MEEG666: Development of Control Systems for a Mobile Crop Sensing Robot

Independent Study Report

Name: Annamalai Muthupalaniappan **UD ID:** 702724707

Table of Contents:

Abstract:

This study focuses on the design and development of an automated control system for a novel mobile crop-sensing robot capable of traversing the truss rods of linear move or center pivot irrigation system [1]. The primary challenge involves enabling the robot to detect and navigate truss pockets (~20 feet apart) while maintaining stable and efficient locomotion between end towers. A prototype featuring three articulated arms with drive and up-stop wheels, as well as an internal reaction wheel for stabilization, has been developed in the UD Digital Ag Lab. A Finite-state machine is designed to implement real-time control systems addressing speed and stabilization manually for now using the RC controller. While the localization, and obstacle avoidance is taken care of by integrating the MID360 lidar system with the robot and experimenting it to detect the horizontal and vertical truss rods. The work is progressing towards autonomous control which advances robotic automation in agricultural settings,

with potential applications in precision agriculture [3] for real-time crop and soil data collection at field scales.

Introduction:

In modern agriculture, the center pivot irrigation systems [1] are widely used to efficiently water crops over large fields. These systems consist of a central pivot point connected to a long, horizontal truss structure equipped with sprinklers, which rotates in a circular pattern to distribute water evenly. Linear move irrigation systems operate similarly but travel in a straight line. These systems offer precision in water application but present structural challenges, such as the presence of truss pockets, which complicate the integration of automated technologies.

Mobile crop gantry robots [2] represent an innovative solution for enhancing precision agriculture practices. These robots operate on irrigation system structures, navigating truss rods to collect real-time crop and soil data. Their potential lies in addressing critical agricultural challenges, such as optimizing resource use, collecting useful data for improving crop management and agricultural practices and quality. However, designing robots that can reliably traverse the unique structural obstacles of irrigation systems, such as truss pockets, requires advanced control systems and stabilization mechanisms.

Modern agricultural practices increasingly rely on precision technologies to meet the demands of growing populations and environmental sustainability. Precision agriculture [3] leverages real-time data to improve crop yields, optimize inputs such as water and fertilizers to reduce environmental impact. The structural complexity of irrigation systems poses significant challenges. Additionally, automating processes in these environments requires overcoming obstacles such as localization accuracy, obstacle avoidance, and system stability under dynamic loads.

This study addresses these challenges by developing and refining a mobile crop-sensing robot to autonomously traverse and operate within the constraints of irrigation systems, contributing to more efficient and informed agricultural practices.

Problem Statement:

The integration of autonomous systems in agriculture presents a transformative opportunity to optimize resource use and improve decision-making through real-time data collection.

The goals of this independent study are to

1. Develop expertise in real-time robotic control systems that focus on speed, stabilization, localization, and obstacle avoidance.
2. Design a path planning algorithm for the robot in the center pivot based on the given conditions and requirements.
3. Enhance problem-solving and critical thinking skills with the unique challenges under agricultural field conditions.
4. Advance knowledge in precision agriculture [3] that potentially enable farmers to obtain real-time crop and soil information at production field scales.

Rivulet 2.0:

The robot Rivulet 2.0 is a mobile crop sensing robot (Gantry systems) [2] that can be mounted in Centre pivot irrigation systems [1]. The robot moves back and forth in between the towers as directed by the user. While collecting data and useful information that addresses and tries to solve the farmer's needs.

The robot's 1st version was developed in the UD's Digital Agriculture lab by the Junior design team. The 2nd version is an improved design of the previous one with some changes like addition of up-stop wheels to the robot arm which locks it in the horizontal truss rod and allows the robot to open/ close its arms while crossing the truss pockets/obstacles. The arm is designed in a way that it uses linear actuators to open/close and has drive wheels to make it move in the rod in between the towers and overcome the truss pockets (every ~20 feet) while traveling on the rods back and forth.

