TEAM 1868

Space Cookies

DESIGN BINDER

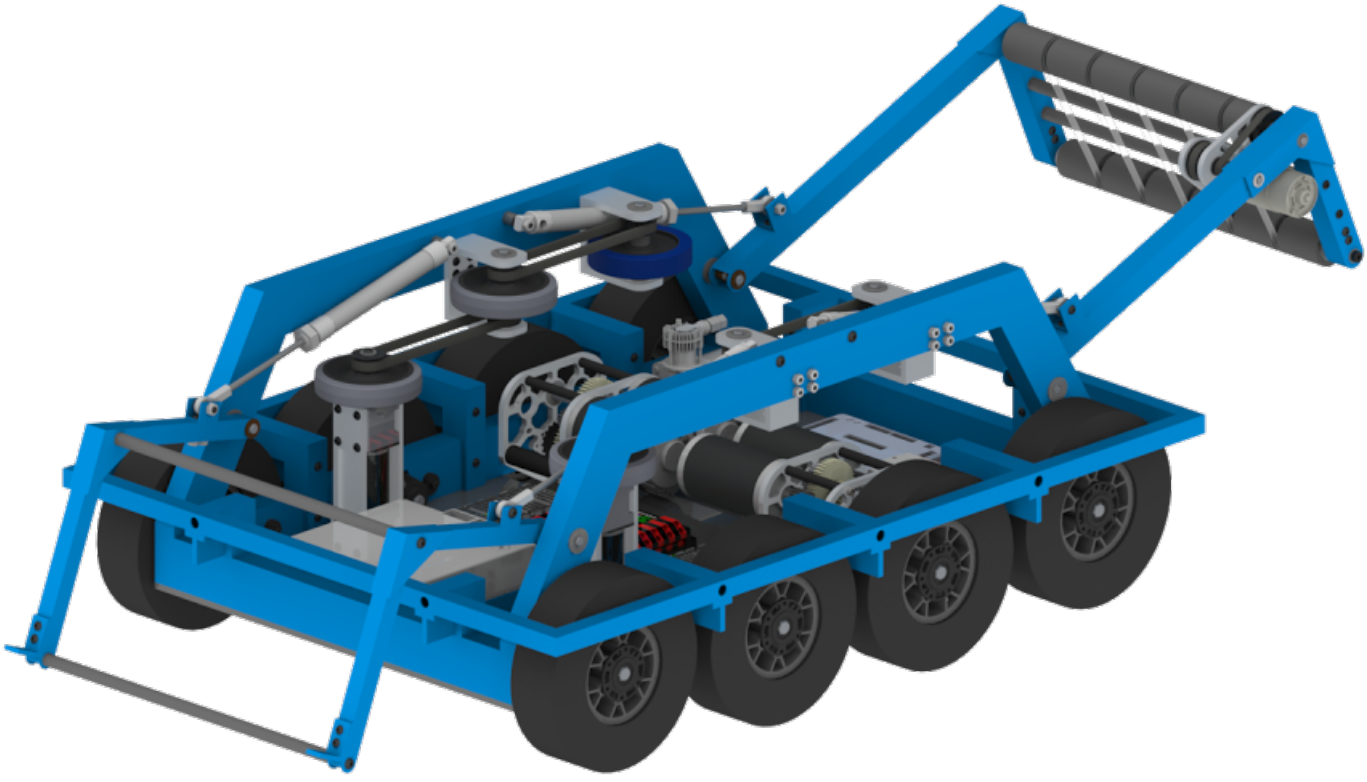
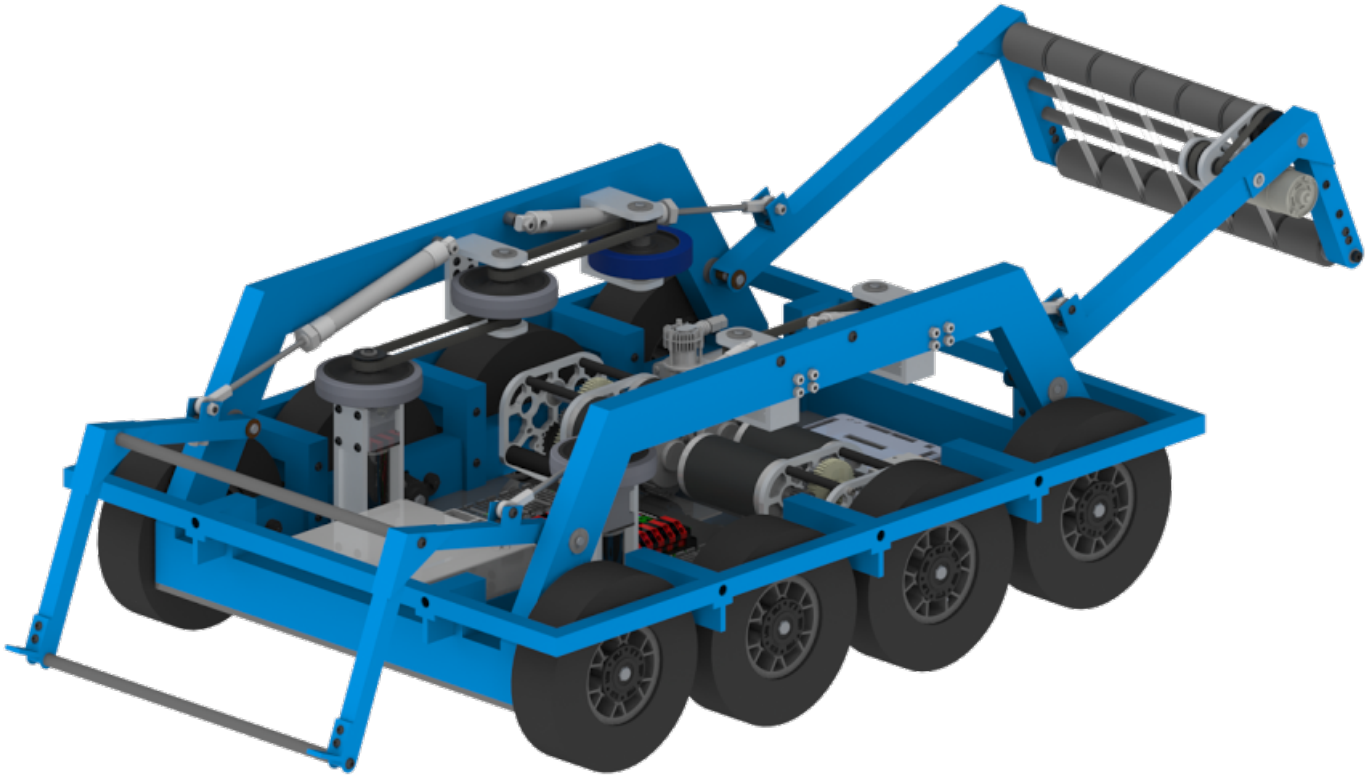


Table of Contents

Mechanical

Programming



MECHANICAL

Strategy

We began the 2016 build season by having our entire team read the game manual to compare the number of ranking points and match points we could get from different strategies. Using a weighted objective table, we analyzed the possible point values against their respective difficulties, and created the following prioritized list:

1. Breach defenses
    a. Cross both defenses in at least 4 categories
    b. Never get stuck
    c. Maintain control of robot while on defenses
2. Score boulders into low goal
    a. Intake boulders quickly
    b. Maintain control of boulders while crossing defenses
    c. Score boulders without driving completely up the batter
3. Challenge
    a. Drive up batter easily
    b. Maintain position on batter after match ends

Design Decisions

Drivetrain

With breaching as our top priority, choosing a drivetrain that could cross all defenses was crucial. By watching Ri3D teams, we narrowed our options to WCD or tank treads, and then prototyped different options to arrive at our final design. Major pros/cons brought up are outlined below, with our final decision bolded.

