Robotics Challenge- Final Report

Kanav Mehta, Harmish Zala, Hassan Shahzad

ABSTRACT

We created a tree-climbing robot using grippers, a linear actuator, and a rack and pinion. This report will talk about the control algorithms involved, software programming, and a mathematical analysis of the robot's performance.

Keywords: Control, Programming, Mathematical Evaluation

1 CONTROL SYSTEM

Higher Level Control Description

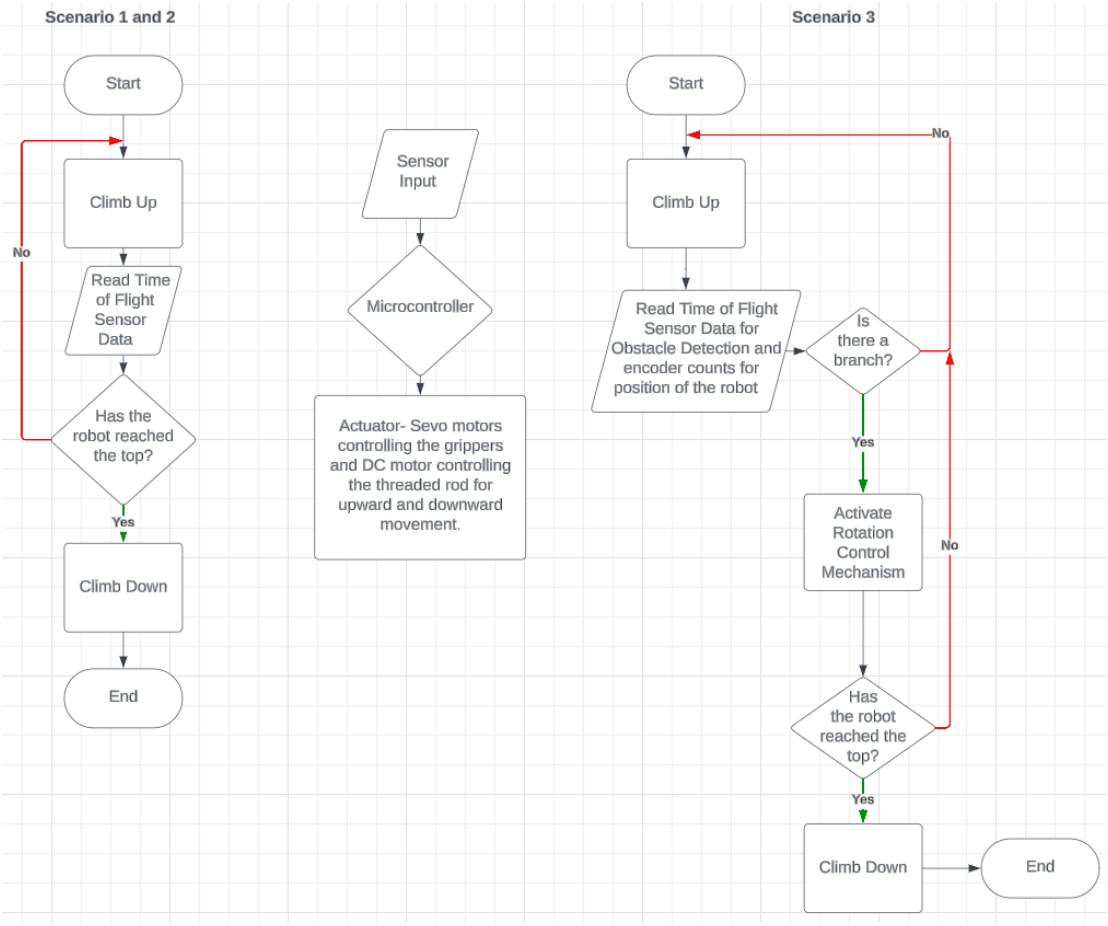

Figure 1. Simplistic Overview

Figure 1 provides a straightforward, easy-to-understand overview of the detailed flowcharts that will be presented later. It also illustrates how the symbols are associated with the control components of the robot, helping to clarify their roles and interactions.

For our tree-climbing robot, TreeRunner, we will primarily be using sensors like the time-of-flight that is used for detecting obstacles such as branches and determining the highest point of ascent and the

quadrature encoder within the DC motor which is crucial for managing the linear velocity of the grippers and facilitating the robot's descent. These sensors interact with the Seeeduino Xiao microcontroller, which compares threshold values to real-time data from the sensors. Based on this comparison, the microcontroller sends control signals to actuators to activate or deactivate the DC motor, adjusting the grippers' ascent and descent. Additionally, it sets the servo motors to specific angles to control the opening and closing of the grippers, as specified in the code.

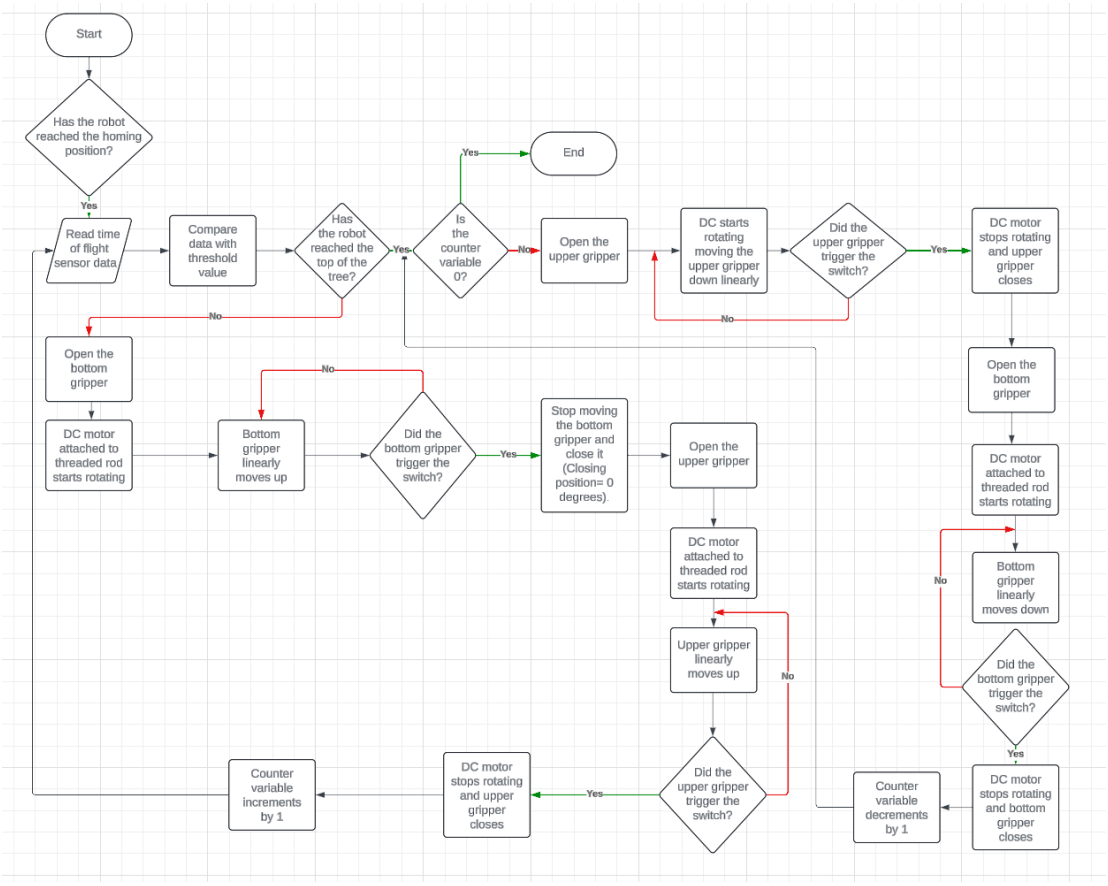

Figure 2. Flowchart for Scenario 1 and 2

As seen in Figure 2, for the ascent sequence, the bottom gripper is first opened. This is achieved through the communciation between the Seeeduino Xiao microcontroller and the servo motor, which is linked to one of the gripper's fingers via a gear, to rotate to a specific angle. Once the bottom gripper is open, the sequence of the code is followed by the DC motor being activated to rotate the threaded rod which drives the bottom gripper upward. This upward motion continues until the bottom gripper activates a switch. A while loop controls this process, allowing the threaded rod to continue rotating until the switch is either engaged or disengaged. Upon triggering the switch, the bottom gripper then closes, with the servo motor set to an angle of 0 degrees. It is essential during testing to verify using a servo tester that an angle of 0 degrees corresponds to the gripper being fully closed.

The process then repeats for the upper gripper, albeit with a slight modification. During testing, it was observed that fully opening the upper gripper causes the entire robotic mechanism to lean backward due to the center of mass. This leaning issue is particularly problematic during scenario 2 where the tree's diameter varies, as a fully opened upper gripper might fail to securely clasp the trunk. To address this, the upper gripper is set to open only to a specific angle—termed the "loosening angle" in the code—which prevents it from making excessive contact with the tree while also avoiding any backward lean. Following this adjustment in the code, the upper gripper opens to the loosening angle, and the threaded rod is activated, driving the upper gripper upwards. This continues until the bottom switch is activated, which halts the DC motor that rotates the threaded rod, allowing the upper gripper to then close securely.

For climbing up during scenarios 1 and 2, each complete sequence—comprising opening the bottom gripper, rotating the threaded rod to elevate the bottom gripper, closing the bottom gripper, opening the upper gripper, and rotating the threaded rod again to elevate the upper gripper—is defined as one cycle. A counter variable increments by one after each cycle is completed. This tracking is crucial for the descent phase, where the process is mirrored in reverse. Knowing the number of cycles executed during the climb is essential for designing an autonomous system that accurately repeats these cycles in reverse when descending. Hence, a complete cycle when moving down decrements the counter variable by 1. This process repeats until the counter variable is 0. The use of counter variables helps create an autonomous system which can be used in any tree of varying height.