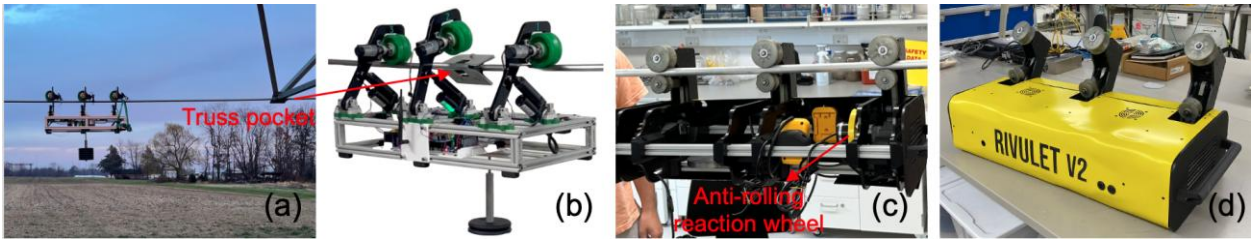


Figure 1. *Version 1 (a, b) and version 2 (c, d) of a mobile robotic sensor platform for center pivots.*

The robot is designed to travel on a truss rod and go over the truss pocket. Compared to V1, V2 uses up-stop wheels to prevent the robot from falling off the truss rod and has an anti-rolling reaction wheel for robot stabilization.

Components Used:

The initial design of the robot has some components which are used to build the robot. These components were studied and programmed by following the respective communication protocols to design the control system. They are

Electrical Components:

1. Servo Motors - My Actuator RMD-L-4015 (for drive wheels): (link) / RMD-L-5015 (for internal reaction Flywheel) : (link)
2. Linear Actuator - Utility Linear Actuator (link)
3. Microcontroller - Adafruit Feather M4 CAN Express (link)
    a. Motor Driver - PN00218-CYT(Cytron 13A DC Motor Driver) (link)
4. RC Controller - Flysky FS-I6s with FS-IA6B Receiver
5. Lidar Sensor – Livox Mid360 (link)

Software and Tools used: Arduino IDE, Livox Viewer, ROS, Rviz, Circuit Python, Visual Studio Code,

Programming Languages: Python, C++

Methodologies:

The methodologies and concepts used to achieve the goals are described below. Firstly, let's start with achieving manual control of the robot using Finite-State Machine concept (FSM). Which is then followed by a path planning algorithm that simulates and displays the optimal coverage for the given area. Further followed by obstacle detection module using the MID360 Lidar which is used to detect the obstacles like truss pockets and horizontal/vertical trusses leading our way to autonomy.

1. Finite-State Machine:

A Finite-State Machine (FSM) is a computational model used to design systems with discrete and predictable states. It operates by transitioning between a set of defined states based on external inputs or conditions, ensuring organized and efficient behavior. FSM is widely applied in robotics, embedded systems, and control systems due to its structured approach to handling complex decision-making processes.

The reason why we chose FSM is because they have several advantages like

1. **Modular Design**: FSM divides functionality into discrete states, making the code modular, organized, and easier to debug.
2. **Scalable System**: Adding new behaviors or states becomes straightforward without affecting existing functionality.
3. **Improved Readability**: The code is structured into state-specific functions, making it easier to understand and maintain.
4. **Efficient Decision Making**: FSM ensures that only the relevant state-specific logic runs at any time, improving system efficiency

Code: (link)

Implementation in the Robot "Rivulet V2.0"

In this project, the FSM is implemented to control the behavior of the robot "Rivulet V2.0." The FSM defines the robot's states and transitions based on input signals from a remote controller. This enables smooth and dynamic operation of the robot for various tasks.

