Precision Harvesting Through a Fruit Orchard

Camden Brayton, Tyler Mu, Theodore Mah, Chase Behrens

RBE 1001 Intro to Robotics
Prof. Aloi

We, as Group Three, hereby declare that we all contributed equally to the design tasks of this project

Signed: Camden Brayton Chase Behrens Theodore Mah

Introduction

For this project, our robot must complete the task of navigating the orchard, while picking at least one fruit, and placing it in the box of matching color. To complete this goal, our group decided to use a holonomic drivetrain with a four bar lifting mechanism. Attached to this lifting mechanism is a jaw gripper, chosen for its simplicity. Our robot's main complexities come in the form of its control mechanism. First, to correct for drift, we have an IMU using P control, and a shaft encoder using PI control (See *Appendix C, Figure 1*). Next, to measure the robot's location, we have a coordinate system in our drivetrain code (See *Appendix C, Figure 2*). Lastly, we have an arm controlled by a vision sensor to properly orient our robot to pick the fruit. Overall, we chose these design specifications to create a robot that could universally handle many of the challenges thrown at it. Henceforth, our group endeavors to build a robot that can navigate the orchard, pick one of each fruit, and place them in their respective baskets, based on color.

Solutions and Justifications

In order to complete the challenge, our team had to design and implement various subsystems and algorithms to successfully navigate the orchard, lift an end effector to a range of heights, manipulate the fruit, and detect the robot's surroundings. To achieve this, we wanted to design systems within specific constraints and with varying considerations (*See Appendix A, Table 1*).

Navigation

The core functionality of our drivetrain is to effectively navigate throughout the orchard . In order to achieve this, we designed a holonomic drivetrain and integrated various sensors to assist in navigation and obstacle detection through the orchard. While conceptualizing how we should approach navigation, we considered two primary drivetrains: standard tank drive and holonomic drive (See *Appendix A, Table 2*). To weigh these options we used a design matrix of which we rated both options in various relevant categories (See *Appendix A, Table 3*).

Ultimately, we decided to implement an x-drive type drivetrain (See *Appendix D, Figures 1-3*) as we determined that the omni-directional movement would be essential to moving around the orchard. While it sacrifices torque and power of the drivetrain, the nature of the task favors mobility over torque and power. To keep track of where the robot is positioned on the table we used a basic coordinate system (See *Appendix C, Figure 2*). As the robot drives, we use its velocity and direction to calculate position every update cycle of 50 milliseconds. We mapped out where we wanted the robot to drive in the orchard and did a combination of guess-and-check and math to figure out the coordinates for the robot's pathing.

Lifting

Another core functionality that our system needed was the ability to lift an end effector up to the level of the fruits for harvest. To accomplish this we constructed a double reverse 4-bar lift, which we later simplified to a 4-bar lift. The arm used a vision sensor to control the elevation of the end effector. The primary advantage of a 4-bar lift type is that due to the sets of parallel bars, the end effector is kept parallel to where the lift was anchored (in this case parallel to the table). One primary challenge that we encountered with the lift was designing a system with

enough torque to move the arm while holding the fruits. The calculations to determine final torque required can be found in Appendix B.

While conceptualizing, we considered several options for lifts including scissor lifts, rack and pinions, and 4-bar lifts. We rated each option based on specific criteria (See *Appendix A, Table 4*) and ultimately decided to implement a 4-bar type lift due to simplicity of construction, and its ability to keep the end effector level with the ground. At first, we implemented a double reverse 4-bar lift (*See Appendix D, Figure 5)* which integrated two separate 4-bar lifts to maximize height and minimize idle space; however, we ultimately simplified it to a single 4-bar lift due to large losses of power from friction and weight (See *Appendix D, Figure 6*).

Lastly, we programmed the arm to lift to a preset height based on the position of the targeted object relative to the center of the picture (See *Appendix C, Figure 5*) by sorting the measured value into different height brackets.

Fruit Manipulation

To achieve fruit manipulation, we designed a simple motor-actuated, non-parallel jaw gripper that used limit switches to sense objects. The design also used rubber bands to conform to the shape of the fruit and better hold the objects (*See Appendix D, Figure 7)*.

