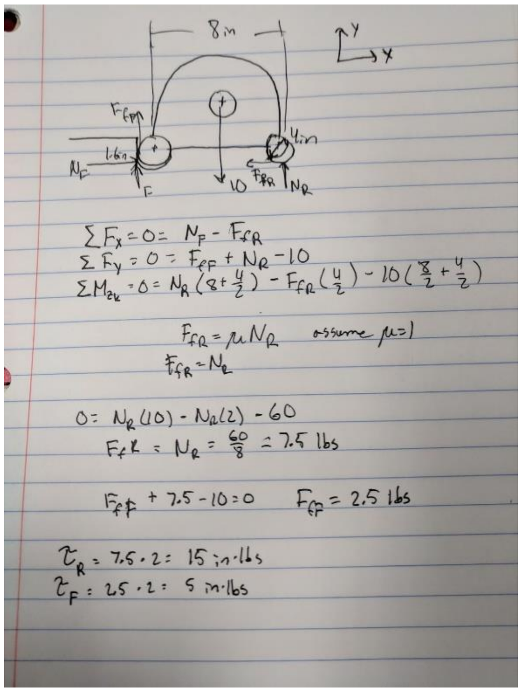


Figure 2. A top down view of our preliminary chassis.

Our preliminary chassis consists of a 15'' wide by 18'' long base. This design allows us to turn more directly by minimizing our wheel base with respect to our wheel track. Moreover, four-inch wheels were chosen to allow for easier climbing of the BLACK HOLE, alongside motor encoders for more accurate navigation of the field. The general U-shape design and V-shaped guide rail are also shown in Figure 2.



$$\sum F_x = 0 = N_F - F_{fR}$$
$$\sum F_y = 0 = F_{fF} + N_R - 10$$
$$\sum M_{z_k} = 0 = N_R\left(8 + \frac{4}{2}\right) - F_{fR}\left(\frac{4}{2}\right) - 10\left(\frac{8}{2} + \frac{4}{2}\right)$$

$F_{fR} = \mu N_R$ assume $\mu = 1$

$F_{fR} = N_R$

$$0 = N_R(10) - N_R(2) - 60$$
$$F_{fR} = N_R = \frac{60}{8} = 7.5 \text{ lbs}$$

$F_{fF} + 7.5 - 10 = 0 \qquad F_{fF} = 2.5 \text{ lbs}$

$$\tau_R = 7.5 \cdot 2 = 15 \text{ in}\cdot\text{lbs}$$
$$\tau_F = 2.5 \cdot 2 = 5 \text{ in}\cdot\text{lbs}$$

Figure 3. Initial calculations for necessary torque to climb the BLACK HOLE.

Initial calculations were also made to ensure that 4x 393 motors with 4'' wheels can climb the BLACK HOLE, as shown in Figure 3. We assumed that our center of gravity would be laterally and vertically centered.

Selection of Final Design

For our final design, we decided to first go with a four-bar mechanism for lifting. We choose this mechanism because it was the easiest to build and we already had experience in analyzing this type of mechanism. We had also considered an elevator style lift, but we deemed it harder to build and functionally the same as a four-bar for our purposes. Our non-trivial transmission requirement would be met by our lifting mechanism. For our custom electrical circuit, we considering using our old line tracker circuit or using an LCD screen to display sensor and debugging information. We went with the LCD screen because we requested and received the lower profile Vex line tracker kit. For our driveline, we decided to use 4'' wheels because that allowed us to climb the BLACK HOLE easier. Furthermore, we went with four motors on our drivetrain to provide high speed and torque (See Final Design Analysis). We also ended up with the same shape and dimensions for our final drivetrain shape as in our preliminary design because initial tests proved that the design worked well for meeting our design goals.

Final Design Analysis

Mechanical Design

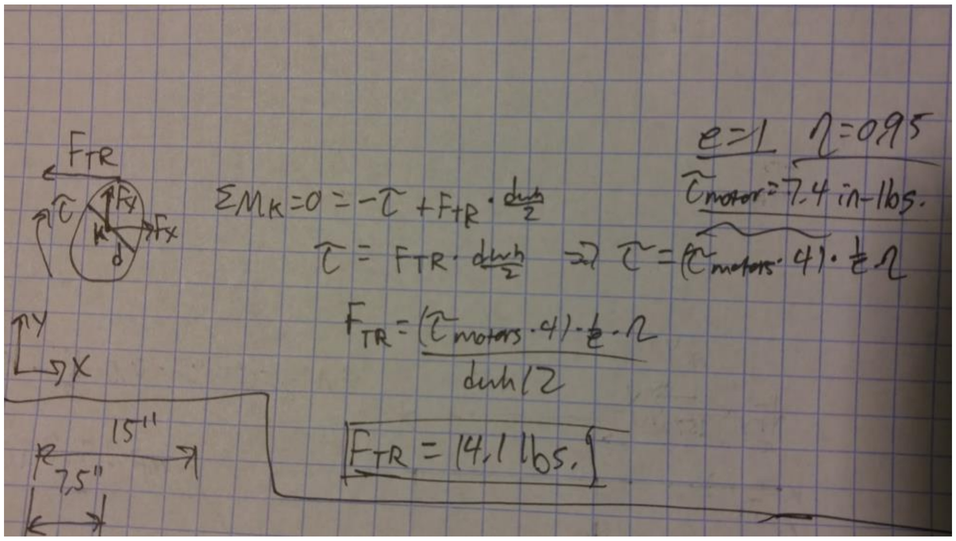
Motors

With our 1:1 gear ratio and 4'' wheels being powered by 4x 393 motors, our robot has a theoretical top speed of 21. in/sec. See Driveline section for derivation. Our desired torque is about 19 in-lbs for our driveline, which is satisfied by 4x 393 motors on our driveline. See Chassis section for derivation of driveline motor torque. Our lift requires a torque of 25.5 in-lbs., which is supplied by two 3-wire motors. See Lifter section for derivation.