	Tank tread	**WCD**
Pros	■ Could cross defenses ■ Possibly more control	■ Could cross defenses ■ Relatively easy/fast ■ Proved to work early on with our own prototype
Cons	■ Difficult to design/fabricate ■ Have not done before ■ Expensive (especially COTS options)	■ Wheel size/spacing would be crucial

	6" wheels	**8" wheels**
Pros	■ More room for mechanisms above frame	■ Cleared all defenses ■ More wheel options
Cons	■ May not clear rock wall and rough terrain	■ Higher COG

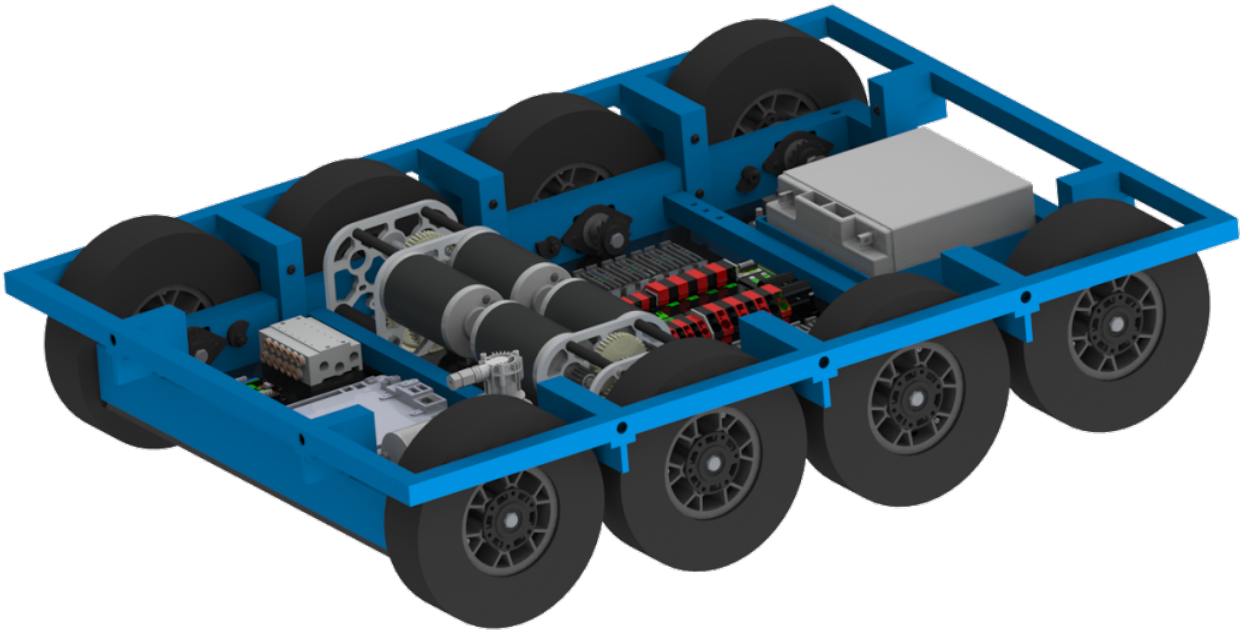
	6WD	**8WD**
Pros	■ Allowed us to be shorter ■ Easier to fit on batter	■ Cleared all defenses ■ More stable (four dropped wheels instead of two)
Cons	■ May not clear rock wall and rough terrain	■ More maintenance

	Traction wheels	**Pneumatic wheels**
Pros	■ Easy to get ■ Easy to maintain in pit	■ Cleared all defenses ■ Maintained more control when crossing defenses
Cons	■ Slipped when trying to clear rock wall ■ Less traction on polycarb (harder to cross defenses and to challenge)	■ Many brands were out of stock ■ Many wheels were either too small or too big (varied from 7.5 to 8.5")

	Short and wide	**Long and narrow**
Pros		■ Number and size of wheels dictated minimum length ■ Thankfully, everything fit! (just barely)
Cons		■ Gearboxes required a minimum width

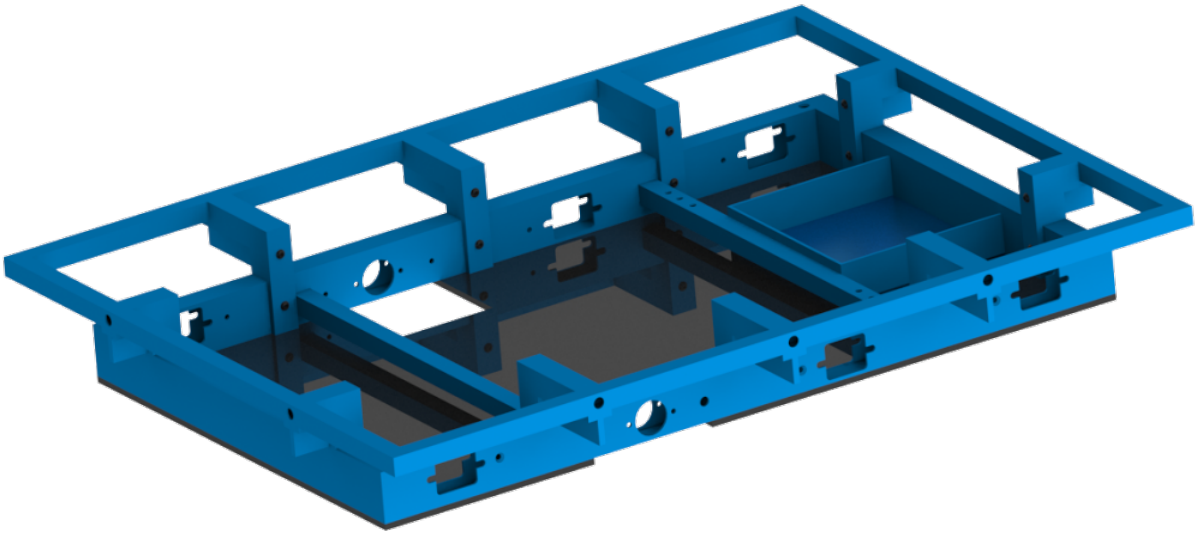
	Single speed	**Double speed**
Pros	■ Smaller ■ Broken up field may mean we don't need to go as fast	■ Faster cycles ■ More force for / to avoid defense ■ Could use whichever was needed for defenses
Cons	■ Would have to compromise with a middle speed	■ More maintenance

Drivetrain



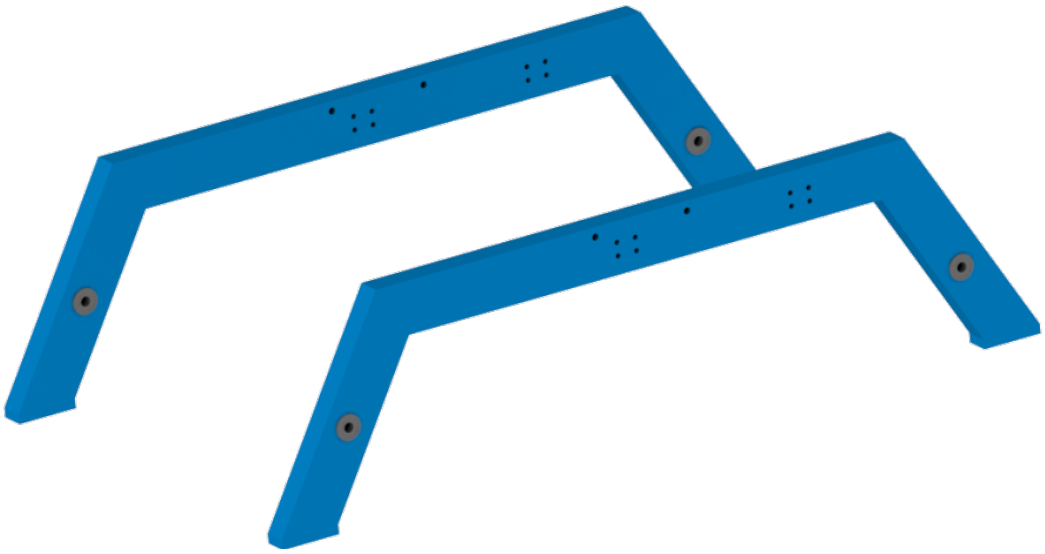
- 8 wheel WCD
- 200 x 50mm pneumatic wheels using AndyMark pneumatic wheel hubs and VEXpro VersaHubs
- 5/16” drop on center 4 wheels
- 8.25” wheel spacing
- West Coast Products 2 CIM DS gearbox, geared to 7.95:1 and 16.63:1 (adjusted speed 18.9 and 9.2 fps)
- Designed to cross all defenses

Chassis



- 25.75” x 33.875” frame perimeter
- 1”x2”x1/8” frame with 3/4”x3/4”x1/16” supports
- 1/4" polycarbonate base
- Bumper rails are held on by 1/4-20 bolts so they can be removed for easy maintenance
- Designed to maximize base area while fitting onto the batter to challenge

Superstructure



- 8 1/2" tall at 60 degree angles
- 1x2x1/8” tubing
- Designed to maximize height while clearing the low bar, optimizing mechanism mounting locations

Defense mechanisms:

With our prototypes for a category C mechanism requiring a tall robot, we realized that to meet our initial goal of crossing 4 defenses, we would be picking one or the other. After realizing that the sally port door could be crossed by careful driving, we decided that the risk of designing for the drawbridge without an official version to test on wouldn't be worth it.