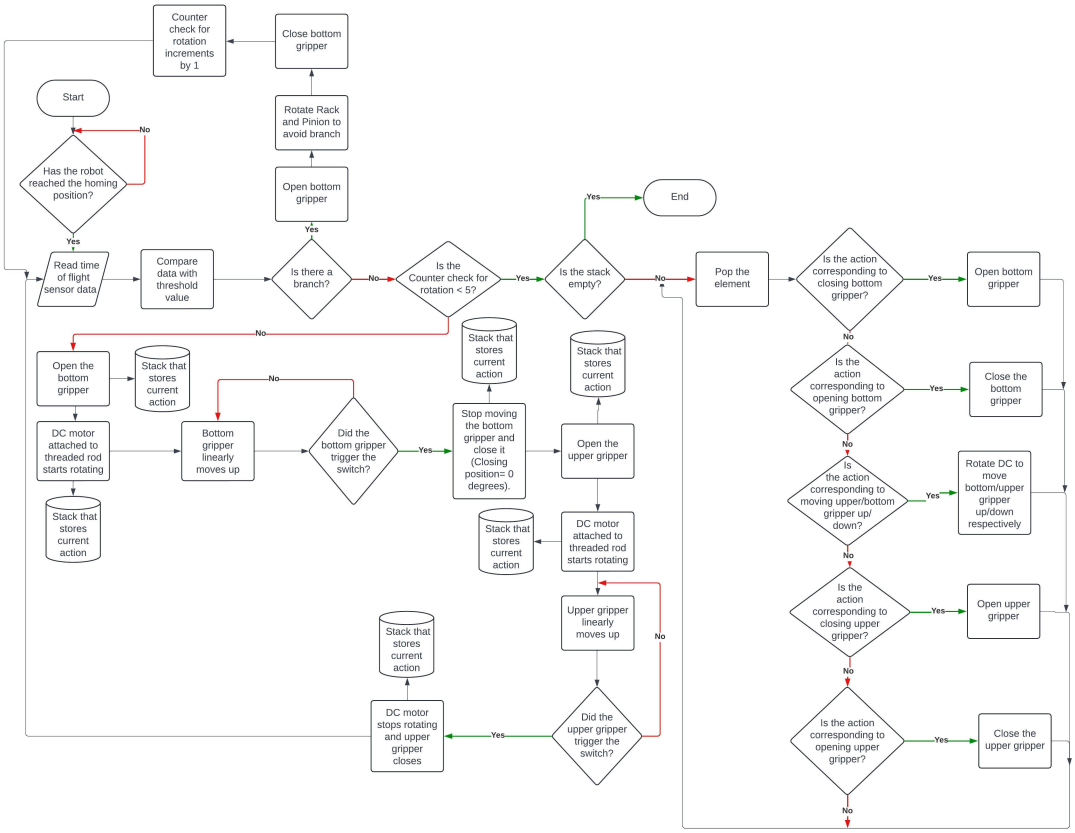

Figure 3. Flowchart for Scenario 3

Figure 3 illustrates the control logic for scenario 3. The ascent sequence remains unchanged except for the use of counter-variables. In scenario 3 of the tree climbing robot's operation, branches pose significant obstacles that must be strategically avoided using a Time-of-Flight (ToF) sensor during the ascent. The Time-of-Flight sensor is crucial in determining the distance of the robot from incoming branches. By continuously measuring the distance to obstacles, the ToF sensor provides real-time data that enables TreeRunner to execute branch avoidance maneuvers effectively. When the sensor detects an obstacle within a predefined threshold distance, the control system initiates a sequence to open the gripper, rotate the rack and pinion mechanism, and re-securing the gripper in a new position to navigate past the obstacles. This process is logged in detail, recording the specific encoder positions at which each avoidance maneuver is executed. Furthermore, an issue arises with the turning maneuver. Due to the presence of branches, the robot needs to determine whether it has reached the top. Testing revealed that after at least four turns of the rack and pinion, the robot should no longer have the branch in its line of sight. If the robot continues to detect an obstacle within that proximity, it is likely that it has reached the top, where a platform is obstructing the path.

In order to fully avoid the branch, the robot must ideally be in a position where it is on the opposite side of the branch. In order to acheive this we have an algorithm that runs every time the the robot climbs

up in which the rotating mechanism sweeps side to side to detect branches. Every degree the rotator moves, it records the furthest distance the ToF sensor can read and once within a threshold, the robot stops and moves 180 degrees- the minimum value the ToF sensor captures. We construct a 2-dimensional array of data where each element stores the angle of the rotating mechanism along with the recorded angle from the ToF sensor. We then perform a minimum value-finding algorithm that searches through the entire list and is wiped every time the rotator begins its cycle again.

The real challenge, however, arises during the descent, where the robot must navigate back down the tree without the direct visual feedback on branch locations provided during the ascent (as only 1 time of flight sensor is being used). In the operational cycle of our tree-climbing robot, each major action—opening the bottom gripper, elevating the bottom gripper via the threaded rod, closing the bottom gripper, opening the upper gripper, lifting the upper gripper, and finally closing the upper gripper—is tracked using dedicated variables. As seen in the flowchart (Figure 2), these values are pushed to a stack each time their respective function is called. Upon reaching the top, the stack will contain distinct values corresponding to the sequence of actions executed from the bottom to the top. The descent strategy then involves popping the elements of the stack and reversing these actions in the exact reverse order. As the robot descends, a series of switch case statements will determine the appropriate action to perform. For example, if a number in the stack indicates opening the bottom gripper, the corresponding reverse action will be to close the bottom gripper. Moreover, movements that involve moving the upper/bottom gripper up are reversed by moving the respective grippers down. The aim is for the stack to be empty. If the stack is empty, the robot would have reached the bottom. This ensures every action taken on the way up is precisely undone on the way down. This systematic approach helps in carefully retracing the robot's path, maintaining stability and accuracy throughout the descent.

This method could also be applied to the rotation mechanism. However, our initial plan involved using the encoder in the DC motor. Essentially, the robot would use the recorded encoder positions from its ascent to accurately replicate the avoidance maneuvers in reverse order during descent. For each recorded maneuver point, the descent logic calculates the corresponding position from the top of the tree (subtracting the ascent maneuver position from the total tree height). As the robot descends and reaches these calculated positions, it performs the reverse maneuvers—rotating back to the original path and closing the gripper, effectively retracing its steps. This method ensured that the robot can descend safely by accurately mirroring the ascent path, avoiding branches without needing to detect them a second time. However, during testing, we discovered that the encoder values were not as accurate as expected, leading us to opt for the stack method instead. Additionally, using the encoder values would not provide an autonomous solution, but rather a hard-coded one, which did not align with the task's objectives. Therefore, we chose the stack method.

Sensor Deposition

For the sensor deposition, we created a 3D support for the block with a magnet at the center. The support had holes for strings that were wrapped around the block to hold it in place. The surface magnet was intentionally weak to ensure the block detached when it made contact with the magnet at the top of the tree. This platform was mounted at the head of the robot, positioned on the same plane as the time-of-flight sensor. It neither affected nor was affected by the opening and closing of the grippers.

Low Level Algorithm

We have implemented two actuators in our design: a servo motor and a DC motor. Initially, we employed a PID algorithm to control the linear velocity of the grippers during their ascent and descent, based on a formula we derived for linear velocity while designing the robot's kinematics. The reason we chose servo motors is due to their built-in closed-loop system that uses feedback to precisely reach and maintain a desired angle. This internal feedback mechanism involves a potentiometer and a control circuit, allowing for continuous adjustments to ensure accurate positioning based on the input signal.

The inputs for the PID tuning include angular velocity data from the built-in quadrature encoder, as well as pitch, diameter, RPM, and screw length, to accurately control the target velocity of the DC motor that powers the linear actuator. The intended output control signal was a PWM signal to the DC motor, adjusting its speed up or down based on error calculations performed by the microcontroller. However, testing revealed inaccuracies in the angular velocity measurements provided by the quadrature encoder. As a result, we decided to simplify the control approach by merely turning the motor on and off, rather than modulating PWM signals.

Special Considerations

Two switches have been installed for the respective grippers. These switches are crucial as they signal the code to stop rotating the threaded rod in the spine that moves the grippers vertically. Additionally, the time-of-flight sensor, with a 27-degree field of view, aids in detecting branches even after the robot has rotated. This capability significantly enhances the robot's ability to avoid collisions with branches. The rack and pinion system has its limitations, as it does not completely encircle the bark, restricting the robot's rotation. If a branch is detected post-rotation, the code will command a further rotation to completely evade the branch.

Strategies for Enhancing Code Efficiency

We decided to modularize the operations of opening and closing the gripper, rotating the rack and pinion, and rotating the threaded rod into separate functions within the Arduino sketch. This approach enhances the readability and efficiency of the code. Each function accepts the servo number as an input parameter, and a switch-case statement within the function determines the specific servo motor to actuate. The servo angles were predetermined during a rigorous testing and trial-and-error process, ensuring optimal performance for each operation. These functions are then invoked within the void loop() to execute the climbing mechanism's sequence effectively.

The inclusion of delay commands within the code is strategic. It aims to ensure that the movements of different servos are well-coordinated and not simultaneous. This is crucial for preventing abrupt transitions and interactions between the servos. By introducing these delays, we ensure that each part of the mechanism completes its action fully before the next begins, facilitating a smooth and controlled operation of the entire system. This approach helps avoid mechanical stress and potential errors that may arise if multiple servos were to operate simultaneously without proper sequencing.