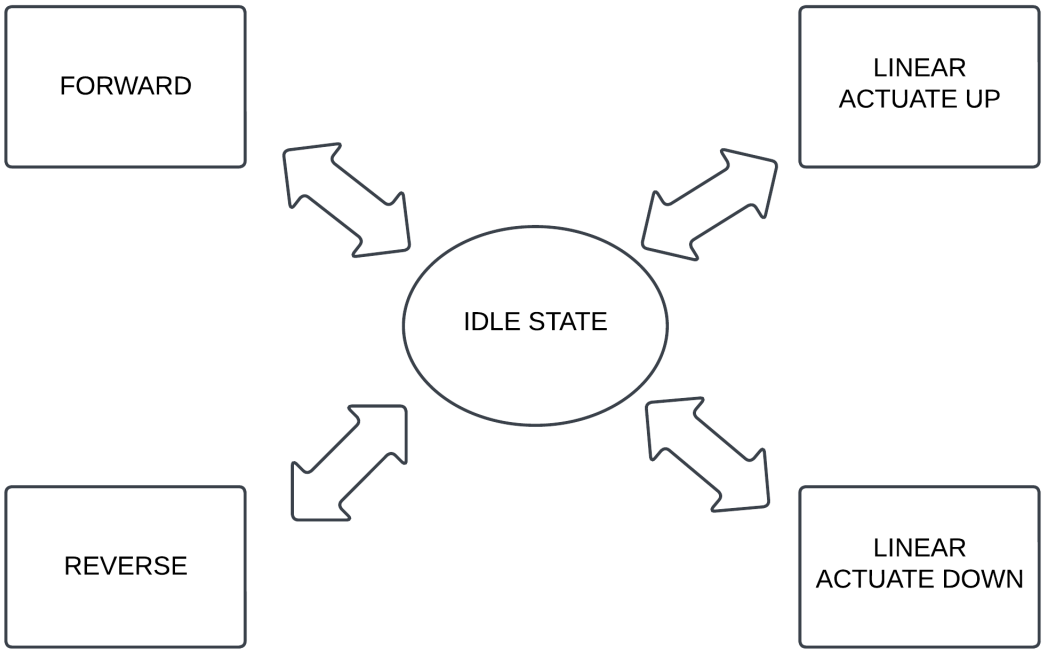
Telemetry:

Input Signals processing: In order to implement the FSM, we first need to process and filter the input signals and assign respective controls for it. Which makes our telemetric control over the remote control successful. The telemetry and input signal processing in the code revolves around interpreting signals from an RC controller (Flysky FS-I6s) to control the robot's state and actuators.

States of the Robot

The robot operates with the 5 key states: The following are

1. **Forward**: Moves the robot forward using its motorized wheels.
2. **Reverse**: Moves the robot backward.
3. **Linear Actuate Up**: Extends the linear actuators to help the robot navigate obstacles or adjust its configuration.
4. **Linear Actuate Down**: Retracts the linear actuators for stabilization or locking the robot in place.
5. **Idle**: Keeps the robot in a stationary state while continuously monitoring input signals.



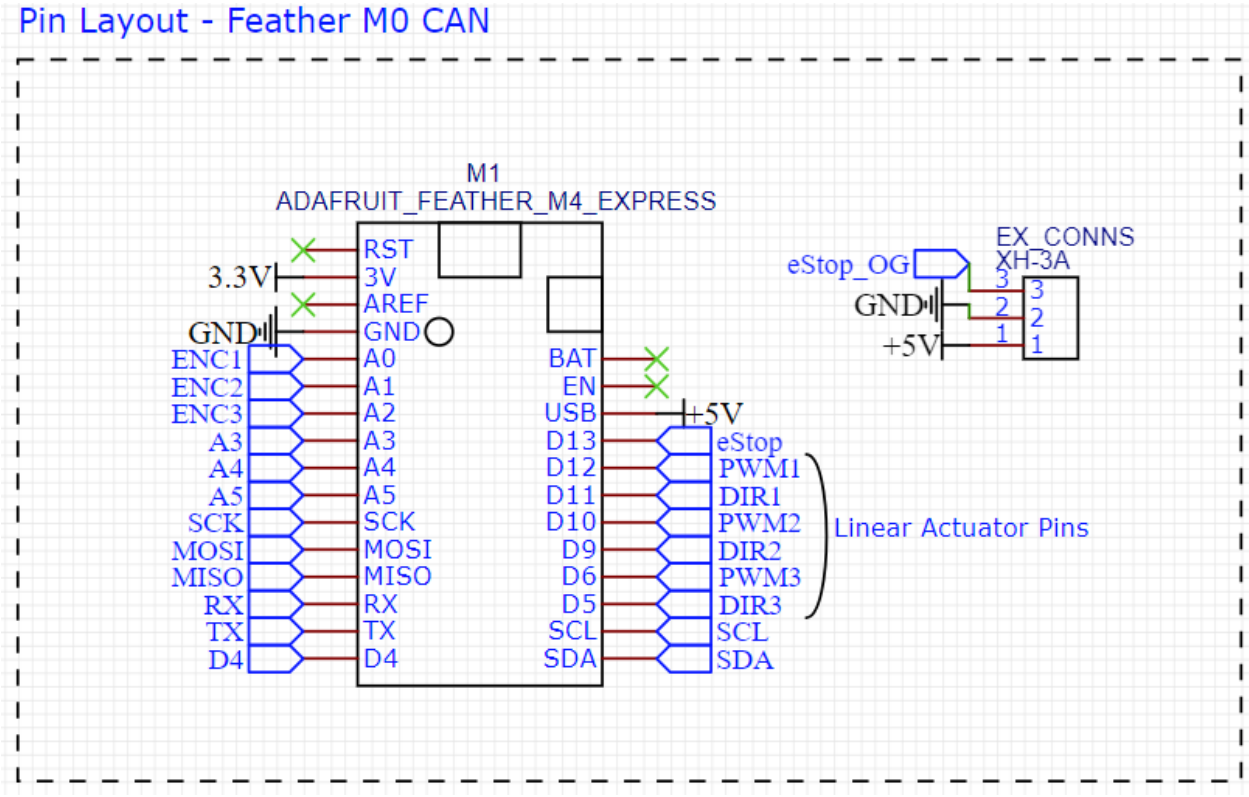
Input signals Configuration:

1. **RC Channel Inputs:**
   - Channel 1 (CH1): Connected to Pin 0 (Rx): Reads joystick values for forward and reverse movement.
   - Channel 2 (CH2): Connected to Pin 1 (Tx): Reads joystick values for actuating the linear actuators.
   - Channel 5 (CH5): Connected to Pin 22 (SCL): Determines which actuator to move based on its switch state.
   - Channel 6 (CH6): Connected to Pin A4.: Reads the state of a switch (SWA), acting as a mode selector for movement or actuator control.
2. **Motor Driver Pins:**
   - Linear Actuator Drivers:

1. Actuator 1: PWM = Pin 12, DIR = Pin 11.
2. Actuator 2: PWM = Pin 6, DIR = Pin 5.
3. Actuator 3: PWM = Pin 10, DIR = Pin 9.

3. Pin Diagram:

The circuit and pins are connected as shown below.



Note: The PCB Board connections are given on the backside of the PCB Board.

FSM Flow and Implementation in Code

1. **Input Handling**:

   Functions like readChannel() and readSwitch() process the input signals from the RC controller, converting them into usable data for state transitions. The readChannel function is used to read pulse width values (in microseconds) from the RC receiver, map them to a specific range, and provide default values when the channel is inactive.

   The channel signals are interpreted based on pulse width:

   - Range: 1000–2000 µs.
   - Mapped to values: -100 to 100.

The readSwitch function interprets toggle switch states from specific channels. It outputs a boolean (true/false) depending on the position of the switch.

The RC controller signals (CH1, CH2, CH5, CH6) are continuously read and processed. These values determine the robot's state (e.g., forward, reverse, idle, linear_actuate_up, or linear_actuate_down).

2. **Dynamic State Transition**:

   A global pointer state variable, that tracks the current state of the robot. Based on input values given above (e.g., joystick channels and switches), the code dynamically updates the state pointer to the corresponding function.

3. **State Handlers**:

   Each state (e.g., forward(), reverse(), etc.) is implemented as a function that performs specific tasks and checks conditions for state transitions.

   - These handlers manage motor speeds, actuator positions, and CAN BUS commands.

4. **Control Commands**:

The FSM uses CAN BUS commands to communicate with the motors and actuators, ensuring precise control. The convert_to_Can() function converts a speed value into a CAN command (8-byte array). This array encodes speed as a hexadecimal value and forms a valid CAN protocol command for motor control.

   - For example, in the **Forward** state, the speed of the motors is calculated from joystick inputs and sent as CAN commands.

Advantages of the FSM Approach

Each state is implemented as an independent function, making the code easy to manage and extend new states or behaviors which can be added without disrupting existing functionality. The FSM efficiently processes inputs and transitions between states, ensuring quick response to user commands.