	Category C mechanism	**Low bar capable**
Pros	■ More height for other mechanisms ■ Easier to fit ball within the robot	■ Always on the field ■ Always in the same place ■ Fastest defense to cross ■ Closest to secret passage--faster low goal cycles ■ COG closer to the ground
Cons	■ Required a separate mechanism ■ Harder to design for ■ Harder to do consistently ■ Could be done by two robots ■ Sally port door could be crossed by strategic driving	■ Harder to see robot on field behind defenses

	Integrated intake / Category A	**Separate Category A mechanism**
Pros	■ Fewer mechanisms to design and maintain ■ Possibility for dual intake	■ Easier to design for ■ Likely more reliable ■ Allowed us to intake from back, saving time in low goal cycles
Cons	■ More dependence on a single mechanism	

Defense Manipulator

- Controlled by two 3/4” bore 4” stroke pistons mounted to the superstructure
- Uses symmetrical mounting locations to the intake for simplicity
- Designed to push down Cheval de Frise and lift the portcullis

Intake and shooter:

To meet the final two goals on our priority list, we tested out many different combinations of intake and outtake designs.

	Holding balls outside the robot	**Holding balls inside the robot**

Pros	■ Possibly faster (would not need to go over bumper)
	■ Would not need separate outtake
		■ More control of ball
		  ■ When crossing defenses
		  ■ With defense

Cons	■ Would have to intake/shoot from same side, requiring more turning ■ Harder to design outtake with enough force to shoot from	■ May not fit under low bar

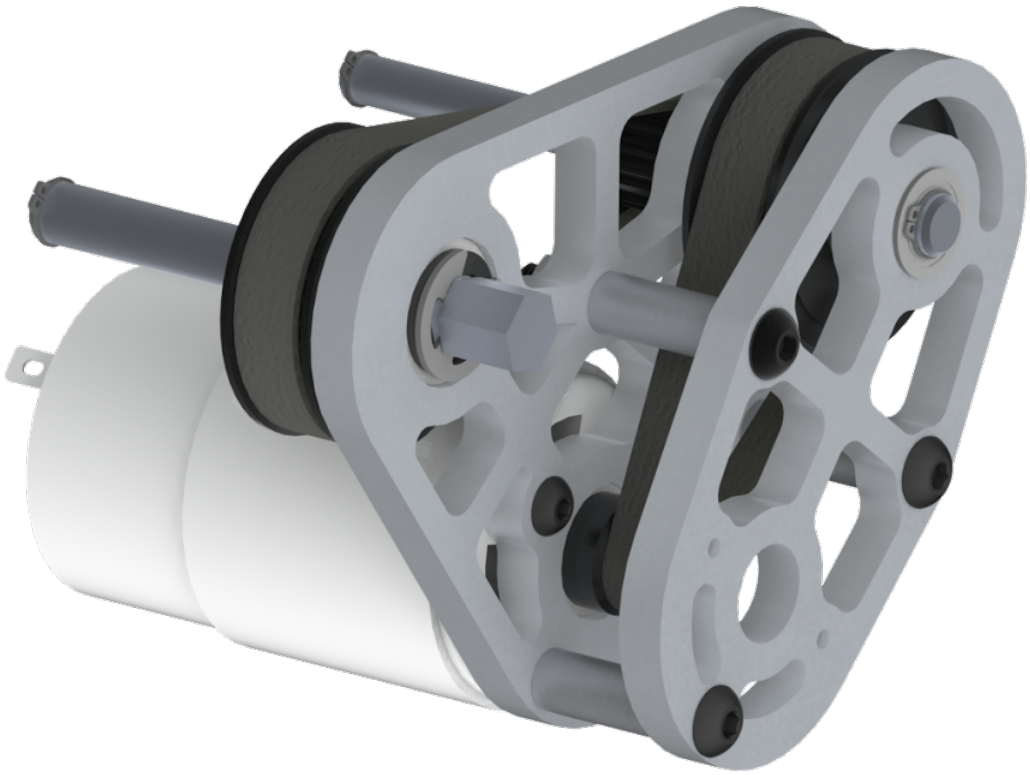
	Intake/outtake same side	**Intake/outtake opposite sides**
Pros	■ Possibly simpler design	■ Could intake from secret passage, drive across field, and score without spinning ■ Faster cycles overall
Cons	■ Ball would have to be able to clear intake when shooting	■ Likely requires separate mechanism

	Combined intake/outtake	**Intake/outtake opposite sides**
Pros	■ Fewer mechanisms to design and maintain ■ Mechanisms would not interfere with each other	■ Can intake/outtake opposite sides more easily ■ Could get more power from the outtake and shoot farther ■ Each design was simpler
Cons	■ Likely has less power ■ Likely have to intake/outtake the same side	■ Designs would need to integrate well--possibly more design work overall

Intake

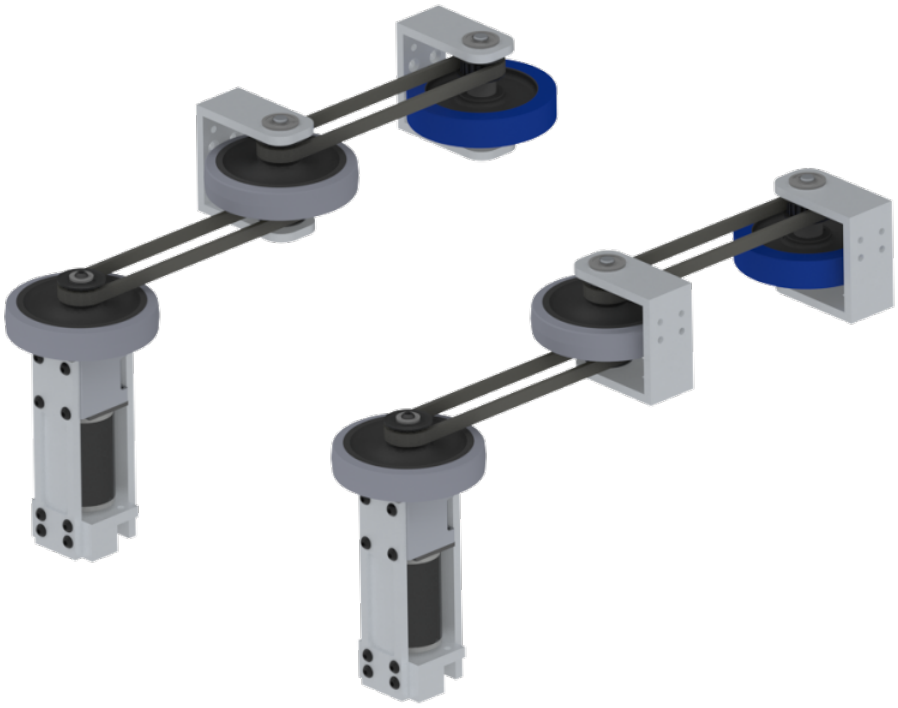
- Pulley box spins the upper roller
- Controlled by two 3/4” bore 4” stroke pistons mounted to the superstructure
- Rollers are soft surgical tubing over aluminum tubes
- Polycord connects rollers and provides constant contact with ball
- Designed to intake ball over bumper from as far out as possible

Pulley Box



- Powered by 775pro at 7.51:1 reduction
- Uses WCP 3mm GT2 pinion and belts with SDP-SI pulleys
- Engages directly into broached cap in intake roller tube
- Designed to minimize how far into the intake the belts were, but also how far out the motor was

Outtake



- Powered by two BAG motors on VersaPlanetaries at 10:1 reductions
- Two 4” Colson wheels and one 3.75” Banebot wheel grip the ball on each side
- Connected with 5mm HTD pulleys and belts
- Attaches to superstructure and to drivetrain supportDesigned to hold ball inside the robot and outtake into the low goal