2 ROBOT PROGRAMMING

Source code : `https://github.com/HarmishZala/Robotics_challenge_Treerunner.git`

Functions and its descriptions

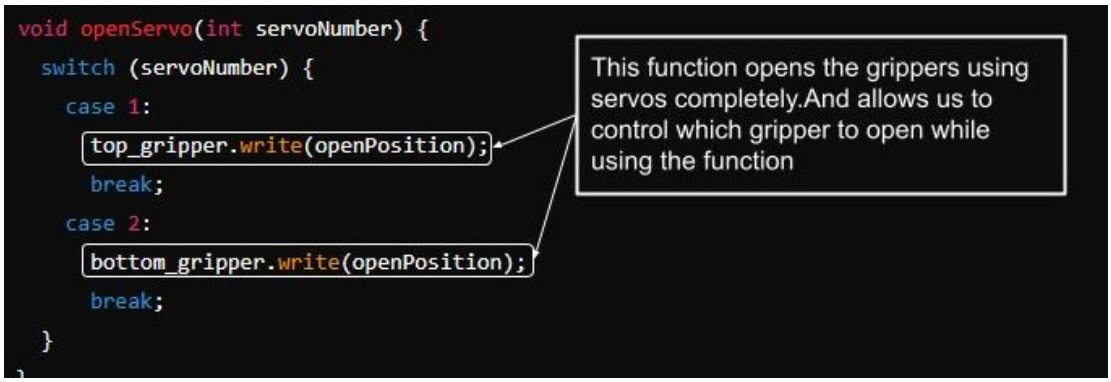

Figure 4. Open servo

This code function is the first iteration. After testing it, we realized that opening the servos fully caused the robot to fall behind (away from the trunk), as the robot's center of mass was not close to or towards the trunk. While creating this function, we tried to be effective and smart as through the switch case, we can control which gripper to open when calling the function instead of creating separate functions for them As you will see in the next function called 'loosenServos' we can tackle this issue.

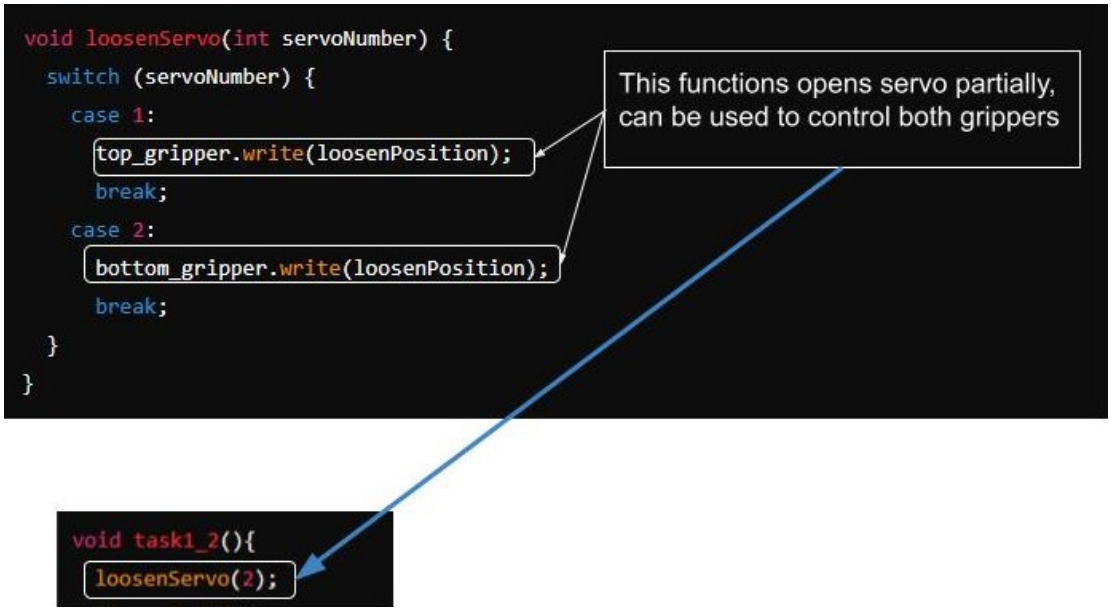

Figure 5. Loosen servo

As the center of mass of the motor is further away from the trunk of the tree, opening the grippers fully leads the robot to fall behind (away from the trunk). Hence, to tackle that problem, we are partially opening the servos so that even if it falls behind, the robot can still hold onto the trunk and maintain its grip. This is an improved version of the 'openServos' functions. Using the switch and case of C++ allows us to specify which grippers to loosen, which is better than making separate functions for each gripper

```
void closeServo(int servoNumber) {
  switch (servoNumber) {
    case 1:
      top_gripper.write(closePosition);
      break;
    case 2:
      bottom_gripper.write(closePosition);
      break;
  }
}
```

Figure 6. Close servo

This function is the opposite of 'openServos' as it closes the grippers to 0 degrees and ensures that the arms are tightly gripped around the trunk. Again, using the switch case allows us to control which gripper to close when calling it. This will be useful as it will make the final algorithm easier and more readable.

```
void linearDown() {
  // Set motor direction to down
  digitalWrite(IN1, HIGH);
  digitalWrite(IN2, LOW);
}
```

```
void linearUp() {
  // Set motor direction to up
  digitalWrite(IN1, LOW);
  digitalWrite(IN2, HIGH);
}
```

Figure 7. linear up and Linear down

This function sets the motor to move in the ”up” direction. It sets the IN1 pin to LOW and the IN2 pin to HIGH. These pins control the motor driver that manages the motor’s direction. Setting IN1 to LOW and IN2 to HIGH will cause the motor to rotate in one direction; in this case, it will rotate, causing the bottom arm to move up. The same goes with linear down but in the opposite direction.

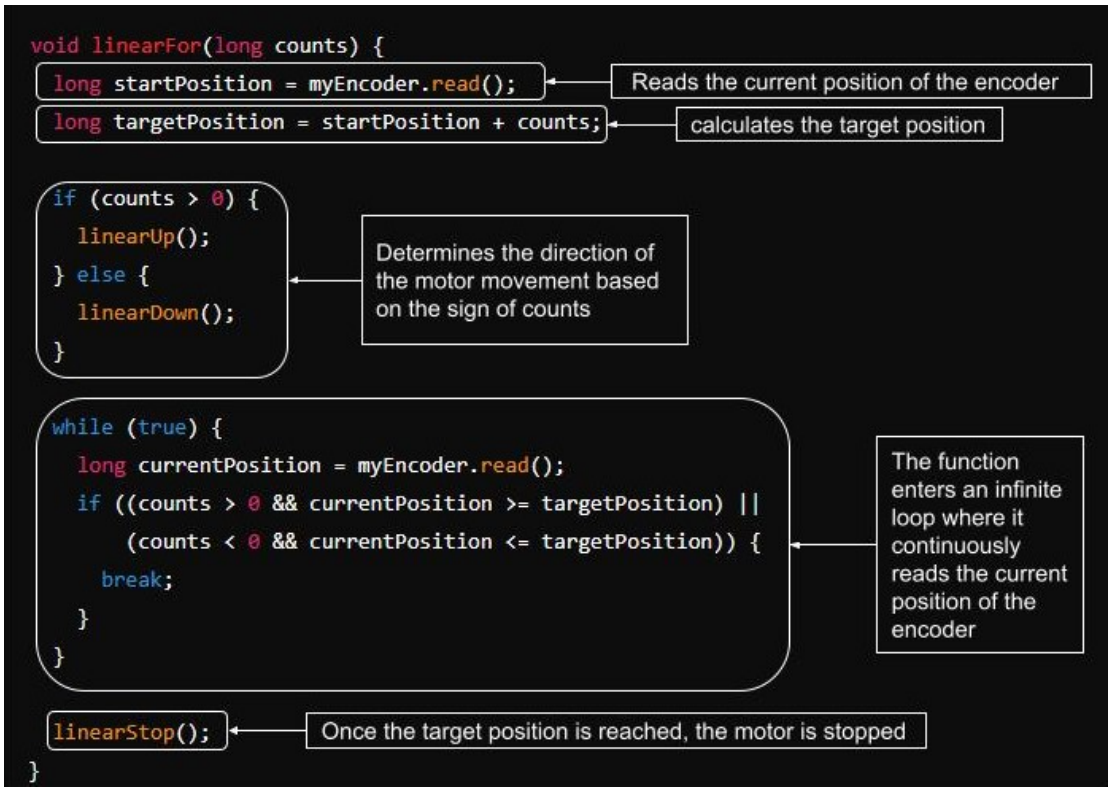

Figure 8. linearFor

- The function starts by reading the encoder’s current position, which provides the motor’s initial position. - It then calculates the target position by adding the desired number of encoder counts (counts) to the initial position (startPosition).

- The function determines the direction of the motor movement based on the sign of counts:

- If the count is positive, the motor moves up by calling linear (). - If the count is negative, the motor moves down by calling ’linearDown().

- The function enters an infinite loop where it continuously reads the current position of the encoder.

- It checks if the motor has reached or surpassed the target position:

- If moving up (counts ¿ 0), the loop breaks when the current position is greater than or equal to the target position.

- If moving down (counts ¡ 0), the loop breaks when the current position is less than or equal to the target position.

- The linearFor function is used in the main tasks (’task12’ in this case) to move the motor by a

specified amount, ensuring precise control over motor movements based on encoder feedback. This is critical for tasks requiring accurate positioning, such as robotic arms or linear actuators.

- The function provides a straightforward and effective way to move a motor to a precise position. Using encoder feedback ensures accurate and repeatable movements, which is essential for tasks requiring high precision. The ability to move both up and down based on the counts provided makes it versatile for various applications in robotics and automation.

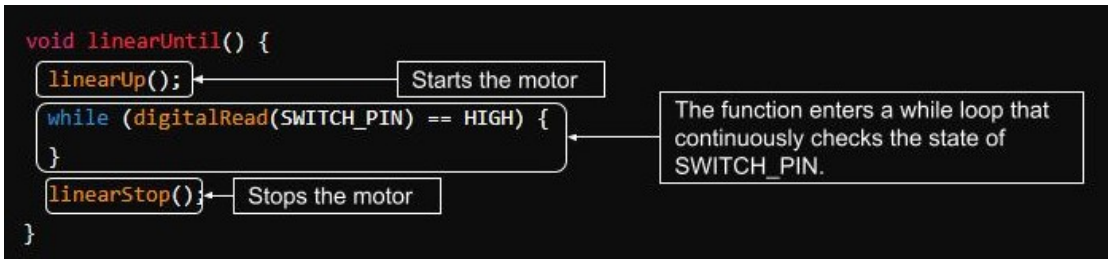

Figure 9. Linear Until

The function begins by calling ‘linearUp()‘, which sets the motor control pins to move the motor in the ”up” direction.