Driveline

The speed of driveline is calculated by converting the angular velocity of the motors to linear velocity. This transform is given by the equation $v = d_{wheel}\pi e\omega$. We assumed our angular velocity to be 100 rpm because we want to examine our top speed condition. Our wheels are 4'' in order to climb the BLACK HOLE easier. We wanted our drivetrain to go about 20 in/s to traverse significant portions of the field in autonomous. Setting v equal to 20 and solving for e gave us a speed ratio of about 1:1. Having 4x 393 motors at this ratio also allowed us to meet the torque requirement of the chassis for climbing the BLACK HOLE, so we went with this ratio. See chassis section for derivation.

With that in mind, we also calculated the tractive force given the 1:1 gear ratio and the output torque of 4x 393 motors in Figure 4 below. Operating at 50% current for safety and efficiency reasons gave us an input torque of 7.4 in-lbs. per motor.


F_{TR}
τ
F_y
F_x
K
Y
X
15"
7.5"
$e=1$ $\eta=0.95$
$\tau_{motor}=7.4$ in-lbs.
$\Sigma M_K = 0 = -\tau + F_{TR}\cdot\frac{d_{wh}}{2}$
$\tau = F_{TR}\cdot\frac{d_{wh}}{2} \Rightarrow \tau = (\tau_{motors}\cdot 4)\cdot\frac{1}{e}\eta$
$F_{TR} = \frac{(\tau_{motors}\cdot 4)\cdot\frac{1}{e}\cdot\eta}{d_{wh}/2}$
$\boxed{F_{TR} = 14.1 \text{ lbs.}}$

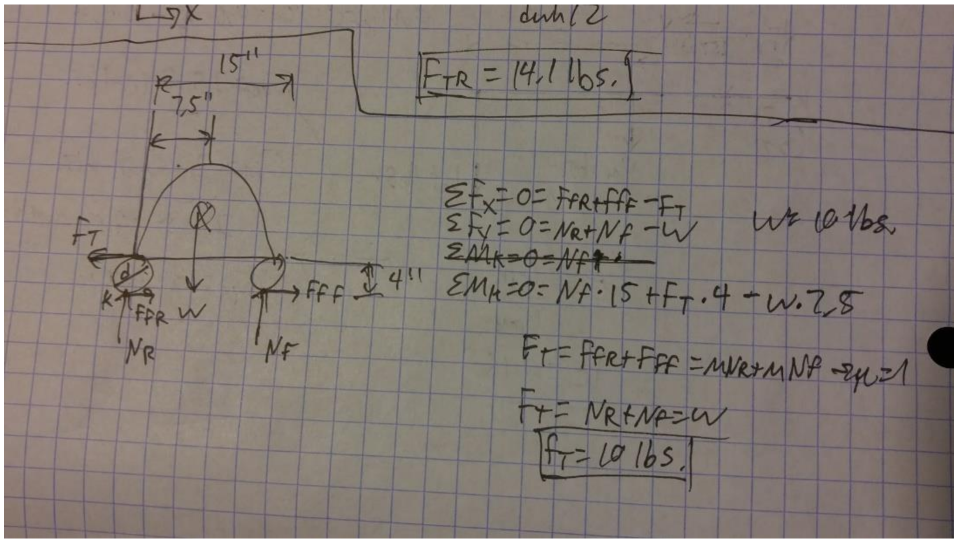


Figure 4. These diagrams represent the tractive force calculations for the motor power and stability cases respectively.

Lifter

For our linkage/lifter, our first consideration is the power requirement. To find our power requirement, we decided that 3 seconds would be a reasonable amount of time for our arm to raise its height to 18 inches from the ground. Consequently, we can use $\wp$= wh/s, where our weight is about 2. lbs. and our height is 18 inches. Our weight comes from the following: 1 lbs. structure + 0.2 lbs for intake motor + 0.6 lbs. for our links + 0.2 lbs. for game objects. Therefore, our power requirement is 12 in-lbs./sec, which converts to 1.36 watts of necessary mechanical power. With this in mind, we decided to use two 3-wire vex motors, which each have a max power of about 1.4 watts to ensure that we can pick up more game pieces worth than 0.2 lbs and so we can add additional parts to our linkages and collector if needed. We also decided to use 15'' links because they are long enough to reach 18'' in height and are readily available.

In terms of speed, we already determined that we wanted our lift to rise to 18 inches in 3 seconds, meaning that the arm goes through roughly 90 degrees, giving us an angular velocity of 5 rpm. Knowing that $e = \omega_{\text{out}}/\omega_{\text{in}}$, we then determined that our e for speed control should be 0.05.