We considered two primary options for our end effector: a parallel jaw gripper and a non-parallel gripper. Both types are simple to build and motor-actuated, with the primary difference being that a parallel gripper keeps the claws of the gripper parallel to each other to ensure even contact. After comparing our options with a design matrix (See *Appendix A, Table 5*), we decided to construct a non-parallel gripper due to it being simpler to construct and more space efficient. In our code, we utilized timers to stop motor motion once the end effector is open

to prevent motor strain (See *Appendix C, Figure 3*). When closing the end effector we opted for a constant low-voltage input to keep grip on the fruit without overstraining the motor.

Final Implementation

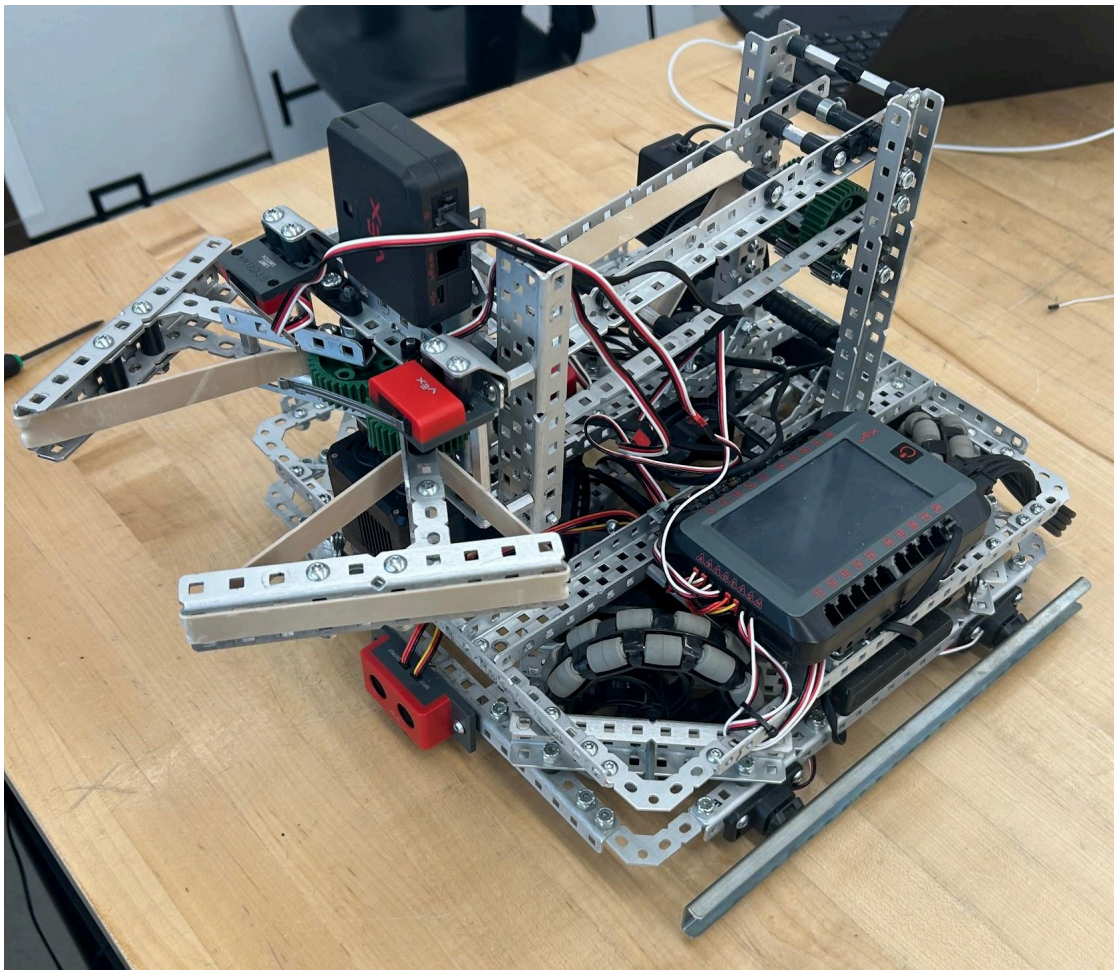


Figure 1: Isometric view of final implementation

Our final implementation that was used for the demonstration consisted of our original holonomic drivetrain that had our modified 4-bar lift mounted to the base with a non-parallel jaw gripper to manipulate the fruit. Our drivetrain utilized 4 standard motors (18:1) positioned symmetrical along the diagonals of the base to allow for holonomic drive, while our arm was

actuated by a single high torque motor (36:1) with an end effector controlled by a standard motor. To assist in navigation we implemented the VEX IMU and shaft encoder to control drift. Additionally, we implemented two bumper sensors on the side of our robot to help us align with walls, as well as an ultrasonic sensor on the front of the robot to help us detect specific distances away from objects. When lifting, we used a vision sensor to control the elevation that the arm raised and had limit switches to control the actuation of the end effector.