The function enters a while loop that continuously checks the state of ‘SWITCH_PIN‘.

digitalRead(SWITCH_PIN)‘:

- Reads the state of the switch connected to ‘SWITCH_PIN‘.

- If the switch is not pressed, ‘digitalRead(SWITCH_PIN)‘ returns ‘HIGH‘, and the motor keeps moving.

- If the switch is pressed, ‘digitalRead(SWITCH_PIN)‘ returns ‘LOW‘, causing the loop to exit.

Once the switch is pressed (‘digitalRead(SWITCH_PIN) == LOW‘), the loop exits, and ‘linearStop()‘ is called to stop the motor.

The primary purpose of ‘linearUntil‘ is to move the motor upwards until a physical limit (represented by a switch) is reached.

This is useful in scenarios where you need to move a component to a predefined position reliably and stop it precisely when it reaches that position. For instance, in robotic systems or automated machinery, stopping the motor at a specific point is crucial for accuracy and safety.

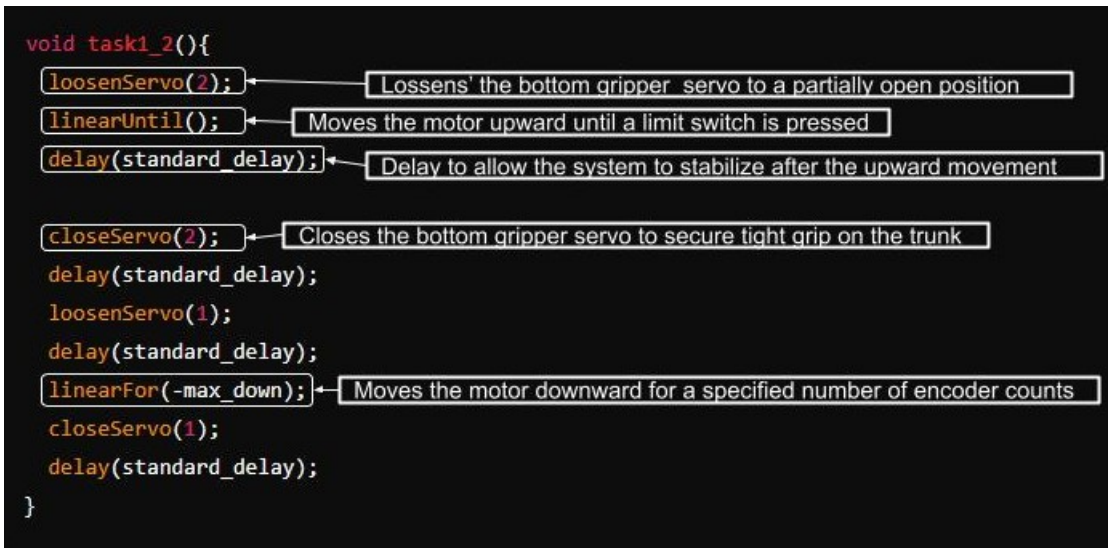

Figure 10. Task1 and 2

- The objective of this function is to perform a series of coordinated movements involving a motor and two servos. This sequence likely corresponds to a specific task such as picking up an object, moving

it, and then releasing it in a different position. - This function can be part of a larger automation or robotic system where such sequences are necessary for tasks like material handling, assembly, or sorting. - The function uses loosenServo and closeServo to control the positions of the top and bottom grippers. - It uses linearUntil to move the motor up until a limit switch is triggered and linearFor to move the motor down for a specific distance based on encoder counts. - Introduces delays (delay(standard_delay)) between movements to ensure stability and allow time for each action to complete.

- The task1_2 function contains a series of precise movements and actions involving motor and servo controls. It is designed to handle tasks that require coordinated movements, such as gripping, lifting, moving, and placing objects in a robotic system. The use of delays ensures stability and accuracy, making the function reliable for automated operations.

```
int getTopToFDistance() {
  VL53L0X_RangingMeasurementData_t measure;
  loxTop.rangingTest(&measure, false);
  if (measure.RangeStatus != 4) {
    return measure.RangeMilliMeter;
  } else {
    return 1000;
  }
}
```

Figure 11. Time of Flight sensor reading

The ‘getTopToFDistance’ function attempts to measure the distance using the top VL53L0X sensor. It returns the measured distance in millimeters if the measurement is valid. If there is an error or the measurement is out of range, it returns a default value of 1000 millimeters.

```
void rackPinionForward() {
  for (int i = 0; i < rackPinionIterations; i++) {
    rackPinionServo.write(rackPinionServo.read() + rackPinionAngle);
    delay(100);
  }
}
```

Figure 12. Curved rack and pinion forward

The function is defined to move the rack and pinion mechanism forward.The loop runs ’**rackPinionIterations**’ times, which is a constant value set elsewhere in the code.On each iteration, it reads the current position of the ’**rackPinionServo**’ using ’**rackPinionServo.read**()’.It then writes a new position to the servo by adding ’**rackPinionAngle**’ to the current position, effectively incrementing the servo angle step by step.A ’**delay(100)**’ is added between iterations to ensure the servo has time to move to the new position before the next increment.

This function moves the rack and pinion mechanism forward in small increments.This is essential for precise control over the robot’s movement around the tree.The delay(100) ensures that the movement is not too fast, allowing for smoother operation and better control.This function is used in scenarios where the robot needs to move the rack and pinion mechanism forward, likely during the process of climbing or adjusting its position around the tree to avoid branches.

```
void rackPinionBackward() {
  for (int i = 0; i < rackPinionIterations; i++) {
    rackPinionServo.write(rackPinionServo.read() - rackPinionAngle);
    delay(100);
```

Figure 13. Curved rack and pinion backward

The function is defined to move the rack and pinion mechanism backward.Similar to the '**rackPinionForward**' function, the loop runs **'rackPinionIterations'** times.On each iteration, it reads the current position of the '**rackPinionServo**' using '**rackPinionServo.read()**'.It then writes a new position to the servo by subtracting '**rackPinionAngle**' from the current position, effectively decrementing the servo angle step by step.A delay(100) is added between iterations to ensure the servo has time to move to the new position before the next decrement. This function moves the rack and pinion mechanism backward in small increments.This allows the robot to reverse its movement around the tree with precision.The delay(100) ensures that the movement is not too fast, allowing for smoother operation and better control.This function is used in scenarios where the robot needs to move the rack and pinion mechanism backward, likely during the process of descending or adjusting its position around the tree. Both functions provide precise control over the rack and pinion mechanism by moving it in small increments.These functions allow the robot to adapt its position around the tree, facilitating both climbing and descending operations. These functions are crucial for the robot's ability to navigate around the tree. By enabling controlled forward and backward movements of the rack and pinion mechanism, they help the robot adjust its grip and position, ensuring stable and efficient climbing or descending actions along with avoiding branches.

```
void performAction(int action, int parameter = 0) {
  actions[actionIndex++] = action;
  switch (action) {
    case OPEN_BOTTOM_GRIPPER:
      openServo(3);
      break;
    case CLOSE_BOTTOM_GRIPPER:
      closeServo(3);
      break;
    case MOVE_UP_UNTIL_SWITCH:
      moveUpUntilSwitchClick();
      break;
    case MOVE_DOWN_BY_COUNTS:
      moveDownByCounts(parameter);
      break;
    case MOVE_UP_BY_COUNTS:
      moveUpByCounts(parameter);
      break;
    case OPEN_TOP_GRIPPER:
      openServo(1);
      break;
    case CLOSE_TOP_GRIPPER:
      closeServo(1);
      break;
    case RACK_PINION_FORWARD:
      rackPinionForward();
      break;
    case RACK_PINION_BACKWARD:
      rackPinionBackward();
      break;
```

Figure 14. Perform action

The function is a redesign of some functions from code for tasks 1 and 2. In the line 'actions[actionIndex++] = action;', the current action is stored in the actions array, and the actionIndex is incremented. This keeps a record of all actions performed, which is useful for reversing actions if needed (e.g., while descending). The switch statement executes different functions based on the value of the action parameter. Each case corresponds to a specific action that the robot can perform.

Cases and Functions: `actions[actionIndex++] = action`: The current action is stored in the `actions` array, and the `actionIndex` is incremented. This keeps a record of all actions performed, useful for reversing actions if needed (e.g., while descending).

The `switch` statement executes different functions based on the value of the `action` parameter. Each case corresponds to a specific action the robot can perform.

Cases and Functions:

- `OPEN_BOTTOM_GRIPPER`: Calls `openServo(3)` to open the bottom gripper.
- `CLOSE_BOTTOM_GRIPPER`: Calls `closeServo(3)` to close the bottom gripper.

- `MOVE_UP_UNTIL_SWITCH`: Calls `moveUpUntilSwitchClick()` to move up until a switch is activated.

- `MOVE_DOWN_BY_COUNTS`: Calls `moveDownByCounts(parameter)` to move down by a specified number of counts.

- `MOVE_UP_BY_COUNTS`: Calls `moveUpByCounts(parameter)` to move up by a specified number of counts.

- `OPEN_TOP_GRIPPER`: Calls `openServo(1)` to open the top gripper.

- `CLOSE_TOP_GRIPPER`: Calls `closeServo(1)` to close the top gripper.

- `RACK_PINION_FORWARD`: Calls `rackPinionForward()` to move the rack and pinion mechanism forward.

- `RACK_PINION_BACKWARD`: Calls `rackPinionBackward()` to move the rack and pinion mechanism backward.

`int parameter = 0`: This default parameter is used in cases where additional information (like the number of counts to move) is required. For actions that do not need an extra parameter, the default value is 0.

This function centralizes the execution of all robot actions, making the code more modular and easier to manage. By abstracting actions into a single function, it simplifies the control logic.

By storing each action in the `actions` array, the robot can reverse its steps, which is crucial for tasks like climbing down the tree.