Key features and summary of the algorithm:

1. **Dynamic Movement**:

   - The **Forward** and **Reverse** states allow the robot to move in both directions, dynamically adjusting the speed based on joystick inputs.

   - A CAN BUS communication protocol sends precise motor control commands to the wheels.

2. **Obstacle Management**:
    - In the **Linear Actuate Up** and **Linear Actuate Down** states, the robot can extend or retract its actuators, enabling it to navigate obstacles (truss pockets) or secure itself on uneven terrain.
    - PWM signals control the actuators for precise movement.
3. **(Safe/ Stop) Idle State**:
    - The **Idle** state ensures the robot halts all movement when no control signals are received, maintaining safety and readiness for new commands.
4. **Seamless State Transitions**:
    - Transitions between states are governed by inputs from an RC controller (joystick and switches). For example:
        - Moving the joystick forward transitions the robot to the **Forward** state.
        - Releasing the joystick transitions it back to the **Idle** state.

Workflow and Results:

The following images illustrate the step-by-step workflow of the manual control system, providing a clear understanding of its operation. They also demonstrate how the Finite State Machine (FSM) is implemented in the system and outline how the same concept can be extended to an autonomous FSM, showcasing its potential for fully automated operation.

Step 1: The first step to set up the robot as shown. The robot is mounted on the horizontal truss, and it is operated by Flysky FS-I6s RC controller. It remains in the idle state.

Figure 2.1. *Displays the Rivulet 2.0 robot hanging in the Truss Rods*

Step 2: The robot is controlled and moved (forward and reverse states are used) by switching the states (Finite-state machine). As the robot moves it reaches the first obstacle, the Truss pocket. Once detected it opens its first arm (linear actuator up state is used) and passes through it and then closes its arm back (linear actuator down state is used).

Figure 2.2. *Displays how the robot's 1st arm navigates the obstacle (Truss pocket)*

Step 2: Similarly, the robot extends its second arm to cross the truss pocket and closes and makes way to the 3rd arm.

Figure 2.3. *Displays how the robot's 2nd arm navigates the obstacle (Truss pocket)*

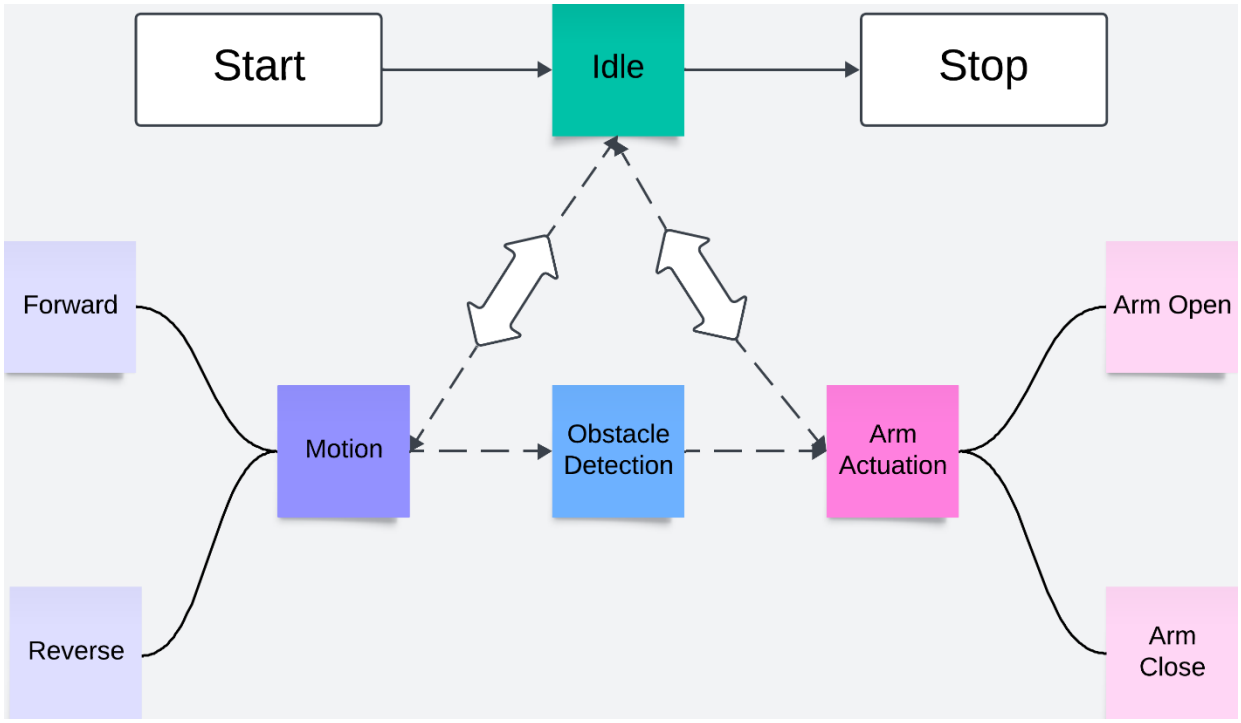
Step 3: Similarly, the robot extends its 3rd arm to cross the truss pocket and closes and proceeds to its normal operation (Collecting data).