System Integration

The robot's state machine consists of a group of main overarching states that each have smaller sub-states that complete basic actions in an ordered fashion.

The "start" state initializes the robot's sensors. Ideally, we wanted the robot to start at a consistent location free from human error. To do this, we used two bumper sensors and a rangefinder to align the robot in the corner. Originally, we had a function that would map the robot's surroundings by creating a dictionary that associated angles with distance values. However, to accurately determine the location of the walls we needed a very high angle resolution which would take too long to find. Using two bump sensors proved to be much more reliable, and was much more time efficient.

In the "moving to row" state the robot navigates to the start of a row. In testing we found that the robot had significant drift. To solve this problem we added an external encoder that would drag a piece of rubber like a caster wheel (See *Appendix D, Figure 4*). This detected the direction the robot was actually moving. We used a PI controller to quickly correct the robot's direction of movement (See *Appendix C, Figure 4*). We also use the IMU to constantly orient the robot in a set direction using P control.

In the “targeting fruit” state we locate and align with a fruit. Our original solution had problems seeing the high branches consistently, so we rotated the camera 90 degrees as the X field-of-view is wider than the Y field-of-view. While this helped, we also raised the arm to its middle position when searching for fruit, which allowed for a greater area to be seen. When centering horizontally we originally used proportional control to center in the field-of-view. However, if the robot lost sight of a fruit, it would quickly pass the fruit. To fix this, we decided to move the robot a small set distance between capturing pictures. When centering vertically, we were originally moving the arm upward until the image was centered. However, because we knew the heights of the trees, we were able to determine a preset height to go to based on one picture.

The “returning to basket” state was entirely reliant on coordinate tracking for the presentation. Because of lack of table space we were unable to adequately test this, and thus the robot had some collisions with the walls. Later, we fixed this by using the ultrasonic sensor to reset coordinate position based on detecting the wall at the end of the row.

Finally, in the “dropping off fruit” state, the positioning of our vision sensor atop the arm prevented us from seeing the color codes on the baskets. Because we could not get a second vision sensor, we opted to use the ultrasonic sensor to detect any basket it could instead.

System Testing Results

As mentioned previously we were experiencing drift issues. To solve this, we first tested the individual motors and found that two were subpar. So, we replaced them. Later, when testing on the table we found that the robot would drift in different directions based on its location.

When calibrating the controller to point the robot in a direction, we found that because our wheels were set up in a way that made turning easy, we only needed proportional control instead of the full PID controller the base bot needed.

When testing fruit identification, we discovered that the camera could lose sight of a fruit for several frames before locking onto it again. The code is set up to check a specific region where the fruit was in previous frames (See *Appendix C, Figure 6*). Because the robot was continuously moving, if it lost its target on a fruit for several frames the fruit would move outside the range the robot expected to see it. This prompted it to target a new fruit. A more complex program that could predict the location of the object based on the robot's movements would have helped with this.

With our first arm, we struggled to get enough torque to raise the arm to the correct heights. Adding higher torque inserts improved the problem, but the reverse 4-bar was still inefficient. Swapping to a single 4-bar reduced the torque problem and while maintaining sufficient reach.

Performance

During the demonstration, our robot exhibited many strong aspects of its design, such as: Near flawless initialization with the corner, allowing for a consistent start position free from human error, fruit recognition, aligning with the fruit, and repeatability. The biggest flaw in our implementation for the demonstration was that we did not get to add another vision sensor to detect what colors matched with what basket. To show our robot as a solid prototype, we coded the robot to always drop the fruit off at the first basket sensed with the ultrasonic rangefinder, which it did consistently (even if it missed the fruit). One main bug plagued us during the run,