`performAction` is used in functions like `task3_up` and `task3_down` to execute various actions required for climbing up or down the tree. For instance, it is called to open or close grippers, move the robot up or down, and control the rack and pinion mechanism.

The `performAction` function is a key component of the tree-climbing robot's control system. It serves to:

- **Modularize Action Execution**: By abstracting individual actions into a single function, the code becomes more organized and easier to maintain.

- **Enable Action Logging**: Storing actions in an array allows for easy reversibility, which is essential for safely descending the tree.

- **Streamline Task Functions**: It simplifies task functions by providing a single interface to perform any action, reducing repetitive code and potential errors.

Overall, this function is crucial for the robot's ability to perform complex sequences of actions reliably and efficiently, enabling it to navigate and climb trees with precision and control.

The performAction function is a key component of the tree-climbing robot's control system. It serves to: Modularize Action Execution: By abstracting individual actions into a single function, the code becomes more organized and easier to maintain. Enable Action Logging: Storing actions in an array allows for easy reversibility, which is essential for safely descending the tree. Streamline Task Functions: It simplifies task functions by providing a single interface to perform any action, reducing repetitive code and potential errors. Overall, this function is crucial for the robot's ability to perform complex sequences of actions reliably and efficiently, enabling it to navigate and climb trees with precision and control.

```
void task3_up() {
  while (true) {
    performAction(OPEN_BOTTOM_GRIPPER);
    performAction(MOVE_UP_UNTIL_SWITCH);
    performAction(MOVE_DOWN_BY_COUNTS, 100);
    performAction(CLOSE_BOTTOM_GRIPPER);

    performAction(MOVE_UP_BY_COUNTS, 200);

    while (true) {
      int distance = getTopToFDistance();
      Serial.print("Top Distance (mm): ");
      Serial.println(distance);

      if (distance < 400) {
        performAction(MOVE_DOWN_BY_COUNTS, 100);
        performAction(CLOSE_TOP_GRIPPER);
        performAction(OPEN_BOTTOM_GRIPPER);
        performAction(RACK_PINION_FORWARD);
        performAction(CLOSE_BOTTOM_GRIPPER);
        performAction(OPEN_TOP_GRIPPER);
        performAction(RACK_PINION_BACKWARD);
        performAction(CLOSE_TOP_GRIPPER);
      } else {
        break;
      }
    }

    performAction(OPEN_TOP_GRIPPER);
    performAction(MOVE_UP_BY_COUNTS, 200);
    performAction(CLOSE_TOP_GRIPPER);
    performAction(OPEN_BOTTOM_GRIPPER);
    performAction(MOVE_UP_BY_COUNTS, 200);
  }
}
```

Figure 15. Task 3 up

The outer `while (true)` loop makes the function run indefinitely. This loop is designed to continually execute the sequence of actions required for the robot to climb up the tree.

- `performAction(OPEN_BOTTOM_GRIPPER)`: Opens the bottom gripper to release its hold on the tree.
- `performAction(MOVE_UP_UNTIL_SWITCH)`: Moves the robot upwards until a switch is

activated, indicating it has reached a specific position.

- `performAction(MOVE_DOWN_BY_COUNTS, 100)`: Moves the robot down by 100 encoder counts to ensure it is in the correct position for gripping again.
- `performAction(CLOSE_BOTTOM_GRIPPER)`: Closes the bottom gripper to secure the robot's position on the tree.
- `performAction(MOVE_UP_BY_COUNTS, 200)`: Moves the robot up by 200 encoder counts.

Distance Check Loop:
This inner `while (true)` loop continuously checks the distance from the top ToF sensor.

- `int distance = getTopToFDistance()`: Measures the distance from the top ToF sensor.
- `Serial.print("Top Distance (mm): "); Serial.println(distance);`: Prints the measured distance to the serial monitor for debugging purposes.
- `if (distance < 400)`: If the measured distance is less than 400 mm, it indicates the robot is close to an obstacle or the end of the current climbing segment.

Actions for Close Distance:

- `performAction(MOVE_DOWN_BY_COUNTS, 100)`: Moves the robot down by 100 counts to adjust its position.
- `performAction(CLOSE_TOP_GRIPPER)`: Closes the top gripper.
- `performAction(OPEN_BOTTOM_GRIPPER)`: Opens the bottom gripper.
- `performAction(RACK_PINION_FORWARD)`: Moves the rack and pinion mechanism forward to adjust the robot's position around the tree.
- `performAction(CLOSE_BOTTOM_GRIPPER)`: Closes the bottom gripper to secure the position.
- `performAction(OPEN_TOP_GRIPPER)`: Opens the top gripper.
- `performAction(RACK_PINION_BACKWARD)`: Moves the rack and pinion mechanism backward to complete the adjustment.
- `performAction(CLOSE_TOP_GRIPPER)`: Closes the top gripper to secure the new position.
- `else { break; }`: If the distance is 400 mm or more, breaks out of the inner `while (true)` loop, indicating that the robot can continue its upward movement.

After exiting the inner loop, the function continues the climbing process:

- `performAction(OPEN_TOP_GRIPPER)`: Opens the top gripper.
- `performAction(MOVE_UP_BY_COUNTS, 200)`: Moves the robot up by 200 counts.
- `performAction(CLOSE_TOP_GRIPPER)`: Closes the top gripper to secure the position.
- `performAction(OPEN_BOTTOM_GRIPPER)`: Opens the bottom gripper.
- `performAction(MOVE_UP_BY_COUNTS, 200)`: Moves the robot up by another 200 counts.

This function is designed to enable the robot to climb up a tree continuously. The inner loop and distance checks ensure the robot adjusts its position when necessary to avoid obstacles and maintain a stable climb.

- The `task3_up` function is integral to the robot's ability to climb trees. It combines several actions, controlled by the `performAction` function, to open and close grippers, move up or down by specific counts, and adjust the robot's position based on sensor feedback. This continuous loop ensures the robot climbs efficiently and adjusts its position as needed to avoid obstacles and maintain stability.
- By utilizing sensor feedback and precise control of the grippers and movement mechanisms, this function allows the robot to autonomously climb trees, making it useful for applications such as inspection and maintenance. The modular approach also allows for easy adjustments and improvements to the climbing logic, enhancing the robot's versatility and reliability.

```
void task3_down() {
  for (int i = actionIndex - 1; i >= 0; i--) {
    int action = actions[i];
    switch (action) {
      case OPEN_BOTTOM_GRIPPER:
        closeServo(3);
        break;
      case CLOSE_BOTTOM_GRIPPER:
        openServo(3);
        break;
      case MOVE_UP_UNTIL_SWITCH:
        moveDownByCounts(200);
        break;
      case MOVE_DOWN_BY_COUNTS:
        moveUpByCounts(100);
        break;
      case MOVE_UP_BY_COUNTS:
        moveDownByCounts(200);
        break;
      case OPEN_TOP_GRIPPER:
        closeServo(1);
        break;
      case CLOSE_TOP_GRIPPER:
        openServo(1);
        break;
      case RACK_PINION_FORWARD:
        rackPinionBackward();
        break;
      case RACK_PINION_BACKWARD:
        rackPinionForward();
        break;
    }
  }
```

Figure 16. Task 3 down

The `for` loop iterates through the `actions` array in reverse order, starting from the last action performed (`actionIndex - 1`) down to the first action (`i >= 0`). Reversing the order of actions allows the robot to undo each action in the correct sequence, effectively retracing its steps to climb down the tree.

The `switch` statement handles reversing each specific action stored in the `actions` array.

- `OPEN_BOTTOM_GRIPPER`:
  - Original action: Opens the bottom gripper.
  - Reverse action: `closeServo(3)` closes the bottom gripper.
- `CLOSE_BOTTOM_GRIPPER`:
  - Original action: Closes the bottom gripper.
  - Reverse action: `openServo(3)` opens the bottom gripper.
- `MOVE_UP_UNTIL_SWITCH`:
  - Original action: Moves up until a switch is activated.
  - Reverse action: `moveDownByCounts(200)` moves the robot down by a predefined number of counts.
- `MOVE_DOWN_BY_COUNTS`:
  - Original action: Moves down by a specific number of counts.
  - Reverse action: `moveUpByCounts(100)` moves the robot up by the same number of counts.
- `MOVE_UP_BY_COUNTS`:
  - Original action: Moves up by a specific number of counts.
  - Reverse action: `moveDownByCounts(200)` moves the robot down by the same number of counts.
- `OPEN_TOP_GRIPPER`:
  - Original action: Opens the top gripper.
  - Reverse action: `closeServo(1)` closes the top gripper.
- `CLOSE_TOP_GRIPPER`:
  - Original action: Closes the top gripper.
  - Reverse action: `openServo(1)` opens the top gripper.
- `RACK_PINION_FORWARD`:
  - Original action: Moves the rack and pinion mechanism forward.
  - Reverse action: `rackPinionBackward()` moves the rack and pinion mechanism backward.
- `RACK_PINION_BACKWARD`:
  - Original action: Moves the rack and pinion mechanism backward.
  - Reverse action: `rackPinionForward()` moves the rack and pinion mechanism forward.

This function is designed to reverse all actions the robot performed while climbing up, enabling it to safely and systematically climb down the tree. By reversing each action, the robot ensures that it maintains stability and control, preventing falls or missteps during descent.
The `task3_down` function plays a crucial role in the tree-climbing robot's operation. By methodically reversing each action the robot took to climb up, it ensures a safe and controlled descent. This function relies heavily on the `performAction` function and the action logging system to accurately retrace the robot's steps.
The benefits of this approach include:

- The robot can safely climb down by undoing actions in the correct sequence.
- The use of action logging and reversal makes the control logic clear and maintainable.
- Ensuring that each action is properly reversed helps maintain the robot's stability and grip on the tree, preventing accidents.

Overall, the `task3_down` function is essential for the robot's ability to navigate back down a tree after climbing, making it a versatile and reliable tool for tasks such as inspection, maintenance, and research in arboreal environments.