Figure 2.4. *Displays how the robot's 3rd arm navigates the obstacle (Truss pocket)*

Step 4: Now the robot moves as normal in the horizontal truss until it meets the next obstacle (Truss pocket). The truss pockets are present at every 20m of the pivot. So, the same steps (1 – 3) of operations are repeated and the robot traverses back and forth in the pivot while collecting the data.

FSM for Autonomous System:

The figure below illustrates the workflow of the design plan for implementing the FSM in the autonomous system for this application.



2. Motion and Path Planning:

Now the setup can be manually operated within a center pivot irrigation system [1]. The setup operates within a center pivot irrigation system [1], designed to irrigate a circular field with a radius of approximately 200 meters. The pivot is centrally positioned and rotates around the field, completing a full rotation in about 6 hours. This rotational motion ensures uniform water distribution across the circular geometry of the plot.

Mounted on the horizontal truss rod of the system, our robot traverses back and forth, navigating obstacles such as truss pockets while collecting data from the field. To optimize this process, we developed an algorithm that simulates the environment and calculates a path to ensure uniform coverage and efficient data collection, maximizing the utility of the robot's operations across the field.

Code: (link)

The provided Python script Algorithm simulates and visualizes the motion of a robotic system using prismatic and revolute joints. The simulation calculates the robot's trajectory within a defined workspace, evaluates its coverage of the workspace, and visualizes the results on a hexagonal grid. The main focus of the script is to model a robot's operation over time and analyze its spatial coverage through computational geometry and visualization techniques. The flow of the algorithm is designed as shown below.

Motion Simulation and Trajectory Generation:

The robot's motion combines a prismatic joint for radial extension/retraction and a revolute joint for angular rotation. The prismatic joint alternates between a maximum extension (L_max) of 197 meters and a list of minimum extensions (L_min_list), while the revolute joint rotates at a constant angular speed (omega). The motion is simulated in discrete time steps, with the trajectory data being recorded as arrays of radial distances (r), angular positions (theta), and corresponding timestamps (t).

The radial motion alternates between extension and retraction phases, with the time required for each phase determined by the prismatic joint's maximum speed (v_max). These radial distances are then combined with angular displacements to calculate the Cartesian coordinates (x, y) of the robot's tip using polar-to-Cartesian transformations. This results in a spiral-like trajectory within the robot's workspace.

Hexagonal Grid Definition and Mapping:

The workspace is overlaid with a hexagonal grid to evaluate the spatial coverage of the robot's trajectory. The hexagonal grid is designed with specific parameters: the hexagon size (hex_size = $3\sqrt{3}$m [Sensor's footprint]) and the maximum reach of the robot (workspace_radius). The grid centers are calculated in a staggered pattern, ensuring full coverage of the workspace. The hexagonal layout enables efficient spatial partitioning and coverage analysis, which is particularly useful for tasks requiring even distribution of robot operations.

Each trajectory point is mapped to the closest hexagonal cell by calculating the Euclidean distance between the point and all hexagonal centers. The script tracks the number of visits for each hexagonal cell, enabling an analysis of the robot's coverage pattern.