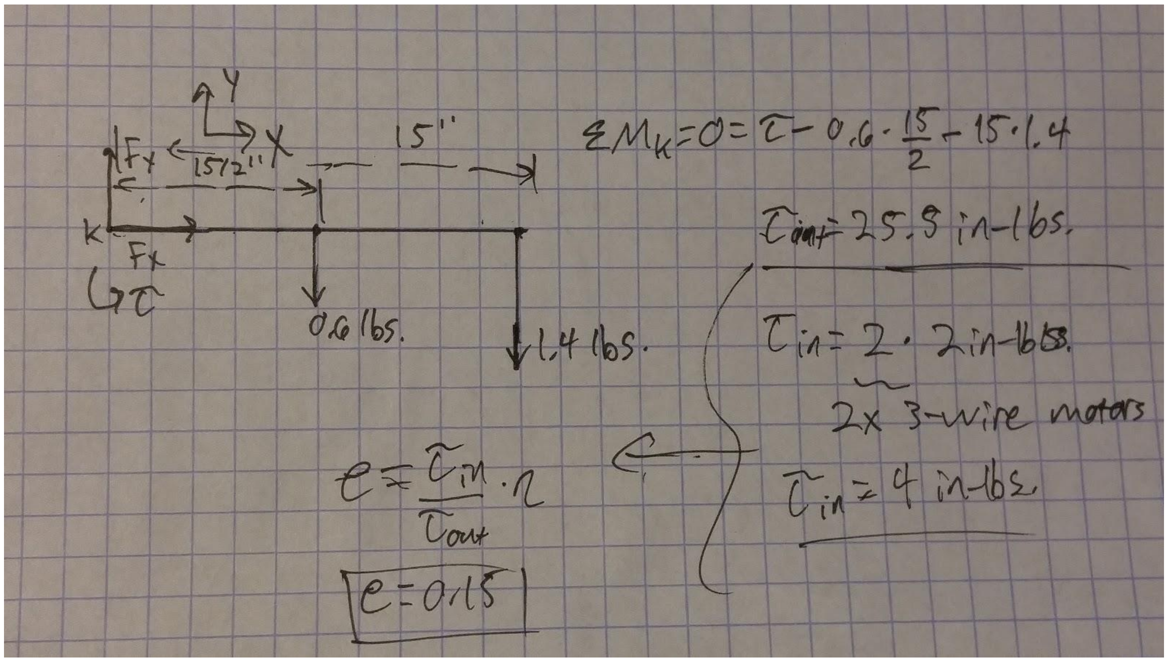


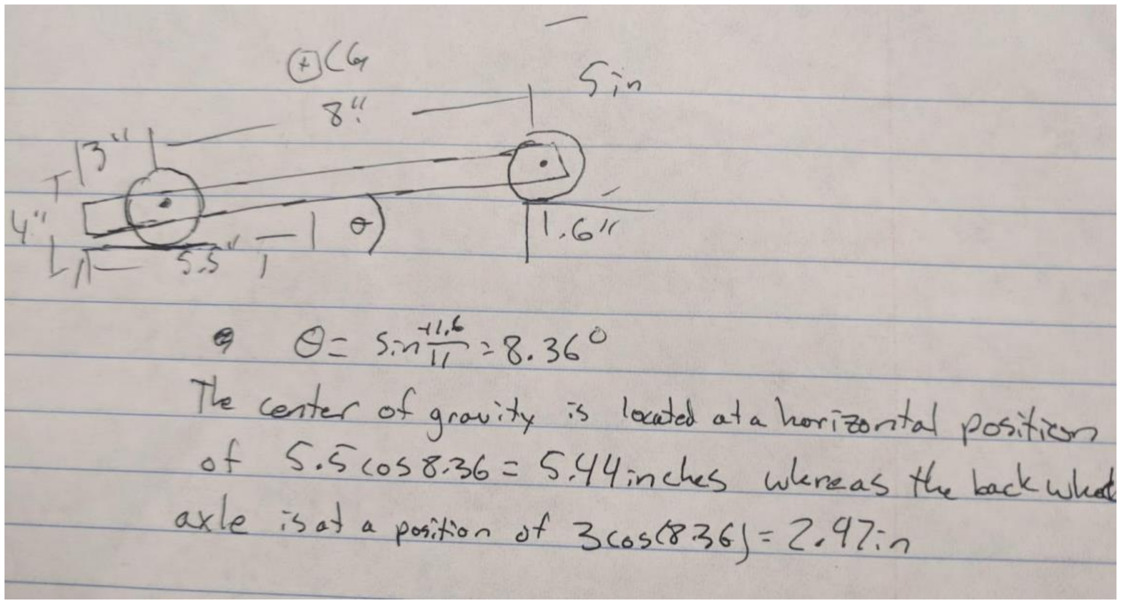
Figure 5. A diagram of calculations for torque requirements on our lift.

In terms of torque, we found that we needed an output torque of 25.5 in-lbs., as derived in Figure 5 above. Hence, with an input torque of 4 in-lbs., we needed a e of about 0.15, also shown in Figure 5.

Considering the gears we have available (12T, 36T, 60T), we decided to use a 1:9 two-stage transmission consisting of two 1:3 stages. This transmission gives us an e of about 0.11, satisfactorily close to our speed requirement and above our torque requirement, even when factoring in efficiency (torque e * $0.95^2 < 1/9$).

Chassis

One requirement for our robot was the ability to climb the BLACK HOLE without tipping over. The center of gravity of our robot was found to be at a location 5.5 inches away from the rear end and 2.5 inches away from the rear axle as well as 5 inches above the ground level. Using simple triangle geometry, we determined the angle that the robot makes when on the BLACK HOLE with respect to the floor. In addition, we found the location of the center of gravity in relation to the rear axle to make sure the robot would not tip backwards. When the front wheel of the robot is on the 1.6-inch-high black hole it creates an 8.36-degree angle with the ground. This causes the center of gravity to shift back to a horizontal distance of 2.47 inches from the rear axle as described in Figure 6. Since the center of gravity does not pass the rear axle the robot will not tip backwards when climbing the BLACK HOLE.