3 CODE ALGORITHM FLOW CHARTS

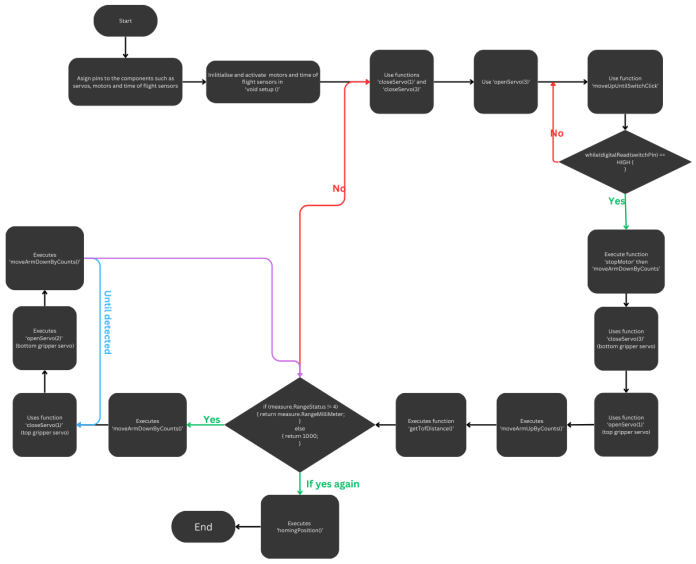

Figure 17. Task 1 code Flow chart

In the flowchart for Task 1, the process begins with assigning pins to various components such as servos, motors, and Time of Flight (ToF) sensors. This initial step involves defining the pin numbers for different inputs and outputs used in the program. For instance, **IN1** and **IN2** are assigned to the motor control, while **ENCODER_A** and **ENCODER_B** are for the encoder inputs. The switch pin and choice pin are also defined here. This setup is crucial for ensuring the correct physical connections between the microcontroller and the hardware components.
The task starts in the `setup()` function, where the pins are configured as either inputs or outputs. The servos and ToF sensors are also initialized in this function. The servos are attached to their respective pins using the `attach()` method, which prepares them for receiving control signals. The ToF sensors are initialized using the `begin()` method to ensure they are ready for distance measurements. The `setup()` function also includes initial actions such as opening and closing the servos to set them to

their default positions. This step ensures that the system starts from a known state, which is critical for consistent and reliable operation.

Next, the flowchart indicates the use of specific functions to control the servos. The functions `closeServo(1)` and `closeServo(3)` are used to close the top and bottom grippers respectively. These functions set the servos to a specific angle that corresponds to the closed position. The `openServo(3)` function, on the other hand, opens the bottom gripper by setting the servo to an angle that corresponds to the open position. These actions are essential for manipulating objects as part of the task sequence.

The process then involves moving the arm up until a switch is clicked, using the `moveUpUntilSwitchClick()` function. This function starts the motor in the forward direction and continues moving the arm up until the switch pin reads a LOW signal, indicating that the switch has been clicked. Once the switch is clicked, the motor is stopped, and the arm is moved down by a specific number of counts using the `moveDownByCounts()` function. This movement helps in positioning the arm accurately for subsequent actions.

After the arm is moved down, the bottom gripper is closed using `closeServo(3)`, securing any object held by the gripper. The top gripper is then opened using `openServo(1)`, releasing any object held by it. These actions are part of the task sequence to manipulate objects as required.

The arm is then moved up by a specified count using `moveUpByCounts()`, which helps in lifting the object to a desired height. This movement is monitored by reading the encoder values to ensure precise positioning. The distance to an object or surface is measured using the `getTopToFDistance()` function. This function reads the distance from the ToF sensor and checks the range status. If the reading is valid, it returns the measured distance; otherwise, it returns a large value indicating out of range.

The arm is moved down by a specific count again using `moveDownByCounts()` based on the distance measurements. This step ensures the arm is positioned correctly for the next action. The top gripper is closed using `closeServo(1)` to secure the object again, followed by opening the rotator using `openServo(2)` to adjust the orientation as needed.

Finally, the arm is moved down again using `moveDownByCounts()`, and the system is moved to a homing position. The `homingPosition()` function would typically involve moving the arm to a known reference position, ensuring the system is ready for the next cycle or task. Conditional checks using if statements are employed throughout the process to ensure actions are taken based on sensor inputs and the current state of the system. These checks are crucial for adapting the task sequence to real-time conditions and ensuring accurate and reliable operation.

In summary, the flowchart for Task 1 outlines a detailed sequence of actions involving servo control, motor movement, and sensor measurements. Each step is carefully orchestrated using specific functions to manipulate the arm and grippers, ensuring precise and reliable task execution. The use of encoder values and ToF sensor measurements ensures accurate positioning and control, essential for successful task completion.

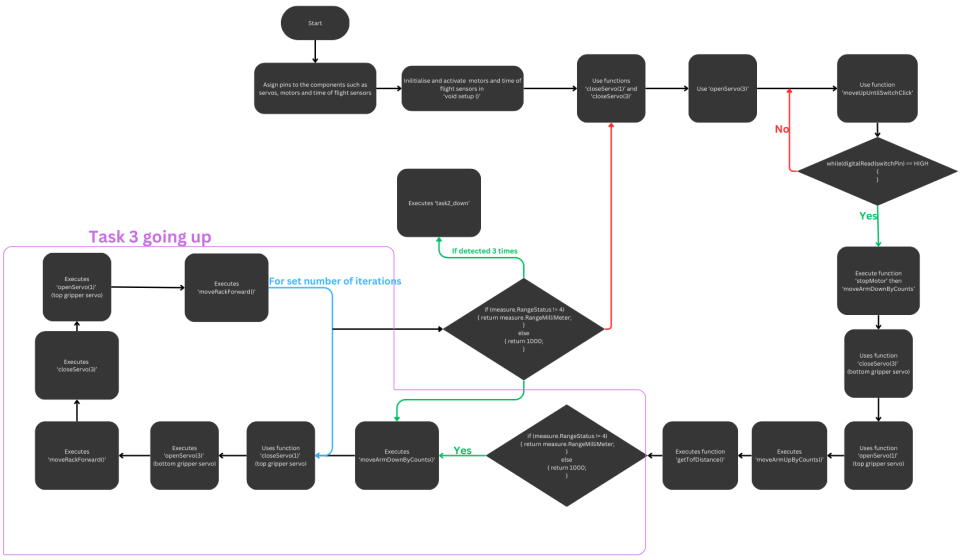

Figure 18. Task 3 code Flow chart

The flowchart for Task 3 outlines a series of operations aimed at moving an arm up and down while managing gripper and rack-pinion operations, ensuring the system functions correctly and efficiently.
The process starts with initializing and activating motors and Time of Flight (ToF) sensors in the `setup()` function. This involves configuring pins as outputs or inputs and attaching servos to their respective pins using the `attach()` method. Additionally, ToF sensors are initialized using the `begin()` method, ensuring they are ready to measure distances accurately.
The task begins by executing the `moveUpUntilSwitchClick()` function. This function initiates the motor in a forward direction, moving the arm upward until the switch pin registers a LOW signal, indicating the switch has been clicked.
Inside this function, a loop runs continuously, checking the state of the switch pin using `digitalRead(switchPin) == HIGH`. Once the switch is triggered, the `stopMotor()` function is called to halt the motor, followed by `moveDownByCounts()`, which moves the arm down by a predefined number of counts using the encoder readings. This sequence ensures the arm is precisely positioned for subsequent actions.
The next step involves closing the bottom gripper using `closeServo(3)`, which sets the bottom gripper servo to a position that secures any object it holds. Similarly, the top gripper is opened using `openServo(1)`, setting the servo to a position that releases any object it may hold. These actions are critical for handling objects during the task.
After these operations, the arm is moved up by a certain number of counts using `moveUpByCounts()`, a function that uses encoder readings to ensure precise upward movement of the arm.
The arm's position is monitored using the `getTopToFDistance()` function, which measures the distance from the top ToF sensor. This function checks the range status of the sensor; if the status is valid, it returns the measured distance in millimeters, otherwise, it returns a default value of 1000, indicating an out-of-range condition. This distance measurement is crucial for determining the subsequent movements of the arm.
Based on the distance measured, the arm is moved down by a specific count using `moveDownByCounts()`. This adjustment ensures the arm is accurately positioned for the next action. Following this, the top gripper is closed using `closeServo(1)`, securing the object, and the bottom gripper is opened using `openServo(3)`, releasing any object it may hold.
The next part of the process involves executing `moveRackForward()`, which moves the rack and pinion mechanism forward. This movement is performed for a set number of iterations, as indicated in the flowchart. The rack and pinion movement is critical for adjusting the horizontal position of the arm or object, ensuring it is correctly aligned for further operations.
The system then enters a loop to execute `task2_down`. This part of the task involves reversing the

actions performed during the upward movement, ensuring the arm moves down correctly while handling objects as needed. The actions include closing and opening servos and moving the rack and pinion backward.

In summary, the flowchart for Task 3 provides a detailed sequence of actions involving servo control, motor movement, and distance measurements using ToF sensors. Each step is meticulously designed to manipulate the arm and grippers, ensuring precise and reliable operation. The use of encoder values and ToF sensor measurements ensures accurate positioning and control, essential for the successful completion of the task. The conditional checks and loops ensure that the system adapts to real-time conditions, maintaining the reliability and efficiency of the task sequence.