however. After the robot grabbed the green fruit from the last row, something went wrong with the coordinate system. The x-coordinate was immediately set to around 200, which made the robot believe it was at the end of the row, which it was not. This caused the robot to proceed to back up through the rows of trees, as opposed to the open path beside them. While this error did end the run, our robot had already proved more than capable of completing the task, as two rows of fruit were navigated and harvested successfully. After the demo, we simply changed the code to have the robot drive in a certain direction and use proportional control, with an ultrasonic sensor, to sense the wall and update its coordinates. This change of code not only fixed our erroneous backing up, but also eliminated some cumulative error the coordinate system faced. During the demo, we picked up the robot so it would not keep running into a tree. However, doing so messed up the encoder, causing some positioning problems with the robot as it tried to drop off the fruit. Debugging the coordinate system and fixing the problem stated earlier would have stopped the robot from running into a tree, removing our need to pick up the robot and ruining the encoder readings. Another slight error in our run was the arm not going to the correct height to grab the fruit. This usually happened when sensing the yellow fruit. It was difficult enough for the vision sensor to pick up on the yellow color due to lighting (See *Table 6*). This led to the arm not raising at all to pick the fruit, and was ultimately due to limited testing of the height ranges for raising the arm. However, this did not end our trial, as the robot would sense it had grabbed nothing and move on. Overall, while not perfect, our robot demonstrated several design aspects working together in unison to complete the task of navigating the orchard and picking fruit. With a little more time, this design could have been built upon and perfected.

Appendix A (Tables and Figures)

Figure 1: Below is the State Diagram decided upon for the robot to complete its task.

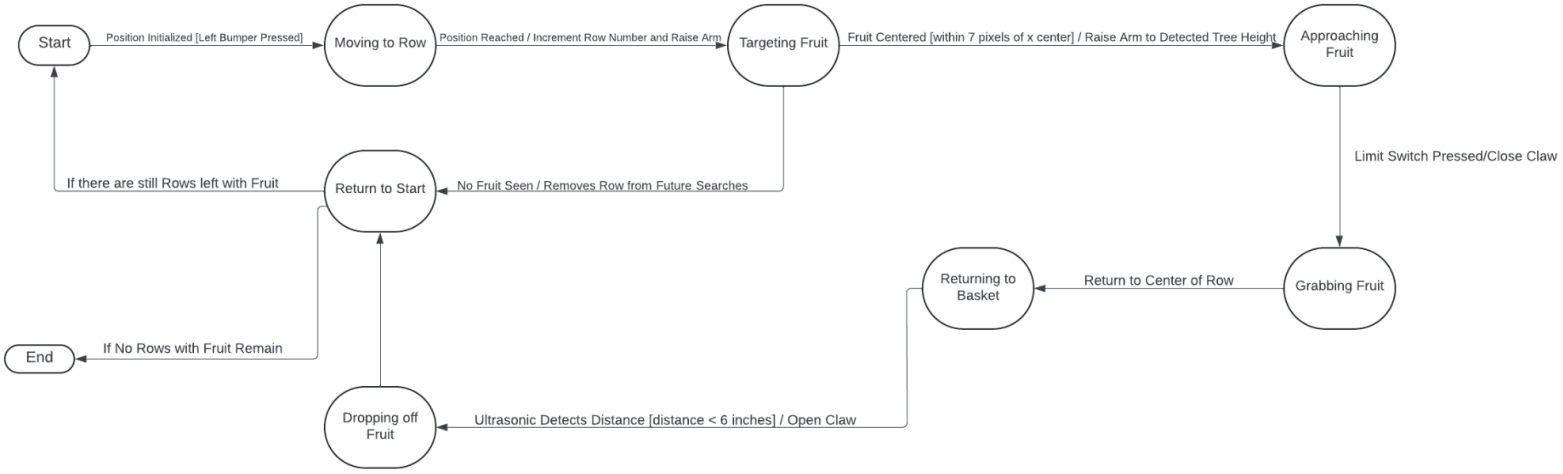


Table 1. Design Constraints and Considerations

System	Constraints
Drivetrain	• Cannot exceed 15.25” x 15.25” • Small enough to fit between trees • As lightweight as possible • Sense obstacles and keep track of its position
Lift	• Can reach all heights of fruits • Enough torque to lift and pick fruit • Lightweight and minimal space
End Effector	• Takes up minimal space • Can detect objects in grasp • Can hold fruits of varying sizes

Table 2. Comparison of simple tank and omni drive

Drivetrain	Simple	Omnidrive
# of motors	4 standard: 18:1 Internal Gear Ratio 200 Max RPM 2.1 Nm Max Torque 11W Max Power	4 standard: 18:1 Internal Gear Ratio 200 Max RPM 2.1 Nm Max Torque 11W Max Power
Direction of motion	Standard tank drive: Linear motion point turning cannot strafe	Omni: Point turning omni-directional motion linear strafing curving motion
Advantages	Simpler design Higher traction Can be geared to increase drive torque or speed	Can drive any direction Light weight Geometrically centered center of mass Space efficient
Disadvantages	Limited mobility Not omni-directional Takes up more space	Cannot be geared up or down Difficult to design Subject to slight mechanical inconsistencies