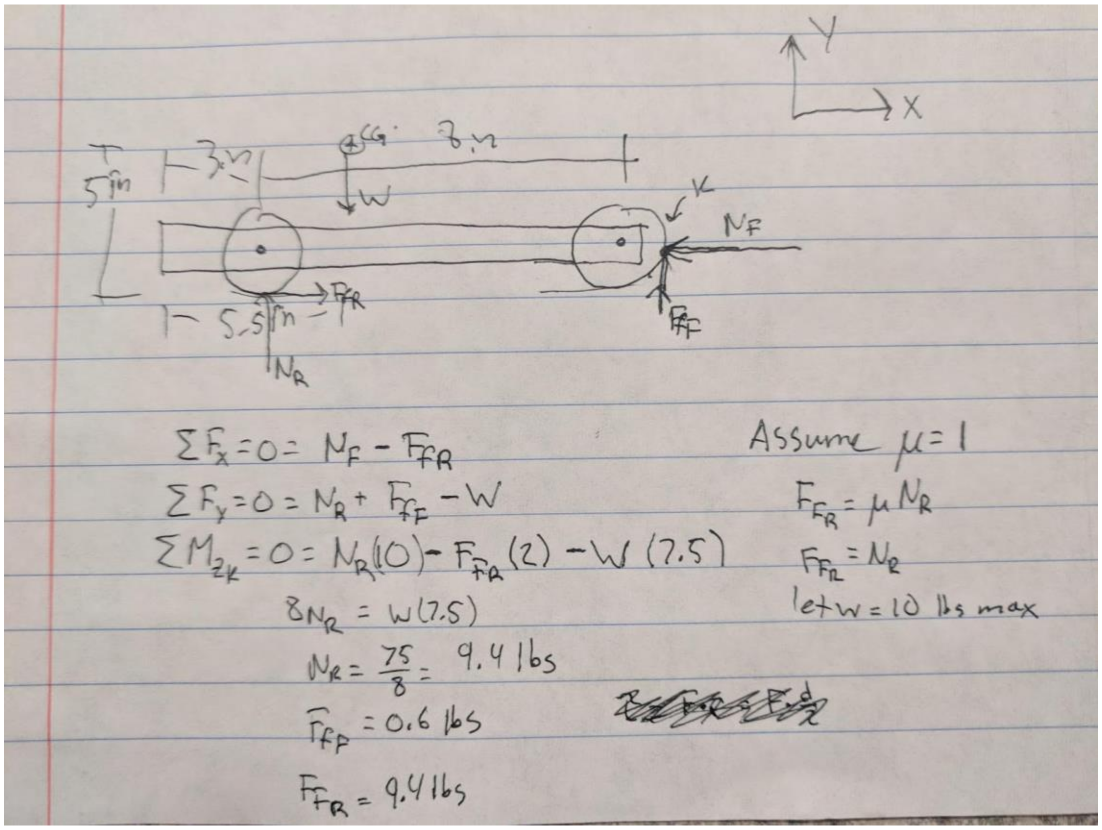


$\theta = \sin^{-1}\frac{1.6}{11} = 8.36^\circ$

The center of gravity is located at a horizontal position of $5.5\cos 8.36 = 5.44$ inches whereas the back wheel axle is at a position of $3\cos(8.36) = 2.97$ in

Figure 6. A diagram describing the stability of the robot when climbing the BLACK HOLE

In addition to making sure our robot does not tip over and is stable enough to climb the BLACK HOLE we must also provide enough torque to the wheels for the robot to climb the it. The FBD and calculations for this are pictured below.



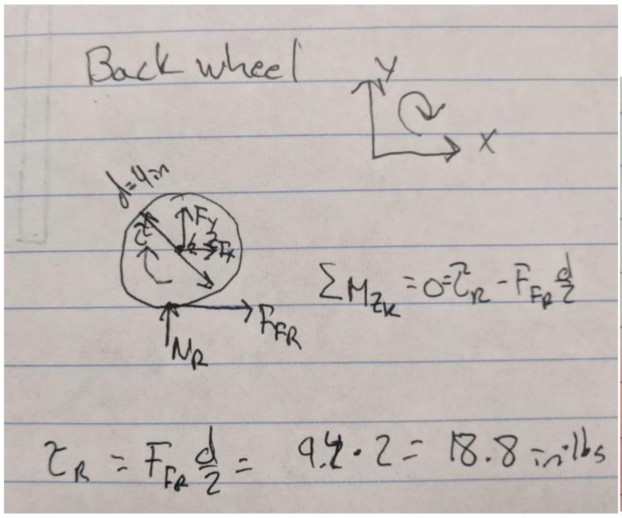
$$\Sigma F_x = 0 = N_F - F_{FR}$$
$$\Sigma F_y = 0 = N_R + F_{FF} - W$$
$$\Sigma M_{zK} = 0 = N_R(10) - F_{FR}(2) - W(7.5)$$
$$8N_R = W(7.5)$$
$$N_R = \frac{75}{8} = 9.4\text{ lbs}$$
$$F_{FF} = 0.6\text{ lbs}$$
$$F_{FR} = 9.4\text{ lbs}$$

Assume $\mu = 1$

$$F_{FR} = \mu N_R$$
$$F_{FR} = N_R$$

let $w = 10$ lbs max

Figure 7. FBD and EOE for the robot to climb the BLACK HOLE


Back wheel
Y
X
d=4in
F_y
F_x
τ
N_R
F_{FR}
$\Sigma M_{zk} = 0 = \tau_R - F_{FR}\frac{d}{2}$
$\tau_R = F_{FR}\frac{d}{2} = 9.4 \cdot 2 = 18.8$ in·lbs

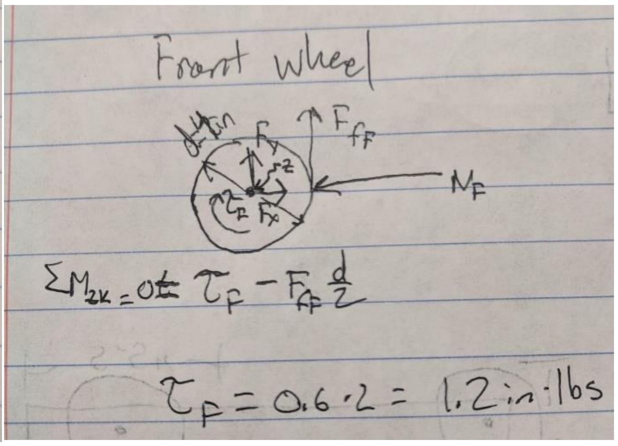

Front wheel
d=4in
F_y
F_x
F_{fF}
N_F
τ_F
$\Sigma M_{zk} = 0 = \tau_F - F_{fF}\frac{d}{2}$
$\tau_F = 0.6 \cdot 2 = 1.2$ in·lbs

Figure 8. The rear wheel FBD and EOE Figure 9. The front wheel FBD and EOE

Using the equations for equilibrium and the assumption that the coefficient of friction is one we found that while climbing the BLACK HOLE the front wheels would have to overcome a frictional force of 0.6 lbs and the back wheels would have to overcome a frictional force of 9.4 lbs, shown in Figure 7. An analysis was done on each the front and back wheel to find the necessary torque required to climb the black hole. Using the sum of the moments around the center axle of each wheel the necessary torque required to make the robot climb was determined. It was calculated that the front wheel would need to produce a torque of 1.2 in-lbs, Figure 8, and the back wheels would need to produce a torque of 18.8 in-lbs, Figure 9.