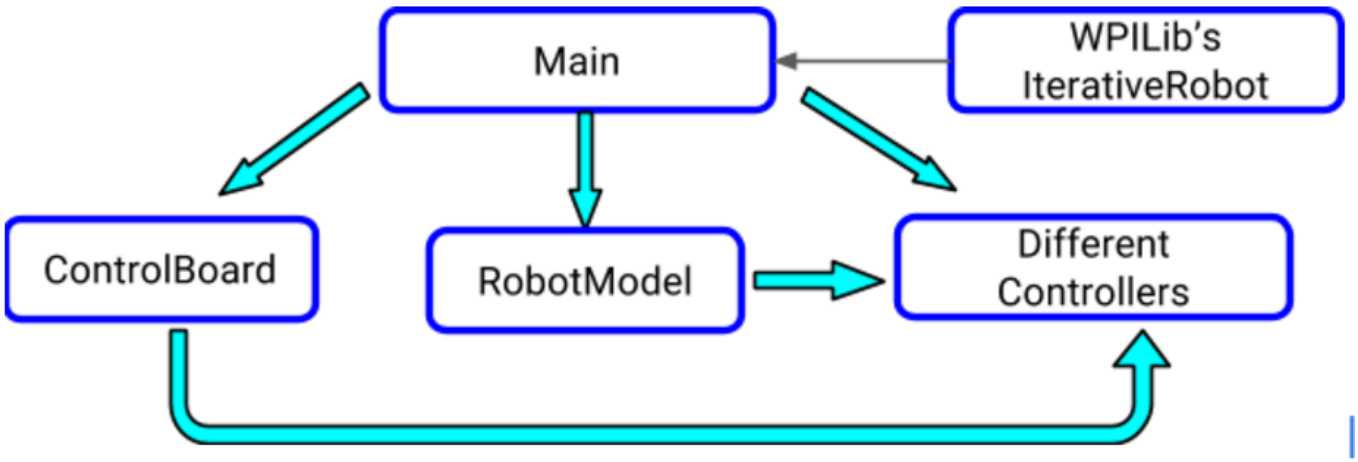
```
void TableReader::FindConvexHull() {
    int size = myPoints.size();
    if(size < 3) {
        return;
    }
    int indexOfFirstPoint = findLowestY();
    std::swap(myPoints[indexOfFirstPoint], myPoints[0]);
    firstPoint = myPoints[0];
    std::sort(myPoints.begin() + 1, myPoints.end(), comparePoints);
    std::vector<MyPoint> temp;
    temp.push_back(myPoints[0]);

    int i = 1;
    while(i < size - 1) {
        int orientation = findCounterClockwise(myPoints[0], myPoints[i], myPoints[i + 1]);
        if(orientation != 0) {
            temp.push_back(myPoints[i]);
        }
        i++;
    }
```

PROGRAMMING

Background

Framework

Our code is built on the Iterative Robot base class from the WPI library. In our framework, we have a ControlBoard class, which takes input from the driver station, a RobotModel class, which controls the functionality of the robot, and multiple controllers that combine ControlBoard and RobotModel to control our robot. The DriveController uses joystick and button inputs to control the robot's drive system. The SuperstructureController controls all of the secondary mechanisms. In addition, the AutonomousController creates a queue of commands that are run in autonomous.



Autonomous Structure and Interface

Problem: We wanted to have all of the functionality of teleop and more in autonomous, including the ability to do multiple functions and call on any function.

```
void AutonomousController::Update(double currTimeSec, double deltaTimeSec) {
    if (currentCommand != NULL) {
        if (currentCommand->IsDone()) {
            DO_PERIODIC(1, printf("Command complete at: %f \n", currTimeSec));
            currentCommand = currentCommand->GetNextCommand();
            if (currentCommand != NULL) {
                currentCommand->Init();
            } else {
                timeFinished = currTimeSec;
            }
        } else {
            currentCommand->Update(currTimeSec, deltaTimeSec);
        }
    } else {
        DO_PERIODIC(100, printf("Queue finished at: %f \n", timeFinished));
    }
}
```

Solution: Our autonomous is essentially a queue of commands, each of which inherits from one class, AutoCommand. Because all of our commands follow this basic interface, they can be easily used in the AutonomousController. However, our autonomous is not just a queue of commands. By creating different subclasses of commands, we can change the queue depending on sensor feedback and run multiple commands at once.

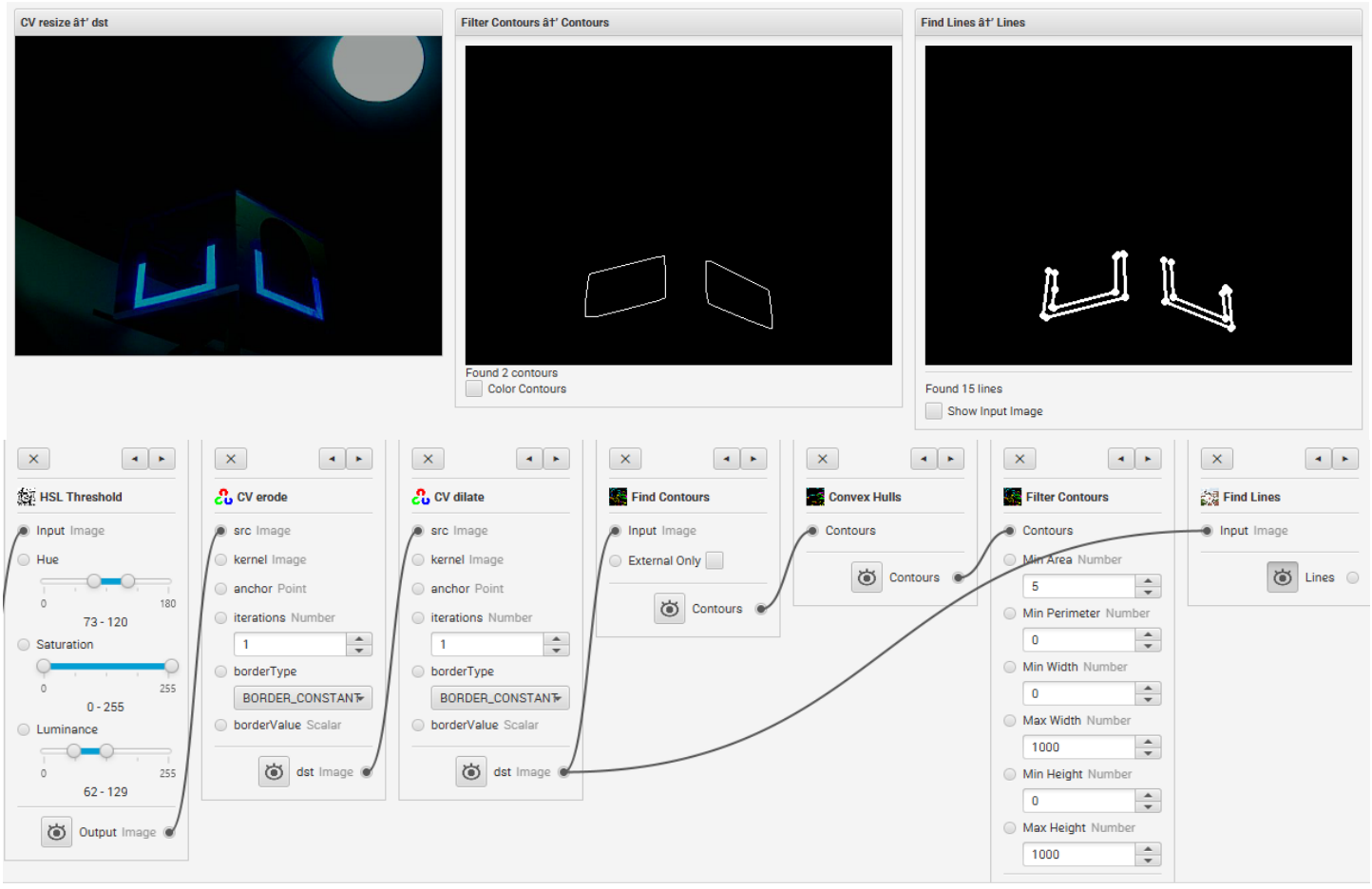
The robot acquires knowledge of the situation and environment to act successfully.