Table 3. Drivetrain Design Matrix

Drivetrain	Maneuverability	Speed	Traction	Complexity	Total
X-Drive	10	8	6	7	31
Tank Drive	6	8	8	8	31

Table 4. Lifting Mechanism Design Matrix

Lift Mechanism	Needed Torque	Max Height	Weight	Complexity	Space Efficiency	Stability	Total
Rack & Pinion	8	8	9	9	5	7	46
Scissor Lift	3	8	5	6	7	4	33
4-Bar	6	9	8	10	10	10	53

Table 5. Gripper Design Matrix

Gripper	Size	Grip Strength	Complexity	Total
Parallel Jaw	7	8	10	25
Non-Parallel Jaw	9	8	10	27

Table 6. Confusion Matrix for Vision Sensor with Threshold 3

Presented	Trained			
| V	Orange	Lemon	Lime	Grapefruit
Orange	10/10	0/10	0/10	0/10
Lemon	0/10	9/10	0/10	1/10
Lime	0/10	0/10	10/10	0/10
Grapefruit	0/10	0/10	0/10	10/10

Appendix B (Calculations)

Field-of-View Calculations:

Horizontal:

$FOV = 60\ degrees\ (Measured\ with\ Camera)$

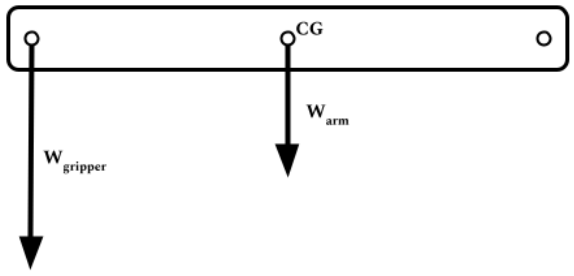
$\frac{60\ degrees}{315\ pixels} = 0.19\ degrees\ per\ pixel$

Vertical:

$FOV = 33\ degrees\ (Measured\ with\ Camera)$

$\frac{33\ degrees}{211\ pixels} = 0.16\ degrees\ per\ pixel$

Max Arm Torque Calculations:



$\tau_{motor} = e(Fd)$

$\tau_{motor} = \frac{12\ teeth}{36\ teeth}(F_{arm}d_{arm} + F_{gripper}d_{gripper})$

$\tau_{motor} = \frac{1}{3}((9.81ms^{-2})(0.38kg)(0.1275m) + (9.81ms^{-2})(0.48kg)(0.255m))$

$\tau_{motor} = \frac{1}{3}(1.66502\ Nm)$

$\tau_{motor} = 0.555Nm$

Appendix C (Code)

Figure 1. PID Controller implementation.

```
def calculate(self, target_value, current_value):
    """Calculates the controller ouput given current and target values.

    Parameters:
        target_value: The value the controller is attempting to reach.
        current_value: The current value.
    """
    if (self.output == self.max_output and self.errorsum > 0) or \
       (self.output == -1 * self.max_output and self.errorsum < 0):
        integral_clamp = 0
    else:
        integral_clamp = 1
    self.error.pop()
    self.error.insert(0, target_value - current_value)
    self.errorsum += self.error[0] * integral_clamp
    self.output = min(self.max_output, max(-1 * self.max_output,
                  (self.proportional_gain * self.error[0]) + \
                  (self.derivative_gain * (self.error[0] - self.error[-1])) + \
                  (integral_clamp * self.integral_gain * self.errorsum) + \
                  (self.min_output * sine(self.error[0]))))
    return self.output
```

Figure 2. Coordinate tracking system.

```
def update_position(self, direction, velocity):
    """Updates the coordinate positioning based on expected movement.

    Parameters:
        direction: The direction the robot is driving in.
        velocity: the velocity the robot is traveling in.
    """
    velocity /= 50
    self.x_position += sin(direction) * velocity
    self.y_position += cos(direction) * velocity
```

Figure 3. Fruit release.

```
def release_fruit(self):
    """Opens the gripper."""
    if self.grabbing:
        self.gripper_motor.spin(REVERSE, 3, VOLT)
        self.grabbing = False
        Timer().event(self.gripper_motor.stop, 1000)
```