Electrical Design

As previously stated, we decided to use a LCD screen to display relevant sensor and debugging information. The pins 1 and 2 of the LCD display were connected to ground and a +5-volt power supply to power the screen. Pin 3 was connected to the 2nd pin on a potentiometer whose other two pins were also connected to ground and power. This allows for the brightness of the display screen to be altered. This is useful for achieving the correct brightness of the display screen so that the information is readable. The R/W pin was then connected to ground. When connected to a low power source or ground in this case the R/W pin is set to write mode which allows for characters to be written to the display. It was not necessary to connect this pin to Vcc or a high-power voltage supply at this would set it to read mode which reads the information being displayed on the LCD. The LED+ and LED- pin was then connected to power and ground allowing for the LEDs to turn on and illuminate the display. The RS pin connects to pin 40 on the Arduino which determines if the incoming data bit values are interpreted as commands or data to display. This will be important to have connected later so that the information we wish to have displayed can be displayed. The enable (E) pin connects to Arduino pin 41 which latches onto whichever data bits are being sent to the display. The enabling pin reads either high or low pulses. When this pin is set to low it ignores the information being sent through the R/W, RS, and data bus lines. When the E pin mode is set to high the LCD processes the incoming data. Arduino pins 42-45 are connected to data bus lines DB0-DB3 on the display which can be used later in our code to send information to be displayed on the LCD screen, and if needed, information can be sent back from the LCD screen to the Arduino.

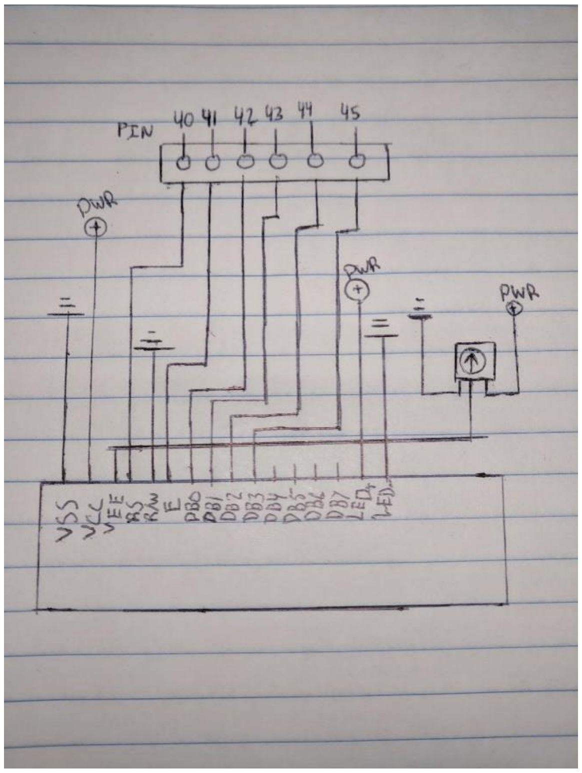


Figure 10. LCD screen circuit diagram.

Software Design

Software was designed with a top-down methodology. We first created a set of classes representing each of the physical subsystems on our robot, including the chassis, the lift, the collector, and the sensor array. Each class consists of necessary components, like corresponding motors and encoders.

The chassis class controls the four motors on the drivetrain in a master-slave schema (left and right sides) and uses encoders for error correction in autonomous driving. The lift class controls the two motors responsible for lifting our four-bar linkage and uses the potentiometer to control the angle to which the linkage is set. The collector class controls the intake motor. The sensor array class consists of methods for reading each of the sensors. The sensors we used include two line trackers and a limit switch.

A master control algorithm runs inside of the MyRobot.cpp file in the DFW template. This file instantiates objects corresponding to each of the aforementioned subsystem classes and calls on a header file for all the constant variables. Moreover, the file runs our autonomous and tele-op routines.

Our algorithm for autonomous uses a state machine to run through different preset tasks. A high-level view of the algorithm is that it first directs the robot to drive until the limit switch is hit against the highest ORBIT, while also lifting our linkage, and then using the collector to output the ORBs. Afterwards, the robot backs up, turns, and then uses the line trackers and encoders to navigate towards the lowest ORBIT and deposit the final

ORB there. Finally, the robot backs up again, turns, and moves to the center of the field using dead (deduced) reckoning to prepare for the ORB drop.

Our algorithm for tele-op consists of continually updating our non-sensor subsystems based on joystick values. We also use our potentiometer to control specific linkage preset heights to allow for easier outputting of small balls.

Summary

For our Critical Design Review (CDR), we feel that we did not perform as well as we were capable of at the time. Despite the robot being able to drive autonomously and lift the bucket up to the 18-inch goal, we scored poorly in autonomous due to the fact that we failed to release the starting payload of four ORBs. This failure was due to a design flaw in our intake/outtake system which prevented the ORBs from being released and even caused some to become trapped within the rubbers bands used in this system. We made note of this and added in a piece of cardboard within our storage system which allowed for the ORBs to be dispensed from the outtake system effortlessly. In the Optional Extended Demonstration (OED), our modification allowed us to complete the autonomous phase with great success, as we were then able to drive up to the goal, lift the arm, and successfully unload all four ORBs into the 18-inch goal. Overall, after a slight modification our robot was able to perform successfully in the autonomous section of the OED after learning of problems during the CDR.