Vision

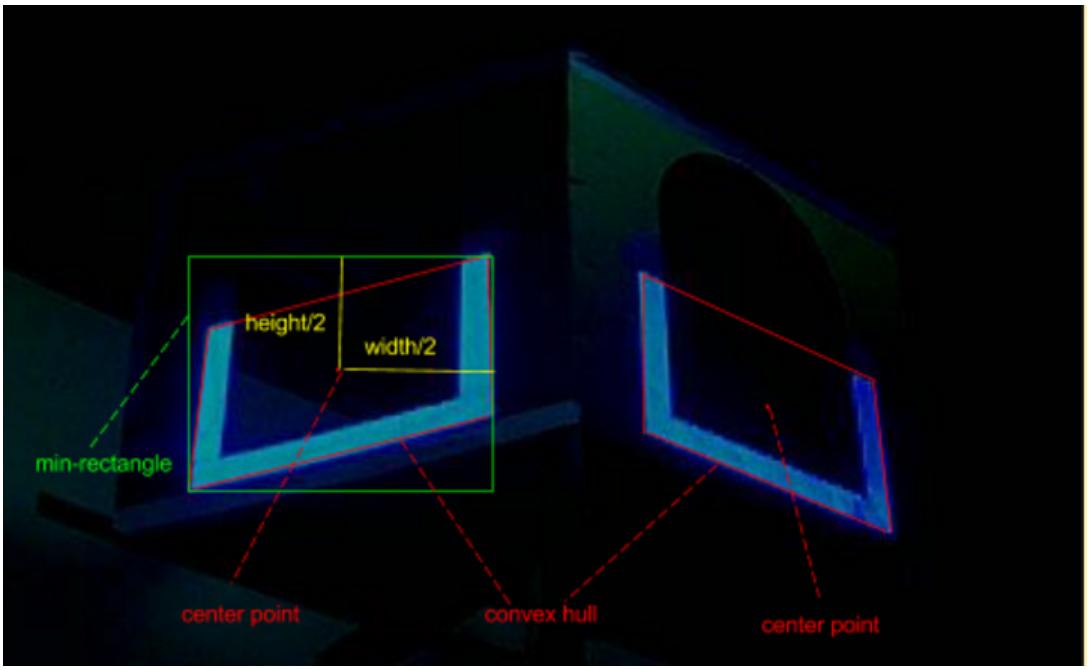
Problem: We need to know our position on the field at various times to be able to line up with the goal in autonomous.

Solution: Using vision targeting, we obtain our distance from a high goal target. To do this, we split our image analysis into categories: GRIP algorithm, post-processing and distance algorithm.

1. GRIP algorithm: To determine where the target is, we first process the image using GRIP. We run the image through a sequence of processes to find lines and contours.



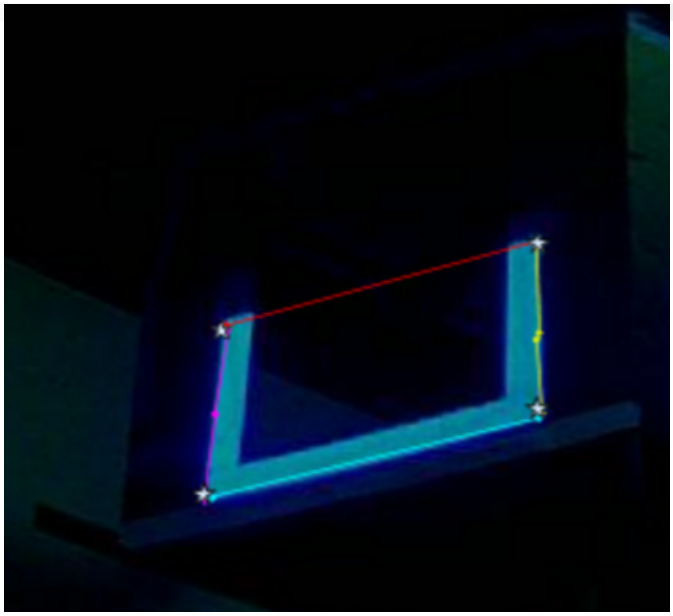
2. Post-processing: To post-process the image, we run many different processes to isolate the image coordinates of the four outer corners of the desired target. First, we need to isolate a specific target. Because we are lining up with goals underneath the left and right target, we first input which goal we go under. Then, using the coordinates of the centers of our convex hulls, we identify the left or right target.

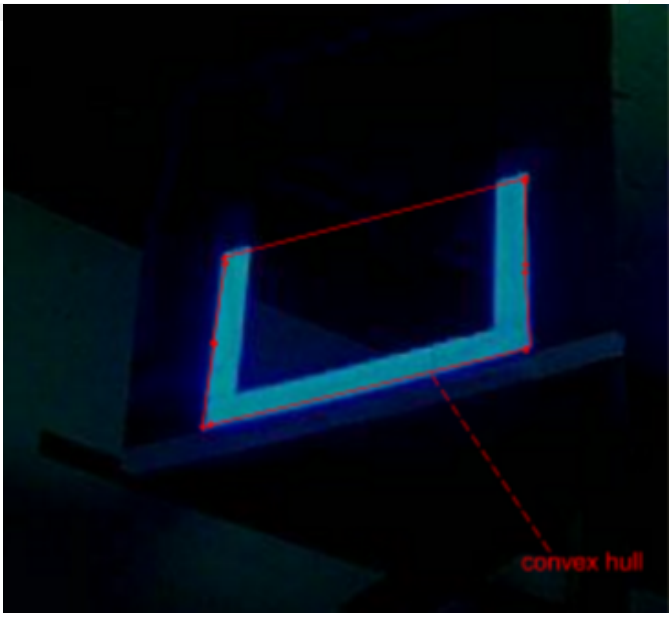


Next, using the fact that contour's publishes the width and height of the min-rectangle, we identify which points from the lines report can belong to the specific target.

Next, we wrote code following the Graham Scan Algorithm to find the convex hull. With this we could also gain the coordinates of the vertices we wanted, which were not provided by the contours.

```
void TableReader::FindConvexHull() {
    int size = myPoints.size();
    if(size < 3) {
        return;
    }
    int indexOfFirstPoint = findLowestY();
    std::swap(myPoints[indexOfFirstPoint], myPoints[0]);
    firstPoint = myPoints[0];
    std::sort(myPoints.begin() + 1, myPoints.end(), comparePoints);
    std::vector<MyPoint> temp;
    temp.push_back(myPoints[0]);

    int i = 1;
    while(i < size - 1) {
        int orientation = findCounterClockwise(myPoints[0], myPoints[i], myPoints[i + 1]);
        if(orientation != 0) {
            temp.push_back(myPoints[i]);
        }
        i++;
    }

    temp.push_back(myPoints[size - 1]);
    int size2 = temp.size();
    if(size2 < 3) {
        return;
    }
```



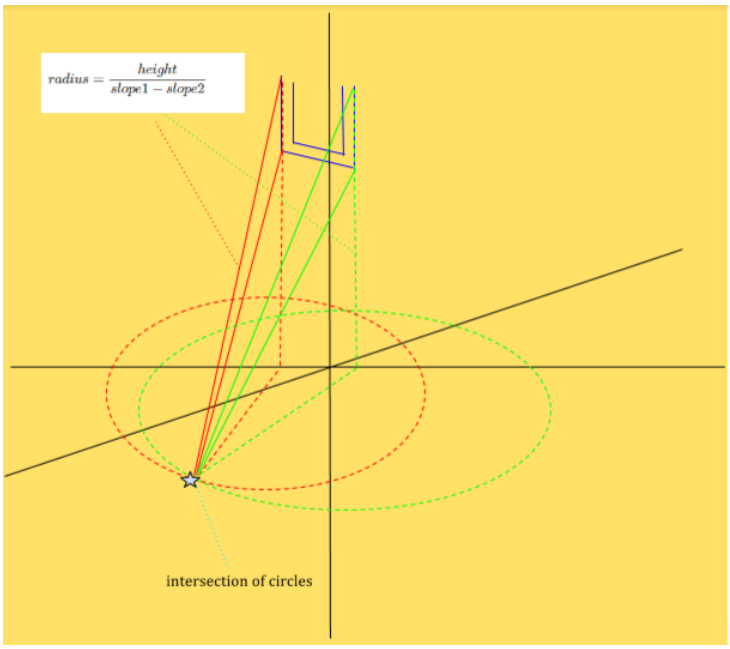



```
    temp.push_back(myPoints[size - 1]);
    int size2 = temp.size();
    if(size2 < 3) {
        return;
    }

    myConvexHull.push_back(temp[0]);
    myConvexHull.push_back(temp[1]);
    myConvexHull.push_back(temp[2]);

    for(int i = 3; i < size2; i++) {
        int size3 = myConvexHull.size();
        while (-1 != findCounterClockwise(myConvexHull[size3 - 2], myConvexHull[size3 - 1], temp[i])) {
            myConvexHull.resize(size3 - 1);
            size3--;
        }
        myConvexHull.push_back(temp[i]);
    }
}

void TableReader::printMyConvexHull() {
    print(myConvexHull);
}
```

Next, we split the segments between consecutive points of our convex hull into four groups -- top, left, bottom and right. Then we run least-squares regressions to find the edges of our target and find the intercepts of these lines.