4 MATHEMATICAL EVALUATION

Here are the plots of voltage, current and power as functions of time:

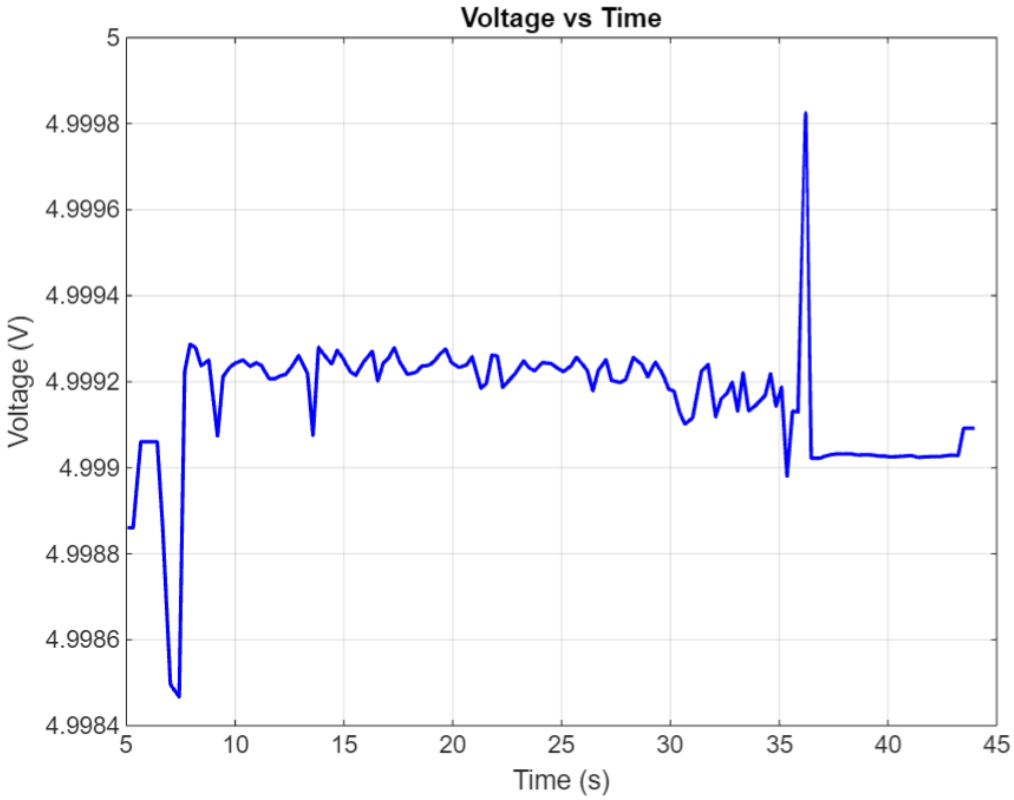

Figure 19. Voltage Plot

As seen in the Figure 19, for most of the observation period, the voltage levels remain stable around 4.992 V. Between 5 to 10 seconds, there are initial fluctuations where voltage drops slightly below 4.999 V.

There is a significant spike and drop in voltage around 35 seconds. This sharp fluctuation could be indicative of a sudden change in operational state or load. Most likely, this occurred during the descent phase of the wheeled robot.

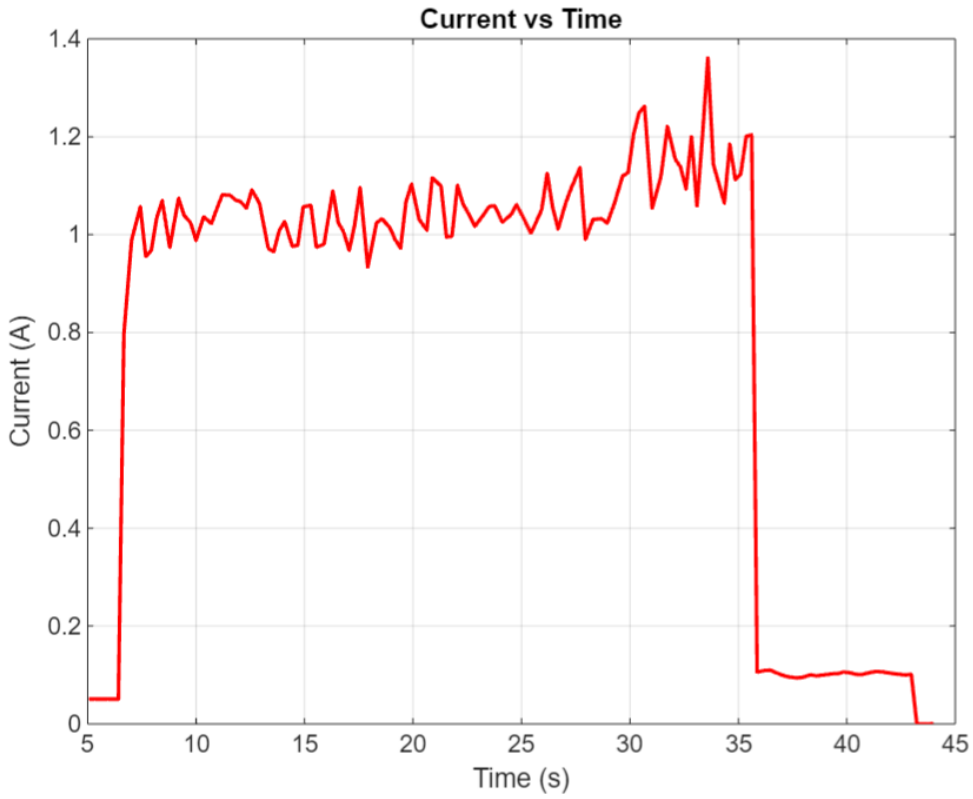

Figure 20. Current Plot

From the current plot in Figure 20, it can be observed that the current drawn by the robot is relatively stable around 1 A for the majority of the time, indicating a consistent load. The current quickly rises from 0 to around 1 A, indicating the robot transitioning from an idle to an active state. This transition from idle to active state likely represents the ascent phase of the robot, during which it draws more current to move upwards and counteract gravity.

There are noticeable spikes in the current around 30 seconds, reaching values higher than the average (peaks approaching 1.2 A). This suggests a period of higher activity or load. Furthermore, from 5 to 30 seconds, the current is almost stable around 1 A which suggests consistent effort in climbing. The spikes in current indicate the robot might be navigating more difficult terrain, such as overcoming obstacles or adjusting its grip on the tree.

The current sharply drops back to 0 A, indicating a return to an idle or low-power state. This low power state is when the robot eventually comes down and settles at the bottom of the tree.

In summary, the robot's operation can be divided into distinct phases based on the voltage and current data. Initially, the robot is in an idle state before 5 seconds, as indicated by slight fluctuations in voltage around 4.999 V and current close to 0 A. This suggests the robot is either preparing to start its climb or is in a state of minimal activity. From 5 to 30 seconds, the robot enters the ascent phase, characterized by stable voltage around 4.9992 V and current around 1 A with minor oscillations. This steady current indicates the robot is consistently climbing the tree, maintaining a **moderate to high load**. Around 30 seconds, there are minor spikes in voltage and peaks in current reaching up to 1.2 A, indicating the robot encounters increased resistance or a challenging section of the tree, requiring more power to overcome obstacles or adjust its grip. After 35 seconds, the robot transitions to the descent phase, where the current sharply drops to near zero, and the voltage stabilizes, reflecting minimal power consumption. This significant reduction in current suggests a **low load** and that descending the tree is less power-intensive compared to climbing.

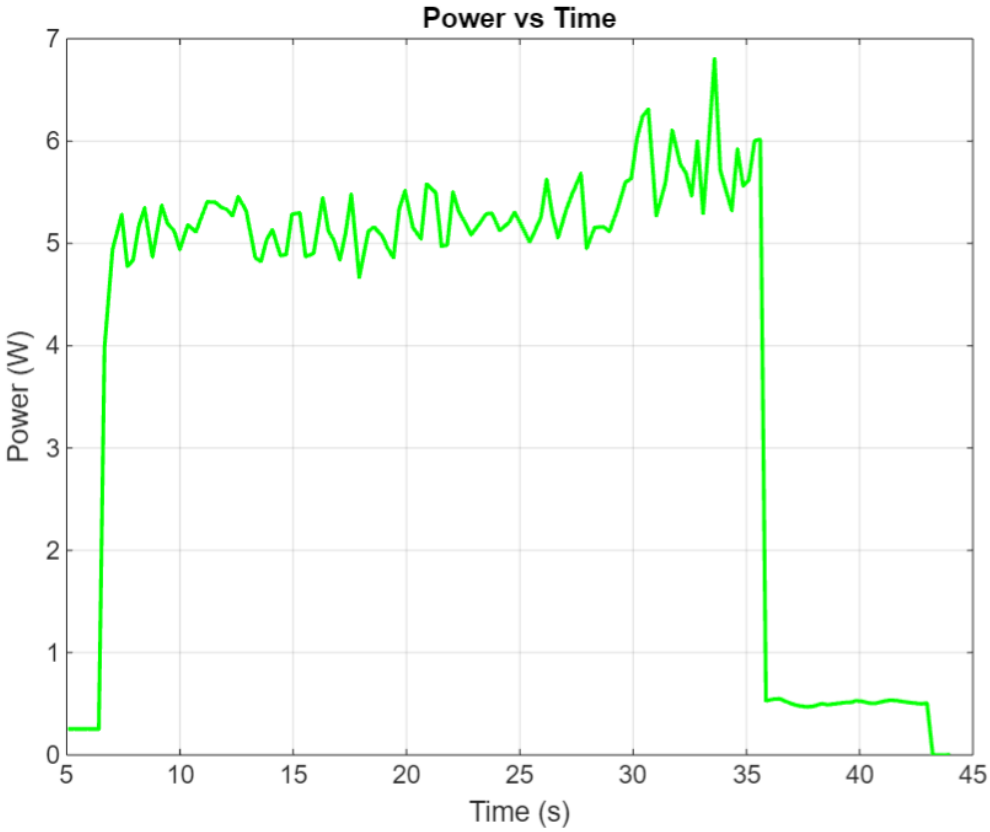

Figure 21. Power Plot

Since Figure 21 represents instantaneous power consumption, it shows the exact power usage at each second. Similar to the current and voltage plots, the power consumption plot illustrates a comparable trend. There is a sudden spike in power from 0 to 5 W within the first 5 seconds, indicating that the robot has started the ascent phase and needs to overcome inertia, gravity, and start the motors. Between 5 to 30 seconds, the power fluctuates around 5 W. After 35 seconds, the power consumption abruptly reduces, indicating that the robot is descending and requires less current for the motors compared to climbing. The robot's power consumption averaging around 5 W during the climbing phase seems reasonable for a small, wheeled climbing robot. This average provides a baseline understanding of the robot's energy demands during active operation. In the context of battery-operated or energy-efficient systems, maintaining power consumption around 5 W indicates a balanced design, prioritizing performance without excessive energy use.