During the CDR our tele-op performed somewhat better than our autonomous did, but not to the full extent to which it could have. Initially our intake/outtake system was designed to pick up and dispense the ASTEROIDs in addition to the ORBs. For the CDR, our robot was able to successfully collect an ASTEROID through the intake system during tele-op. However, we were not able to dispense the ASTEROID because the combination of the starting ORBs and an ASTEROID caused our intake motor to stall. This issue was primarily due to the fact that the ORBs would not properly dispense themselves, meaning that the ASTEROID could also not be dispensed. After learning of this problem, it was concluded that the previous modification stated above for autonomous would also allow us to unload ORBs easily and prevent jamming during tele-op. Additionally, we were not able to climb the BLACK HOLE in tele-op which was part of our original design consideration. This difference is largely because our center of gravity was too far back, which caused our front wheels to slip once they were on the BLACK HOLE. Furthermore, we modified our code for controlling the intake/outtake system. We observed that the speed of the outtake system was too much, and, on occasion, the ORBs would overshoot and miss the intended goal. This problem was solved by adding additional code to allow for slower intake/outtake speeds, thus preventing ORBs from missing the goal. We had hoped to gain more points back in OED after making the modification that would allow our robot to collect and dispense ORBs easily during tele-op. However, we could not demonstrate this capability because of a problem existing with our Arduino and/or Xbee controller which caused us to not have any communication between our robot and controller. After speaking with multiple people and spending several hours attempting to find a solution to this problem, no RBE 1001 personnel were able to fix this problem in time for the OED. Despite that, our robot was fully capable of a max accomplishment score in tele-op had there been no communication issue, as all the testing we had done prior to this issue indicated that we could score many ORBs and ASTEROIDs reliably and quickly after correcting the issues in CDR.

Appendix

Program Documentation

RBE_1001_Team_4_CodeV2.ino

```
/* This is the RBE 1001 Template as of
 *
 * 3/28/17
 *
 * This Template
 * is designed to run the autonomous and teleop sections of the final
 * competition. Write and test your autonomous and teleop code on your
 * own and place the code in auto.cpp or teleop.cpp respectively.
 * The functions will be called by the competition framework based on the
 * time and start button. DO NOT change this file, your code will be called
 * by the framework. The framework will pass your code a reference to the DFW
 * object as well as the amount of MS remaining.
 */
#include <DFW.h>
#include "MyRobot.h"

MyRobot robot;
DFW dfw(&robot); // Instantiates the DFW object and setting the debug pin.
The debug pin will be set high if no communication is seen after 2 seconds

void setup() {
        Serial.begin(9600); // Serial output begin. Only needed for debug
        dfw.begin(); // Serial 1 output begin for DFW library. Buad and port
#."Serial 1 only"
        robot.initialize();
        robot.dfw=&dfw;
}

void loop() {
        dfw.run();
  //robot.autonomous(0);

  robot.teleop(0);
}
```

Chassis.cpp

```
#include "Chassis.h"
#include "Arduino.h"

/**
 * Attaches the motors and encoder pins for the chassis.
 * @param leftPin the left motor pin
 * @param leftSlavePin the left slave motor pin
 * @param rightPin the right motor pin
 * @param rightSlavePin the right slave motor pin
 * @param rightEncPin1 the first right encoder pin
 * @param rightEncPin2 the second right encoder pin
 * @param leftEncPin1 the first left encoder pin
 * @param leftEncPin2 the second left encoder pin
 */
void Chassis::initialize(unsigned leftPin, unsigned leftSlavePin, unsigned
rightPin, unsigned rightSlavePin, unsigned rightEncPin1, unsigned
rightEncPin2, unsigned leftEncPin1, unsigned leftEncPin2) {
  leftMotor.attach(leftPin, 1000, 2000);
  leftSlave.attach(leftSlavePin,1000,2000);
  rightMotor.attach(rightPin, 1000, 2000);
  rightSlave.attach(rightSlavePin,1000,2000);
  encL1 = leftEncPin1;
  encL2 = leftEncPin2;
  encR1 = rightEncPin2;
  encR2 = rightEncPin2;
}

/**
 * Writes speeds to the motors.
 * @param leftSpeed the left motors speed
 * @param rightSpeed the right motors speed
 */
void Chassis::motors(int leftSpeed, int rightSpeed) {
  leftMotor.write(leftSpeed);
  leftSlave.write(180 - leftSpeed);
  rightMotor.write(180 - rightSpeed);
  rightSlave.write(rightSpeed);
}

/**
 * Writes speeds to the motors, but normed to -90 -> 90.
 * @param leftSpeed the left motors speed
 * @param rightSpeed the right motors speed
 */
```

```
void Chassis::motorsNorm(int leftSpeed, int rightSpeed) {
  leftMotor.write(90 + leftSpeed);
  leftSlave.write(90 - leftSpeed);
  rightMotor.write(90 - rightSpeed);
  rightSlave.write(90 + rightSpeed);
}

/**
 * Drives the motors at a normed speed and a direction.
 * @param speed the motor norm speed
 * @param direction the direction of the driving
 */
void Chassis::drive(int speed, int direction) {
  motorsNorm(speed + direction, speed - direction);
}

/**
 * Encoded version of driving. NOT USED****
 * @param vel speed of the driving
 * @return the average ticks of encoders ****not implemented
 */
int Chassis::encodedDrive(int vel) {
  int leftEncoderValue = digitalRead(encL1);
  int rightEncoderValue = digitalRead(encR1);

  int turn = leftEncoderValue - rightEncoderValue;

  drive(vel, turn);

  // fix to return average ticks of encoders
  return 1;
}

/**
 * Stop the motors.
 */
void Chassis::stop() {
  motors(90, 90);
}
```

Chassis.h

```
#pragma once

#include "Servo.h"

class Chassis {

private:
  Servo leftMotor;
  Servo leftSlave;
  Servo rightMotor;
  Servo rightSlave;
  unsigned encL1;
  unsigned encL2; // dir?
  unsigned encR1;
  unsigned encR2; // dir?

public:
  void initialize(unsigned leftPin, unsigned leftSlavePin, unsigned rightPin,
unsigned rightSlavePin, unsigned rightEncPin1, unsigned rightEncPin2,
unsigned leftEncPin1, unsigned leftEncPin2);
  void motors(int leftSpeed, int rightSpeed);
  void drive(int speed, int direction);
  int encodedDrive(int vel);
  void motorsNorm(int leftSpeed, int rightSpeed);
  void stop();
};
```