3. Distance algorithm: Using the image coordinates of the four corners, we implement the pinhole camera approximation to find the corresponding real coordinates with our camera matrix. From there, we calculate the slope of our line in the real plane and our radius to the corresponding edge of the target. Then, we take the two radii and find the intersection of the circles to find our distance from the center of the target.

```
void CameraController::CalculateDistance() {
    table->ReadValues();
    printf("Read Table Values \n");

    leftR = CalculateRadius(table->GetTopLeftX(), table->GetTopLeftY(),
                            table->GetBottomLeftX(), table->GetBottomLeftY());
    rightR = CalculateRadius(table->GetTopRightX(), table->GetTopRightY(),
                            table->GetBottomRightX(), table->GetBottomRightY());
    x = ((rightR*rightR) - (leftR*leftR) - (leftDisFromCenter*leftDisFromCenter - rightDisFromCenter*rightDisFromCenter))
            / (2*(leftDisFromCenter + rightDisFromCenter));
    y = sqrt(rightR*rightR - (x + rightDisFromCenter)*(x + rightDisFromCenter));
}
```

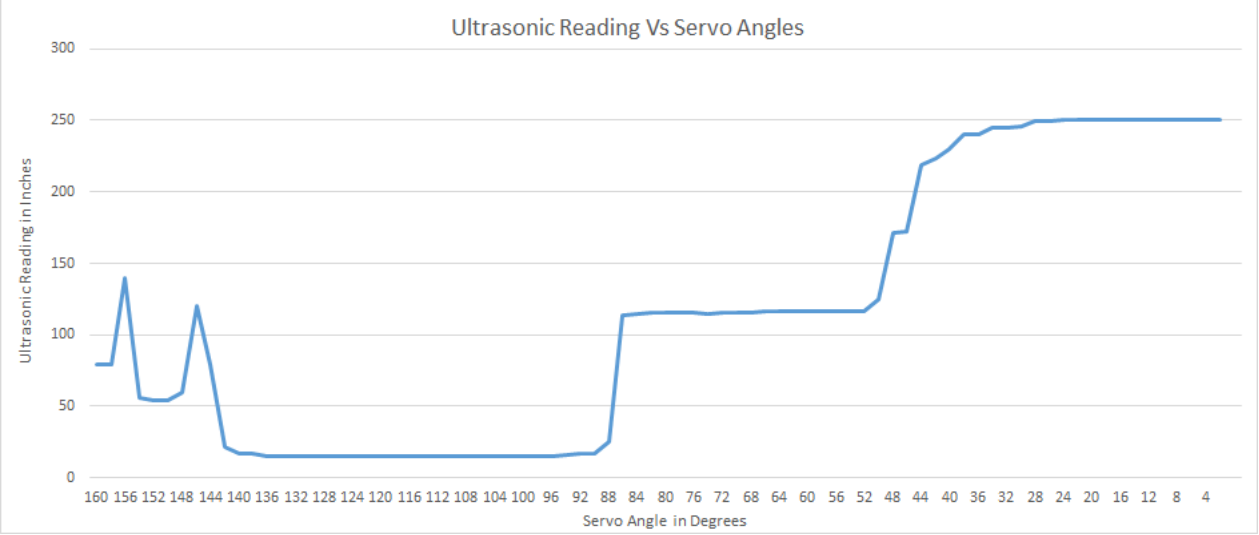


Ultrasonic Sensor

Problem: In order to intake boulders from the midline in autonomous, we must be able to quickly and accurately detect the boulders and know our distance from them.

Solution: By utilizing an ultrasonic sensor to detect our distance from the boulder, we can pick up a boulder in autonomous from the midline without crossing the midline.

How it works: We use an ultrasonic sensor coupled with a servo that allows the sensor to rotate and scan the field in front of us. We determine the angle towards the center of the boulder by averaging the servo angle for the two sides of the boulder. In addition, we find the X and Y distances from the center of our robot to the boulder. Using this information, we can drive and pivot to align our robot to the boulder accurately.