Figure 4. Drift correction.

```
# Corrects for drift using a shaft encoder when readings are in a believable range.
def drive_in_direction(self, velocity, direction):
    """Sets motor velocity to drives in a direction relative to its suroundings using an IMU sensor.

    Parameters:
        velocity: The velocity to drive in RPM.
        direction: The direction to drive the robot relative to its suroundings.
    """
    self.update_position(direction, velocity)
    direction -= self.imu.heading()
    current_direction = (self.encoder.position() - (360 * math.floor((self.encoder.position() - direction + 180) / 360)))
    if abs(direction - current_direction) < 20:
        direction -= self.drive_direction_controller.calculate(direction, current_direction)
    direction += 45
    self.front_left_motor_velocity = velocity * sin(direction)
    self.front_right_motor_velocity = velocity * cos(direction) * -1
    self.back_left_motor_velcity = velocity * cos(direction)
    self.back_right_motor_velocity = velocity * sin(direction) * -1
```

Figure 5. Arm movement.

```
@staticmethod
def raise_arm(target_height):
    """Moves the arm to one of three preset position

    Parameter:
        target_height: The position of the target fruit in pixels.
    """
    if target_height < 150:  # ≈100
        arm.move_to_position(3)
    if 150 < target_height < 220:  # ≈200
        arm.move_to_position(2)
    if 220 < target_height:  # ≈280
        arm.move_to_position(1)
```

Figure 6. Object tracking

```
def update_target_position(self, signatures):
    """Updates target_position based on a new picture"""
    def distance_from_past_target_location(element, target_position):
        """calculates the distance from one point [x, y] to another"""
        threshold = 20
        distance = threshold + 1
        if target_position:
            distance = (((target_position[0] - element.centerX) ** 2) + ((target_position[1] - element.centerY) ** 2)) ** 0.5
        return distance < threshold
    target_within_range = []
    fruits_detected = list(self.take_snapshots(signatures, 8))
    if fruits_detected:
        for fruit in fruits_detected:
            if distance_from_past_target_location(fruit, self.target_position):
                target_within_range.append(fruit)
    if target_within_range:
        self.target_position = [target_within_range[0].centerX, target_within_range[0].centerY]
        self.target_lost_frame_count = 0
    elif self.target_lost_frame_count > 10:
        self.target_position = None
    else:
        self.target_lost_frame_count += 1
```

Appendix D (Pictures)