Collector.cpp

```
#include "Collector.h"
#include "Arduino.h"

/**
 * Attaches the collector motor.
 * @param motorPin the collector motor pin
 */
void Collector::initialize(unsigned motorPin) {
  motor.attach(motorPin, 1000, 2000);
}

/**
 * Outtakes balls.
 */
void Collector::outtake() {
  motor.write(180);
}

/**
 * Outtakes balls at a slower speed.
 */
void Collector::outtakeSlow() {
  motor.write (125);
}

/**
 * Outtakes balls at an even slower speed.
 */
void Collector::outtakeSuperSlow() {
  motor.write (110);
}

/**
 * Intakes balls.
 */
void Collector::intake() {
  motor.write(0);
}
```

```
/**
 * Shuts off the collector motor.
 */
void Collector::stop() {
  motor.write(90);
}
```

Collector.h

```
#pragma once

#include "Servo.h"

class  Collector {

private :
  Servo motor;

public:
  void initialize(unsigned motorPin);
  void intake();
  void outtake();
  void outtakeSlow();
  void outtakeSuperSlow();
  void stop();
};
```

Constants.h

```
#pragma once

/**
 * Constant pins
 */
const unsigned LEFT_MOTOR_PIN = 4;
const unsigned LEFT_SLAVE_PIN = 11;
const unsigned RIGHT_MOTOR_PIN = 5;
const unsigned RIGHT_SLAVE_PIN = 9;
const unsigned LEFT_ARM_PIN = 7;
const unsigned RIGHT_ARM_PIN = 10;
const unsigned COLLECTOR_PIN = 8;
const unsigned POT_PIN = A0;
const unsigned LIMIT_SWITCH_PIN = 28;
const unsigned LINE_RIGHT_PIN = A11;
const unsigned LINE_LEFT_PIN = A10;
const unsigned RIGHT_ENC_PIN1 = 26;
const unsigned RIGHT_ENC_PIN2 = 27;
const unsigned LEFT_ENC_PIN1 = 22;
const unsigned LEFT_ENC_PIN2 = 23;

/**
 * Constant helper values
 */
```

```
const int FULL_SPEED = 90;
const int FULL_REVERSE = -90;
const int FORWARD_MEDIUM = 130;
const int BACKWARD_MEDIUM = 50;
const int BACK_NORM_MEDIUM = -60;
const int MOTOR_SAG_SPEED = 8;
const double KP_A = -1;
const unsigned UP_POT = 700;

/**
 * LCD pins
 */
const int rs=40, en=41, d4=42, d5=43, d6=44, d7=45;

// Needs fixing UPDATE: NOT USED****
const unsigned OUTTAKE_TIME = 1000;
const unsigned BACKUP_TICKS = 1000;
```

Lift.cpp

```
#include "Lift.h"
#include "Arduino.h"

/**
 * Attaches the lifter motors and sets the pot pin and proportionality
constant.
 * @param leftPin the left lifter motor
 * @param rightPin the right lifter motor
 * @param K the proportionality constant
 * @param potPin the pot pin
 */
void Lift::initialize(unsigned leftPin, unsigned rightPin, double K, unsigned
potPin) {
  leftMotor.attach(leftPin, 1000, 2000);
  rightMotor.attach(rightPin, 1000, 2000);
  Kp_a = K;
  pot = potPin;
}

/**
 * Sets the lift motors at a certain speed.
 * @param speed the speed of the lifter motors
 */
void Lift::motors(int speed) {
  leftMotor.write(90 + speed);
  rightMotor.write(90 - speed);
}

/**
 * Uses proportional control to set the lifter motors.
 * @param setpoint the setpoint for pot control
 */
void Lift::set(int setpoint) {
  int error = analogRead(pot) - setpoint;
  error = (analogRead(pot) - setpoint);
}

/**
 * Returns the analog value of the pot.
 * @return analog reading of pot.
 */
int Lift::potter() {
  return analogRead(pot);
}

/**
 * Stops the lifter motors.
 */
void Lift::stop() {
  motors(0);
}
```

Lift.h

```
#pragma once

#include "Servo.h"

class  Lift {

private:
  Servo leftMotor;
  Servo rightMotor;
  double  Kp_a;
  unsigned  pot;
  unsigned  tol;

public:
  void initialize(unsigned leftPin, unsigned rightPin, const double K,
unsigned potPin);
  void motors(int speed);
  void set(int setpoint);
  int potter();
  void stop();
};
```