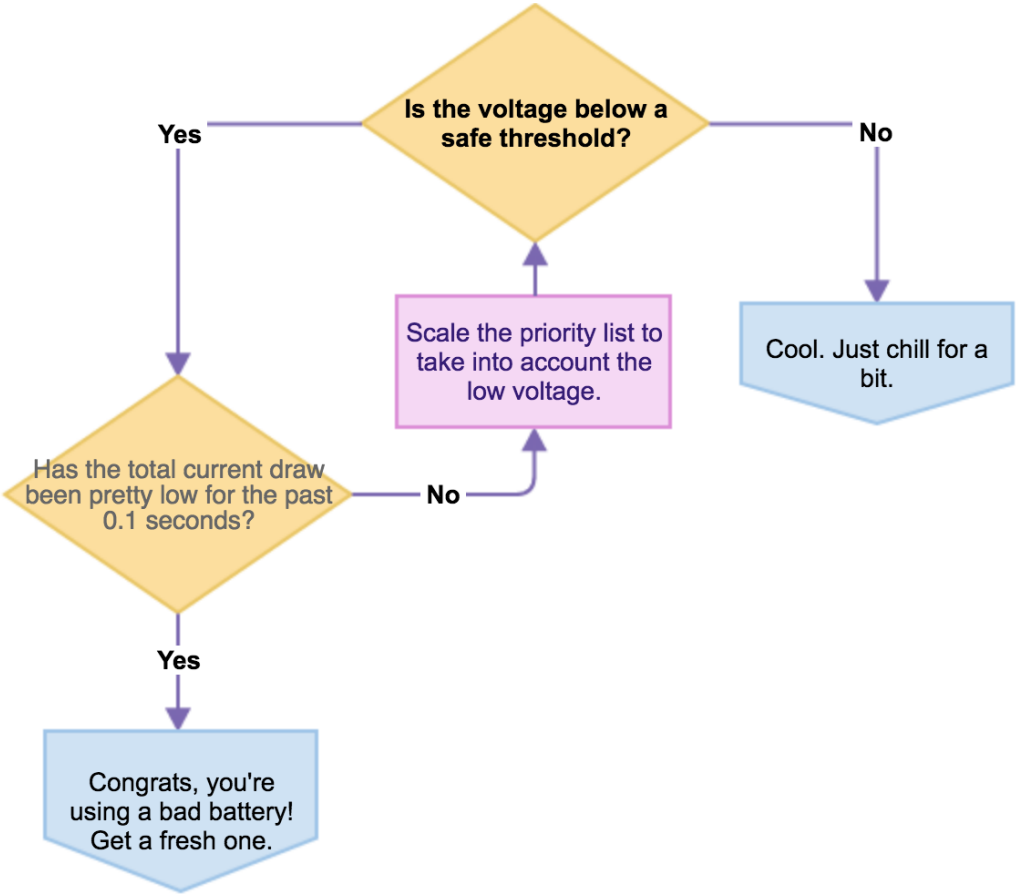
Brownout Protection

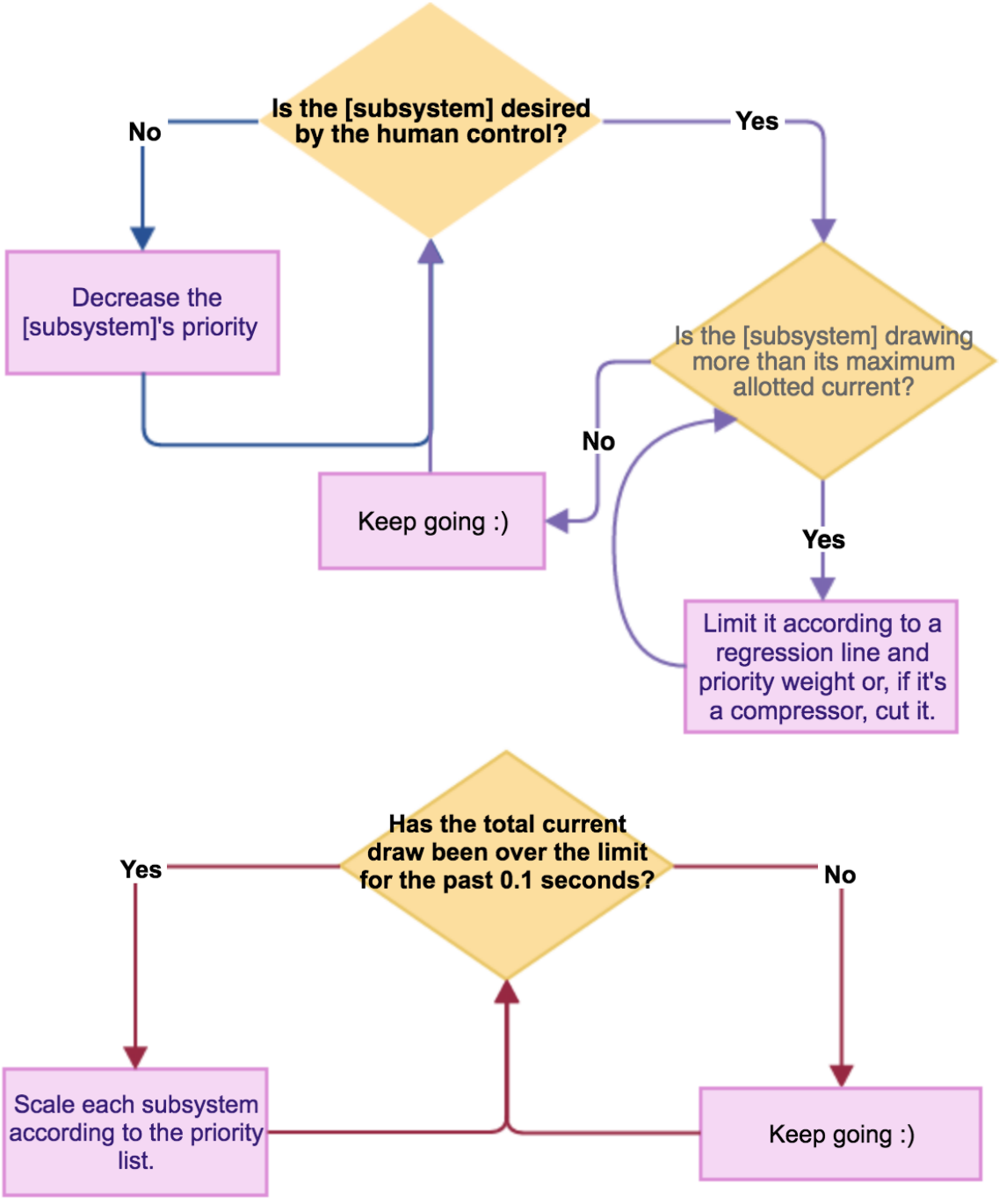
Problem: Due to our 8-wheel drivetrain and aggressive defense-breaching strategy, the drive motors used in conjunction with other robot functions will frequently draw enough current to cause a brownout.

Solution: We developed a current budget that monitors and controls the current flowing to everything on the robot from drive motors to the compressor to the roboRIO. The set of rules includes:

1. Limiting current draw from a single system on the robot (e.g. drivetrain, compressor, intake motors, etc)
2. Scaling the current from all systems according to a priority list
3. Changing the priority order based on whether the drivers are actively using the system, and factoring in battery voltage

The logic of our brownout prevention is a three-step process:





Essentially creating our own version of a pre-brownout procedure that takes place before the roboRIO's built-in procedure, we ensure that we will never experience a brownout while still maintaining a high-performing robot.

Accuracy Commands

Problem: In autonomous, we want to know our exact position on the field at all times.

Solution: With the creation of different driving commands, we can successfully and efficiently navigate the field while maintaining our knowledge of our coordinates.

Pivot Command:

Problem: We want to turn a specific angle in order to shoot reliably in autonomous while knowing our position on the field.

Solution: We created two autonomous functions to deal with this issue and fit our interface. These functions use a PID control loop and the yaw output of the NavX. The first function pivots to a specific angle. At the beginning of each match, we zero the yaw of the NavX, so during a specific match, every time a specific angle is entered into the function, the robot with pivot to the same angle. The other function is used to turn a specific change in angle. For these functions, we make sure that we always turn to the desired angle taking the shorter way around.

Drive Command:

Problem: We want to drive a specific distance and control our drift during autonomous.

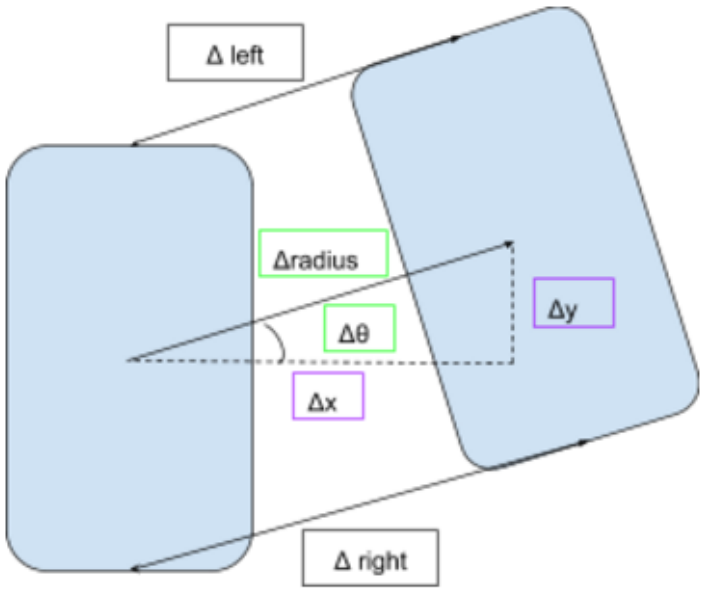
Solution: Similar to PivotCommand, we use a PID control loop on both the encoder distance values and the NavX yaw output in order to drive straight and reach a specific distance.

Curve Command:

Problem: While combinations of pivoting and driving are accurate, they don't take the optimal math to get from one point to another on the field.

Solution: We created a command that drives a specific horizontal and vertical distance using differential odometry.

To do this, we have to calculate our change in x and y coordinates. We use the following equations to estimate the change in the distance of our center of mass:



$$x_{now} = x_{last} + \Delta x = x_{last} + \frac{\Delta left\ encoder + \Delta right\ encoder}{2} sin(\theta)$$

$$y_{now} = y_{last} + \Delta y = y_{last} + \frac{\Delta left\ encoder + \Delta right\ encoder}{2} cos(\theta)$$

These equations were modified for our robot because the angle, on our navX is measured increasing clockwise with 0oon the y axis. So, x corresponds to sine and y corresponds to cosine. Once we have our x and y changes, we can use a PID control loop to control our radius and angle towards a desired point on the field. This conversion into polar coordinates allows us to travel a specific change in x and y. Of course, this command does take a curved path since the desired angle change changes along with the most recent x and y values. This curved path is to optimize the amount of time we spend driving and pivoting to reach a specific point.

Control

The robot uses feedback to accurately and efficiently respond to the drive team.

```
bool PIDControlLoop::ControlLoopDone(double currentSensorValue, double deltaTime) {
    if (fabs(desiredSensorValue - currentSensorValue) <= pidConfig->desiredAccuracy) {
        timeCount += deltaTime;
        if (timeCount >= pidConfig->timeLimit) {
            return true;
        } else {
            return false;
        }
    } else {
        timeCount = 0;
        return false;
    }
}

double PIDControlLoop::Update(double currValue, double desiredValue) {
    double error = desiredValue - currValue;
    error = Saturate(error, pidConfig->maxAbsError);
    double diffError = 0.0;
    if (oldError != 0.0) {
        diffError = error - oldError;
        diffError = Saturate(diffError, pidConfig->maxAbsDiffError);
    }
    sumError += error;
    if (pidConfig->iFac > 0.0) {
        sumError = Saturate(sumError,
                (pidConfig->maxAbsITerm / pidConfig->iFac));
    }
    double pTerm = pidConfig->pFac * error;
    double iTerm = pidConfig->iFac * sumError;
    double dTerm = pidConfig->dFac * diffError;
    double output = pTerm + iTerm + dTerm; //  PID
    output = Saturate(output, pidConfig->maxAbsOutput);
    if (Abs(output) < pidConfig->minAbsError) {
        output = 0.0;
    }
    oldError = error;
    return output;
}

double PIDControlLoop::Update(PIDConfig* myConfig, double currValue, double desiredValue) {
    pidConfig = myConfig;
    return Update(currValue, desiredValue);
}
```

Tank/Arcade Drive

Problem: We wanted to drive in both tank drive, to have the ability to independently control the left and right sides of our robot, and arcade drive, the easier program for our driver to control. This functionality from tank drive would allow us to breach defenses quickly and effectively.