Figure 1. Isometric view of Drivetrain

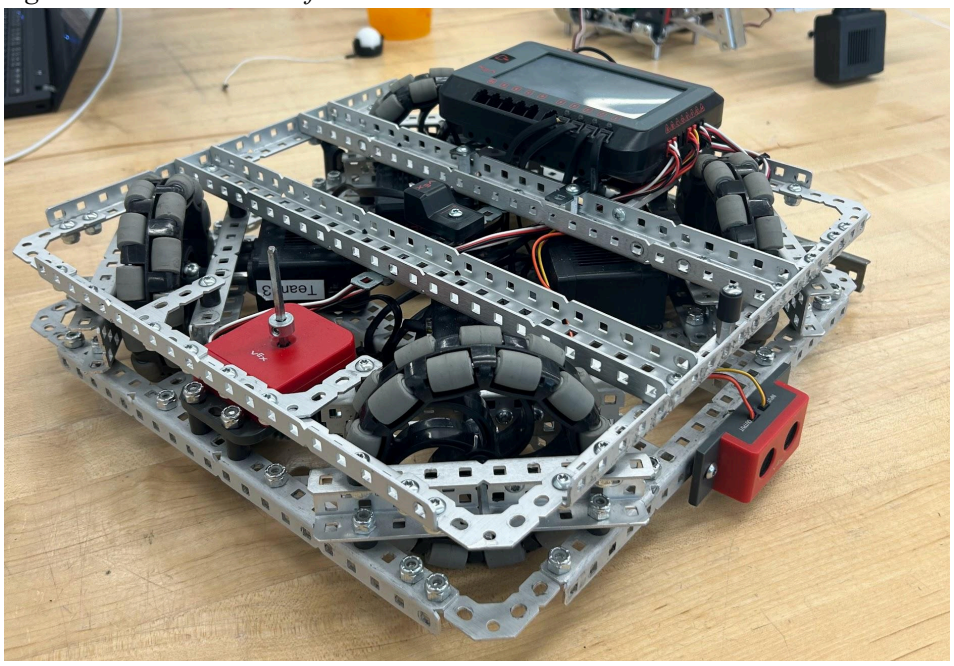


Figure 2. Isometric view of Drivetrain

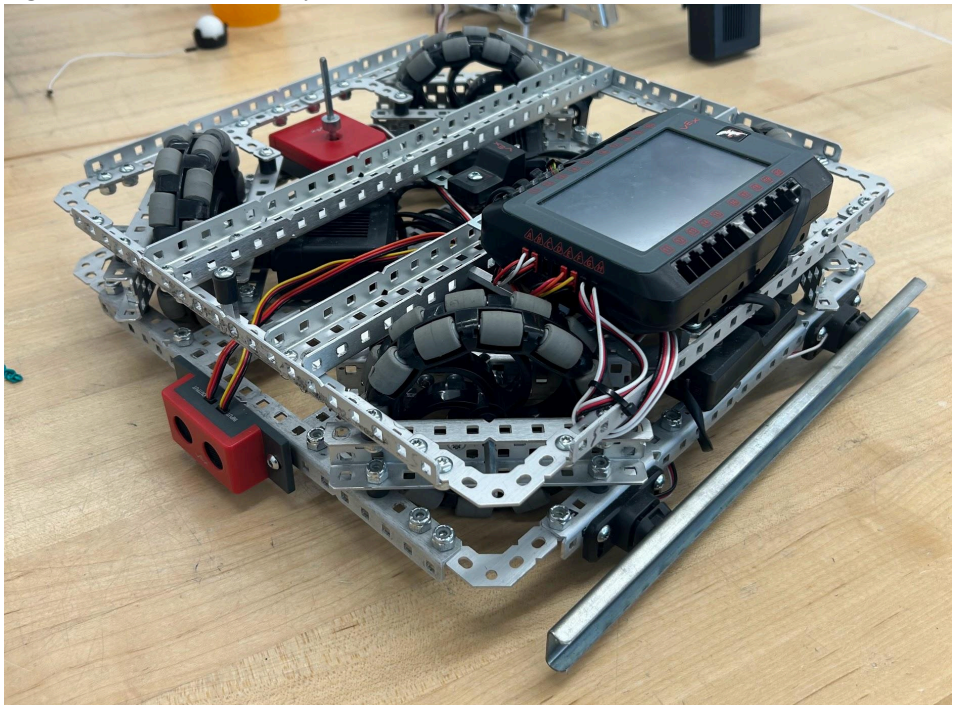


Figure 3: Top-Down View of Drivetrain

Figure 4. Drift Detector