MyRobot.cpp

```
#include "MyRobot.h"
#include "Arduino.h"
#include "Constants.h"

#include <LiquidCrystal.h>

LiquidCrystal lcd(rs, en, d4, d5, d6, d7);


/**
 * These are the execution runtions
 */
void MyRobot::initialize() {
  // Initialize objects
        chassis.initialize(LEFT_MOTOR_PIN, LEFT_SLAVE_PIN, RIGHT_MOTOR_PIN,
RIGHT_SLAVE_PIN, RIGHT_ENC_PIN1, RIGHT_ENC_PIN2, LEFT_ENC_PIN1,
LEFT_ENC_PIN2);
  lift.initialize(LEFT_ARM_PIN, RIGHT_ARM_PIN, KP_A, POT_PIN);
  collector.initialize(COLLECTOR_PIN);
  sensors.initialize(LIMIT_SWITCH_PIN, LINE_RIGHT_PIN, LINE_LEFT_PIN);

  // Set the staging for auto
  stage = DRIVE_UNTIL_SWITCH;

  // Set the LCD settings
  lcd.begin(16, 2);
  lcd.setCursor(0,0);
  printed = false;

  // Settings for limit switch
  pinMode(LIMIT_SWITCH_PIN, INPUT);
  digitalWrite(LIMIT_SWITCH_PIN, HIGH);
}

/**
 * Called when the start button is pressed and the robot control begins
 */
 void MyRobot::robotStartup(){

 }

/**
 * Called by the controller between communication with the wireless
controller
 * during autonomous mode
 * @param time the amount of time remaining
 * @param dfw instance of the DFW controller
 */
 void MyRobot::autonomous(long time){
  // State machine that changes stage for each step of auto routine
  switch(stage) {
    case DRIVE_UNTIL_SWITCH:
      if (!printed) {
        lcd.print("Drive  + switch");
        printed = true;
      }

      chassis.motors(FORWARD_MEDIUM, FORWARD_MEDIUM);
      lift.set(UP_POT);
      lift.motors(FULL_SPEED);

      if (sensors.limitSwitch() == 0) {
        chassis.stop();
        lift.stop();
        startTime = time;
        printed = false;
        stage = OUTPUT_ORBS_HIGH;
      }
      break;
    case OUTPUT_ORBS_HIGH:
      if (!printed) {
        lcd.clear();
        lcd.print("Output orbs");
        printed = true;
      }

      collector.outtakeSlow();
      lift.stop();

      if ((time-startTime) > OUTTAKE_TIME) {
        printed = false;
        stage = DONE;
        startTime = time;
      }
      break;
    case BACKUP:
      if (!printed) {
        lcd.clear();
        lcd.write("Backup");
        printed = true;
      }

      ticks = chassis.encodedDrive(BACK_NORM_MEDIUM);

      if (ticks >= BACKUP_TICKS) {
        printed = false;
        stage = TURN;
      }
      break;
    case DONE:
      if (!printed) {
        lcd.clear();
        lcd.write("Done");
        printed = true;
      }
      collector.stop();
      chassis.stop();
      lift.stop();
      break;
    case TURN:
      break;
    case FORWARD:
      break;
    case LINE_FOLLOW:
      break;
    case OUTPUT_ORBS_LOW:
      break;
    default:
      lcd.clear();
      lcd.print("Error");
      break;
  }
 }
/**
 * Called by the controller between communication with the wireless
controller
 * during teleop mode
 * @param time the amount of time remaining
 * @param dfw instance of the DFW controller
 */
 void MyRobot::teleop( long time){
    // Update joystick for chassis

    if (dfw->up() || dfw->down()) {
      if (dfw->up()) chassis.motors(FORWARD_MEDIUM,FORWARD_MEDIUM);
      else chassis.motors(BACKWARD_MEDIUM,BACKWARD_MEDIUM);
    }
    else chassis.motors(dfw->joysticklv(), dfw->joystickrv());

    // If button pressed, lift or lower arm
    if (dfw->r1() || dfw->one()) {
      if (dfw->r1()) lift.motors(FULL_SPEED);
      else lift.motors(FULL_REVERSE);

      collector.stop();
    }
    else lift.motors(MOTOR_SAG_SPEED);

    // If button pressed, turn on or off intake
    if (dfw->l1() || dfw->l2() || dfw->right() || dfw->left()) {
      if (dfw->l1()) collector.intake();
      else if (dfw->l2()) collector.outtakeSlow();
      else if (dfw->right()) collector.outtake();
      else collector.outtakeSuperSlow();
    }
    else collector.stop();

 }
/**
 * Called at the end of control to reset the objects for the next start
 */
 void MyRobot::robotShutdown(void){
              Serial.println("Here is where I shut down my robot code");
 }
```

MyRobot.h

```
#pragma once

#include "Servo.h"
#include <DFW.h>
#include <AbstractDFWRobot.h>
#include "Chassis.h"
#include "Lift.h"
#include "Collector.h"
#include "SensorArray.h"
```

```
#include "Constants.h"

class MyRobot :public AbstractDFWRobot{
public:
        DFW * dfw;
        /**
         * Called when the start button is pressed and the robot control
begins
         */
         void robotStartup();
        /**
         * Called by the controller between communication with the wireless
controller
         * during autonomous mode
         * @param time the amount of time remaining
         * @param dfw instance of the DFW controller
         */
         void autonomous(long time);
        /**
         * Called by the controller between communication with the wireless
controller
         * during teleop mode
         * @param time the amount of time remaining
         * @param dfw instance of the DFW controller
         */
         void teleop(long time);
        /**
         * Called at the end of control to reset the objects for the next
start
         */
         void robotShutdown(void);
        /**
         * Return the number of the LED used for controller signaling
         */
         int getDebugLEDPin(void){return 13;};

        void initialize();
        ~MyRobot(){};
private:
  // Declare objects and variables
        Chassis chassis;
  Lift lift;
  Collector collector;
  SensorArray sensors;
  enum stages {LIFT_LIMIT, DRIVE_UNTIL_SWITCH, OUTPUT_ORBS_HIGH, BACKUP,
TURN, FORWARD, LINE_FOLLOW, OUTPUT_ORBS_LOW, DONE} stage;
  long startTime;
  bool printed;
  int ticks;
};
```

SensorArray.cpp

```
#include "Arduino.h"
#include "SensorArray.h"

/**
 * Set the pins for each sensor.
 * @param limitPin the pin for the limit switch
 * @param lineTrackRightPin the pin for the right line tracker
 * @param lineTrackLeftPin the pin for the left line tracker
 */
void SensorArray::initialize(unsigned limitPin, unsigned lineTrackRightPin,
unsigned lineTrackLeftPin) {
  limit = limitPin;
  lt1 = lineTrackRightPin;
  lt2 = lineTrackLeftPin;
}

/**
 * Reads the limit switch.
 * @return digital reading of limit switch
 */
int SensorArray::limitSwitch() {
  return digitalRead(limit);
}

/**
 * Reads the right line tracker.
 * @return analog reading of right line tracker
 */
double SensorArray::trackerRight() {
  return (double) analogRead(lt1);
}

/**
 * Reads the left line tracker.
 * @return analog reading of left line tracker
 */
double SensorArray::trackerLeft() {
  return (double) analogRead(lt2);
}
```

SensorArray.h

```
#pragma once

class  SensorArray {

private :
  unsigned limit;
  unsigned lt1;
  unsigned lt2;

public:
  void initialize(unsigned limitPin, unsigned lineTrackRightPin, unsigned
lineTrackLeftPin);
  int limitSwitch();
  double trackerRight();
  double trackerLeft();
};
```