The RMS power consumption formula is:

$$\text{RMS Power} = \sqrt{\frac{1}{N}\sum_{i=1}^{N} P_i^2} \tag{1}$$

In this equation, P denotes the different power values in the datasheet, with N being the number of values (to calculate the mean). The RMS power value, as calculated on Matlab (Figure 22), was 4.59 W, indicating that the robot's average power consumption during the climbing task is relatively moderate. This value suggests that the robot's power management system is efficient, ensuring it can perform energy-intensive tasks like climbing while maintaining a reasonable power consumption level. This efficiency is crucial for autonomous operations, especially in environments where power resources are constrained.

```
% Calculating RMS power
N = length(power);
rms_power = sqrt(mean(power.^2));
```

Figure 22. Computation of RMS Power

By maintaining an RMS power consumption of 4.59 W, the robot demonstrates its capability to balance performance with sustainable power consumption, which is a significant advantage for prolonged operations and for scenarios where the robot might need to perform multiple tasks or operate over extended periods without recharging.

The following total energy consumption values were computed using the cumtrapz function in Matlab with power being integrated with respect to time.

```
time = robotdata{:,1};
voltage = robotdata{:,2};
current = robotdata{:,3};

power = voltage .* current;
cumulative_energy = cumtrapz(time, power);
```

Figure 23. Computation of Total Energy

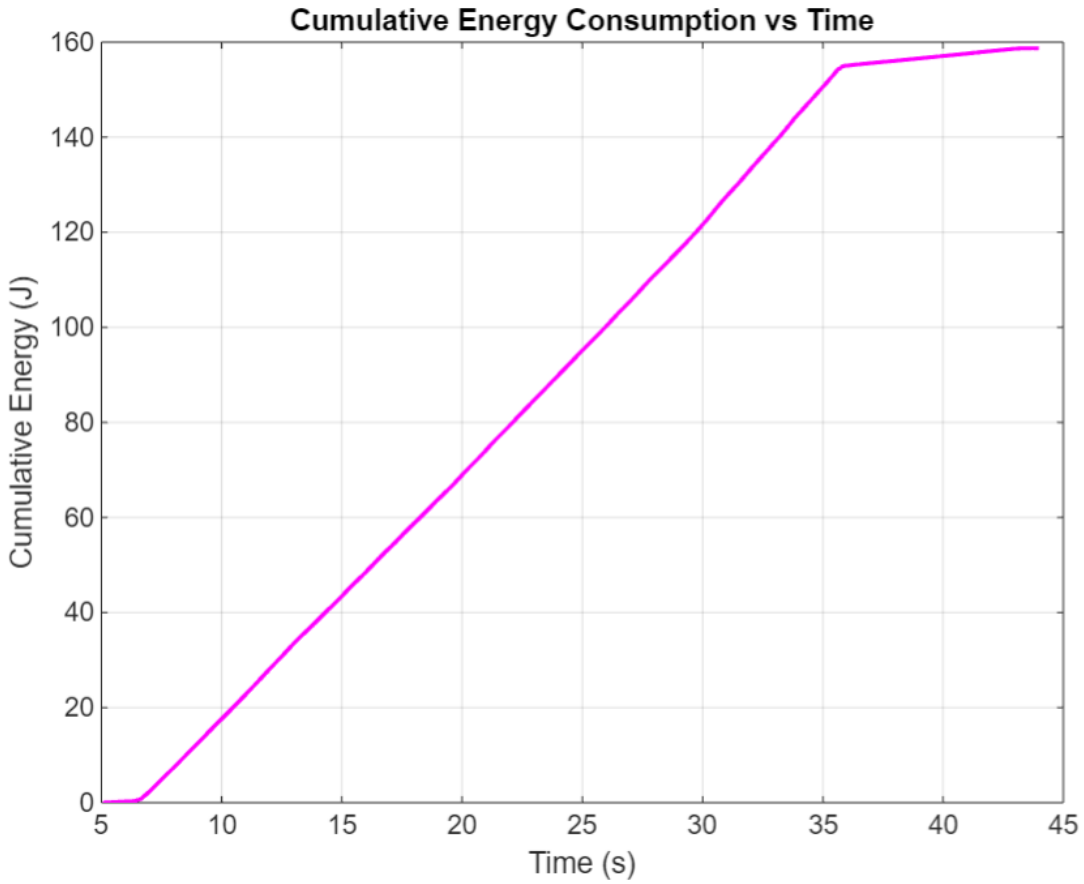

Figure 24. Energy Plot

The cumulative energy consumption plot shows a steadily increasing trend from the start to around 35 seconds, after which it levels off. This indicates that the robot was consistently consuming energy during this period and then transitioned to a state of minimal or no energy consumption after 35 seconds. The rate of increase in cumulative energy remains linear, which is consistent with the robot encountering more challenging sections of the tree, such as branches or rough surfaces, requiring sustained energy input. There are no visible fluctuations in the energy consumption, which aligns with the observation that the robot followed a single path during its climb and descent. As the robot began its descent around 35 seconds, the slope of the energy consumption graph decreased, indicating that descending required relatively less power and load. The total energy consumption of approximately 160 Joules (the precise value was 158.8J which was found using trapz in matlab) provides insight into the energy requirements for similar tasks, which can be useful for planning and optimizing future operations.

Efficiency is a crucial metric for determining the effectiveness of a system. Energy efficiency can be calculated using the following formula:

$$\text{Efficiency} = \frac{\text{Useful Energy Output}}{\text{Total Energy Input}} \times 100 \tag{2}$$

In our case, the robot weighs 300 grams (which is 0.3 kg). The height of the tree is approximately 250 cm, which is equivalent to 2.5 meters. The mechanical power output can be represented by the gravitational potential energy (GPE) the robot gained while climbing. Hence, to calculate the useful energy, we use the formula for gravitational potential energy (GPE), which determines the useful energy consumed:

$$\text{GPE} = m \cdot g \cdot h \tag{3}$$

where:

m is the mass (0.3 kg), g is the acceleration due to gravity (9.8 m/s) h is the height (2.5 m). Substituting the given values:

$$\text{GPE} = 0.3, \text{kg} \times 9.8, \text{m/s}^2 \times 2.5, \text{m} = 7.35, \text{J} \tag{4}$$

The electrical power input is the total energy consumed by the robot, which was calculated using the cumulative trapezoidal integration of power over time and found to be 158.8 J.

$$\text{Efficiency} = \frac{7.35, \text{J}}{158.8, \text{J}} \times 100 = 4.62 \tag{5}$$

Hence, the efficiency of the system is 4.62 percent. An efficiency of 4.62 percent is quite low which could be due to the fact that significant energy was lost because of friction and resistance in the robot's mechanical components, such as gears and wheels. These components had to overcome gravitational forces and the resistance of the tree's surface. Moreover, the control system that manages the robot's movements also consumes a portion of the electrical energy, which is where the energy might have dissipated. This includes the energy required for sensors, processors, and other electronic components. Furthermore, some energy might be lost as heat in the motors. Climbing a tree is a mechanically intensive task that requires the robot to constantly adjust and exert force to maintain stability and progress upwards. This likely adds to the energy consumption due to frequent starts, stops, and adjustments. These might be the possible reasons for the computed value of the energy efficiency.

5 DISCUSSION AND CONCLUSION

Overall, the challenge had its own successes and difficulties. In the challenge, the linear actuator worked very well as it had high torque which translated to faster linear velocity; however, the gripper did not work due to a lack of torque, which could have been mitigated by using springs in the design. The robot was able to ascend and descend with some assistance, providing extra torque to the top servo that controlled the upper gripper. We also successfully deployed the sensor, allowing the robot to climb down the tree successfully. However, the robot struggled to climb the tree in scenario 2 because the bends made it difficult for the gripper to fully grasp the tree.

The biggest takeaways were learning how to troubleshoot the robot before testing it. Being able to identify and fix errors—such as issues with wiring using a multimeter—was a significant achievement. Our biggest takeaway was designing a gripper that did not erode over time. We went through 4 different designs before finding something that worked. First, it was 3D printing the servo horn, but that needed fine tolerances and eroded quickly. We landed on the idea of using a metal servo horn provided with the servo which ended up working perfectly. Also, before, we had no experience working with bearings, but after using them for the gripper, we can see how simple they are to use. We also learned how best to manage power in a robot and prevent voltage spikes that can turn the MCU on and off unintentionally. Additionally, various components, like the threaded rod, broke during the manufacturing process. These challenges are all part of the learning curve, helping us to avoid repeating the same mistakes in the future. Additionally, exploring alternatives when something wasn't working was crucial. For example, our design was highly efficient, with each part serving a specific function. We identified and removed every redundant part, which was a rewarding process. Another major takeaway was realizing that just because something works in theory doesn't mean it will work practically. Practical implementations come with their own challenges, and even small issues like weight imbalance can disrupt the overall performance of the algorithms. For example, while the rotation mechanism seemed fancy and feasible on paper, practical factors such as lighting (which affected the time-of-flight sensor) and the precise initial placement of the robot were critical for proper functionality. The rotation mechanism ultimately did not work as expected, and the servos began to wear out. It was an amazing discovery to find out how applying lubricant to the servos can prevent further wear and tear and fix the output torque they provide. Another surprising discovery was finding out how simply it is to work with gears and how well they can mesh together. Thereby, this challenge taught us that every problem has a solution—you just have to be willing to explore and solve whatever issues arise.

Moreover, this challenge taught us to maximize our potential using the available resources. We were constrained to using only the equipment in the lab, which added another layer of difficulty. This experience mirrored real-world scenarios where you don't always have access to everything you need, so you must make the best use of what's available. This was an important life lesson.

REFERENCES