Solution: We added a button to switch between arcade and tank drive to give us maximum flexibility in driving during our matches.

PID

Problem: Inaccuracies in the electrical and mechanical systems prevent the robot from achieving accurate motion without sensor feedback.

Solution: We use proportional-integral-derivative (PID) control loops to correct for current, past and future errors. We implement position and velocity PIDs for driving in autonomous and teleop to prevent drift and more accurately control the robot's systems. We write our own PID controller to allow us to and tuning our proportional, integral and derivative factors for each scenario, add a changing desired value, and confirm that we stay at that value.

Automation

We minimize human error during teleop by automating more functions.

Defense Button, Autonomous Mode Switches

Problem: With obstructed visibility on the field, it is hard to cross the defenses reliably using solely human control.

Solution:

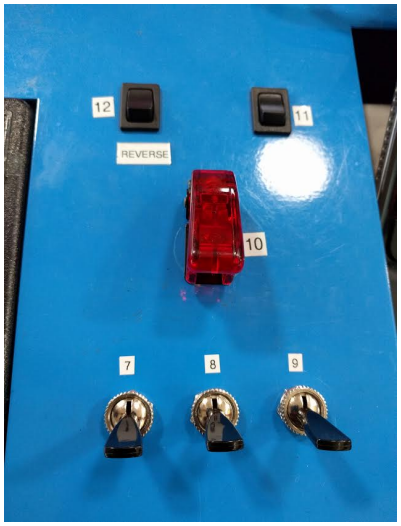
Our autonomous sequences involve crossing the defenses in the quickest and most reliable way and depend on sensors values from the robot. By adding buttons to call the corresponding autonomous functions, we increased our reliability and speed in teleop.

How it works:

1. Autonomous Command: The autonomous command for crossing defenses uses many sensors and implements existing control functions. We use a PID Control Loop on our angle to drive straight over the rock wall and rough terrain. We use the pitch of our navX to find when we have come off the ramp of the defense. In addition, we use our defense manipulator to lift the portcullis and put down the cheval de frise. By fusing our sensor values and different methods of control developed for many areas of our robot, we can reliably cross over the defenses in autonomous.

```
DefenseCommand* cross = new DefenseCommand(robot, superstructure, humanControl->GetDefense());
```

Integrating autonomous and teleop: Through a sequence of switches on our driver station, we can set which defense we plan to cross before and during a match. Our framework allows us to create autonomous commands and implement them in teleop; thus, we can take the functionality of defense crossing in autonomous and apply it to teleop.



Dial-to-Angle

Problem: Although our extremely fast quickturn is useful, it is not accurate in turning to a specific angle. In addition, the driver may have difficulties viewing the robot due to the various defenses and the robot's short build.

Solution:

The robot's pivoting is controlled by a dial on the driver station. Either the driver can turn a dial on the driver station and press a button, causing the robot to pivot to a corresponding angle, or the driver can press a switch and turn the dial, and the robot will pivot in parallel with the dial. The yaw value of the robot's NavX is zeroed at the beginning of a match, so the driver always has a reference for what angle the robot is pointing at, even if they cannot directly see the robot. The robot pivots using our Pivot Command (see Accuracy Command) to turn to the angle that the dial indicates.

The connection between humans and the robot is flexible, powerful, and easy to use.

Interface

On-field Software Configuration (.ini)

Problem: Match strategy often happens in the minutes or seconds before the game starts. In this year's game, the drive team will only know which defenses they will face in the match right before they go onto the field. With field-specific autonomous sequences, there is usually not enough time to deploy the correct sequence onto the robot.

Solution:

A .ini file provides key-value pairs that the robot parses as variables for autonomous mode, PID tuning values, and more.

The INI file allows for an easy and simple way to rapidly change constants, control loops, and states. It gives us the flexibility to make last minute strategic decisions during matches as well. For instance, if we discover that one of our opponents cannot cross a certain defense right before a match, we have the ability to quickly change our autonomous state to cross the defense instead.

In order to quickly change constants and reconfigure robot settings or states, there are a few elements needed in addition to the INI file itself.

Since last year, our INI file has almost doubled in size and helped us make our processes much more efficient.

INI File Implementation Elements

Robot INI File (robot.ini):

This is the standard configuration file on our robot that certain values are retrieved from. Conveniently, the INI file can be edited via a text editor, such as the default notepad within the Windows OS. Thus, any computer including our team driver station can alter the INI file. The structure for an INI file follows the format as follows:

[SECTION]

KEY = VALUE

The [Section] and Key are essential for finding the value that the code reads. In the API and in this documentation, the "name" for a setting is denoted as the key for a setting, demonstrated below:

```
[DEBUGGING]
enableDoPeriodic = 1

[AUTONOMOUS]
AutoMode = 6
HardCodeShoot = 1
FirstDefense = 4
SecondDefense = 0
UseSallyPort = 1

[SUPERSTRUCTURE]
intakeSpeed = 0.75
outtakeSpeed = 0.99
deltaTimePTO = 1.0
deltaEncoderValOT = 0.0

[PIVOTCOMMAND]
PFac = 0.13
IFac = 0.000
DFac = 0.6
DesiredAccuracy = 3.0
MaxAbsOutput = 0.7
MaxAbsError = 60.0
MaxAbsDiffError = 5.0
MaxAbsITerm = 0.0
TimeLimit = 0.04

[PIVOTTOANGLE]
PFac = 0.13
IFac = 0.000
DFac = 0.6
DesiredAccuracy = 3.0
MaxAbsOutput = 0.5
MaxAbsError = 60.0
MaxAbsDiffError = 5.0
MaxAbsITerm = 0.0
TimeLimit = 0.04
```

Logger

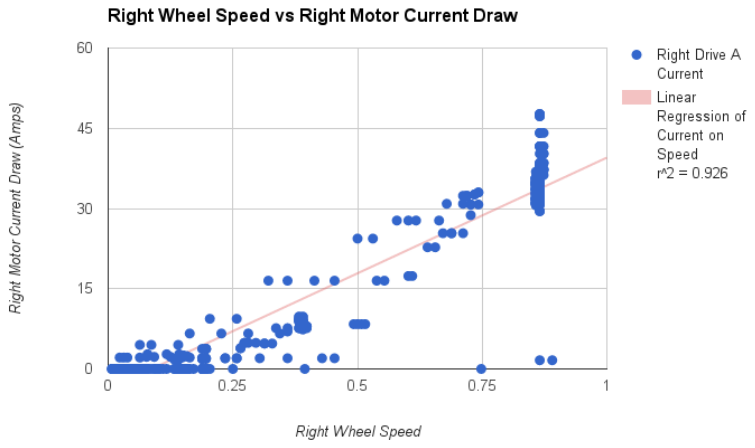
Problem: When something malfunctions during a match, afterwards it is hard to debug without knowing the status of the robot at important times.

Solution:

We created a Logger class to handle both the robot status (e.g. encoder values, joystick values, pressure, etc) and robot actions (e.g. “intaking boulder”, “lowering arm”, etc). This class provides utilities to log data and actions to a timestamped csv file stored on the roboRIO that can be easily interpreted by csv software like Excel or Google Sheets.

```
#define LOG(myRobot, stateName, state) {MATCH_PERIODIC(10, Logger::LogAction(myRobot, __FILE__, __LINE__, stateName, state))}
#define DUMP(stateName, state) {MATCH_PERIODIC(10, Logger::LogAction(__FILE__, __LINE__, stateName, state))}
class Logger {
public:
  static void LogState(RobotModel* myRobot, RemoteControl *myHumanControl);
  /* with time stamp */
  static void LogAction(RobotModel* myRobot, const std::string& fileName, int line,
        const std::string& stateName, double state);
  static void LogAction(RobotModel* myRobot, const std::string& fileName, int line,
        const std::string& stateName, const std::string& state);
  /* without time stamp */
  static void LogAction(const std::string& fileName, int line, const std::string& stateName,
      bool state);
  static void LogAction(const std::string& fileName, int line, const std::string& stateName,
      double state);
  static void LogAction(const std::string& fileName, int line, const std::string& stateName,
      const std::string& state);
```

The macros LOG(myRobot, stateName, state) and DUMP(stateName, state) provide simpler ways to call the LogAction methods that also include the file name and line number from which they were called. There are a variety of methods for LogAction that can take a RobotModel object as an argument if a time variable in the file itself is necessary. LogAction also provides a state value of type bool, double, or string so it can be used in any situation.



A graph on the correlation between wheel speed and current draw generated from Logger file output by Google Sheets