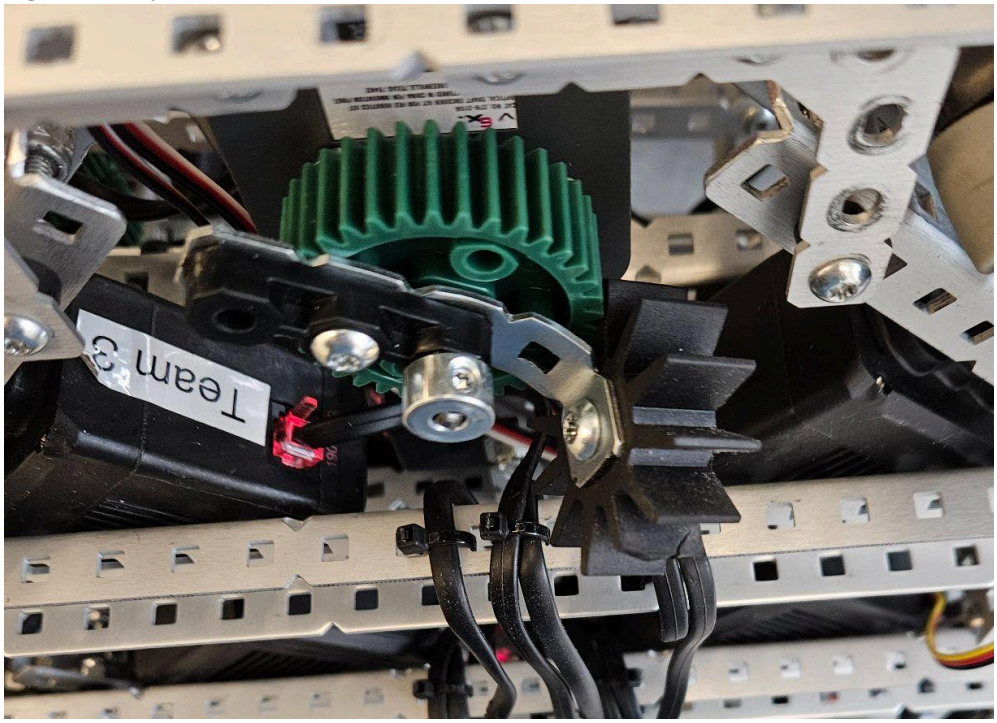


Figure 5. Isometric view of Double Reverse 4-Bar

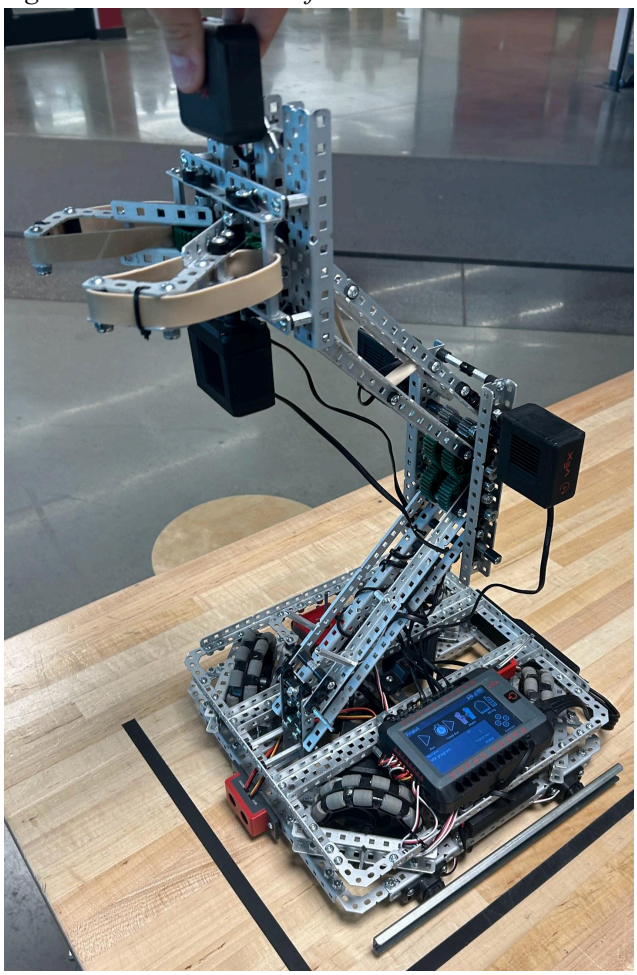


Figure 6. Isometric View of 4-Bar lift

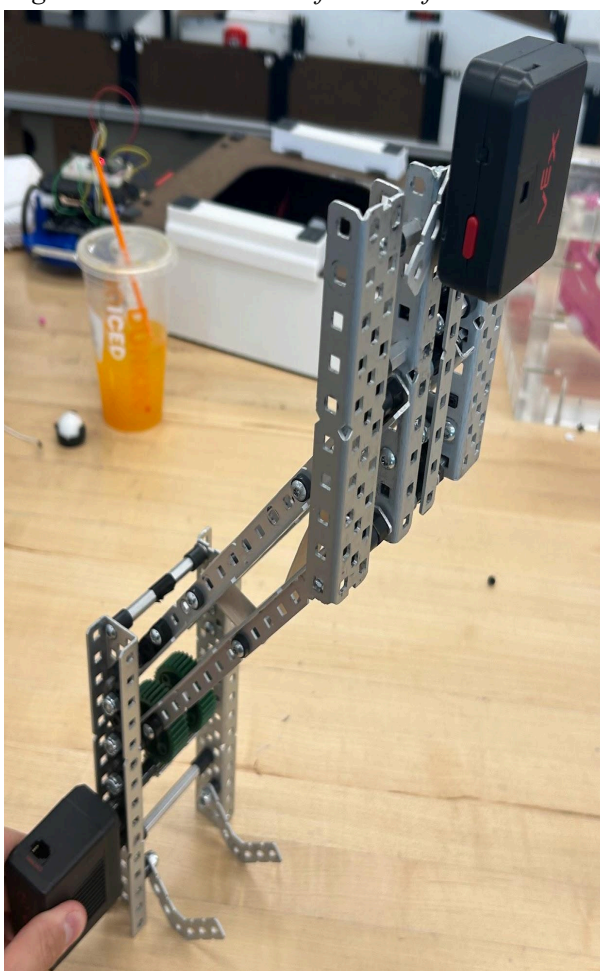


Figure 7. Non-parallel Jaw Gripper