Coverage Analysis and Results:

The script calculates the total number of hexagons within the workspace and the number of hexagons visited by the robot's trajectory. Using this data, the coverage ratio is computed as

the fraction of visited hexagons relative to the total hexagons. This metric quantifies the robot's effectiveness in covering the workspace. The results, including the total operation time, the number of hexagons, and the coverage ratio, are printed for further evaluation.

Data Visualization:

Two key visualizations are generated to illustrate the robot's motion and workspace coverage. The first plot displays the robot's trajectory, showing its path as it alternates between radial extension/retraction and angular rotation. The second plot visualizes the hexagonal grid, with hexagons colored based on the number of visits. A logarithmic normalization scale enhances the color mapping, highlighting regions of high activity while maintaining clarity for less-visited areas. The workspace boundary is also overlaid to provide spatial context.

Applications and Potential Enhancements:

This simulation is highly relevant for robotic systems that require optimized workspace coverage, such as gantry robots [2], drones, or industrial inspection systems. It demonstrates key concepts like kinematics, spatial partitioning, and coverage analysis. Potential enhancements include adding support for dynamic obstacles, animating the trajectory for real-time visualization, and introducing user-defined parameters to increase the simulation's flexibility. These additions could further improve the applicability in real-world robotic tasks.

Plots:

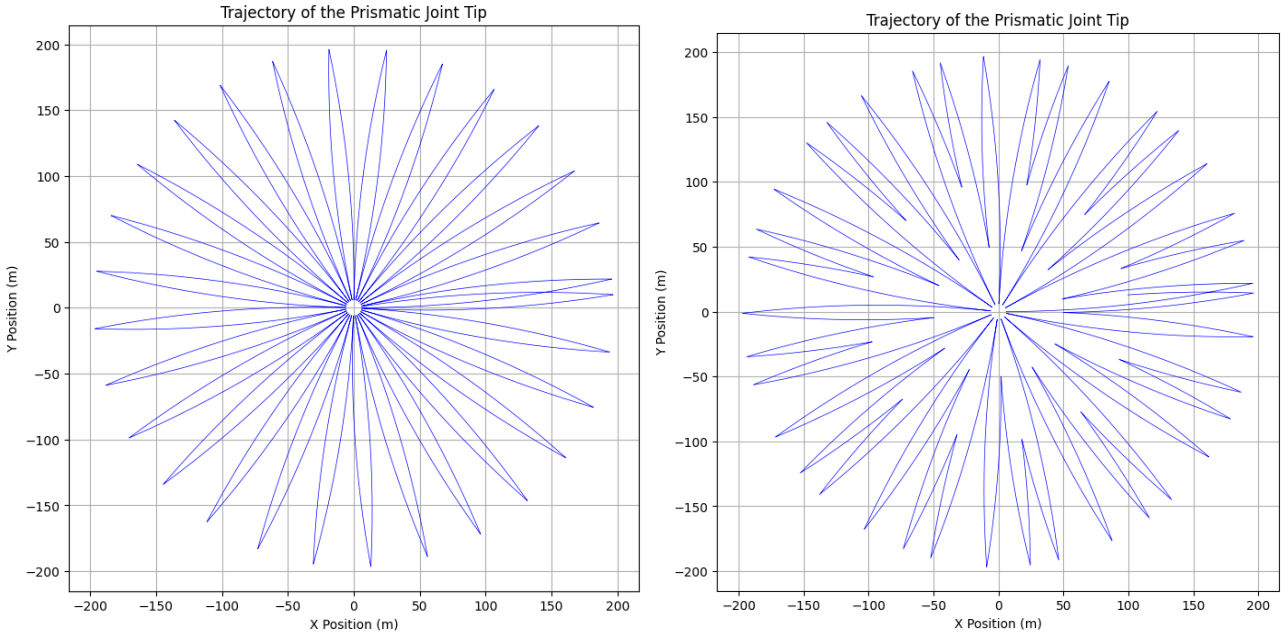


Figure 3.1 *a) shows the standard back and fro motion. b) shows the trajectory of optimized motion*

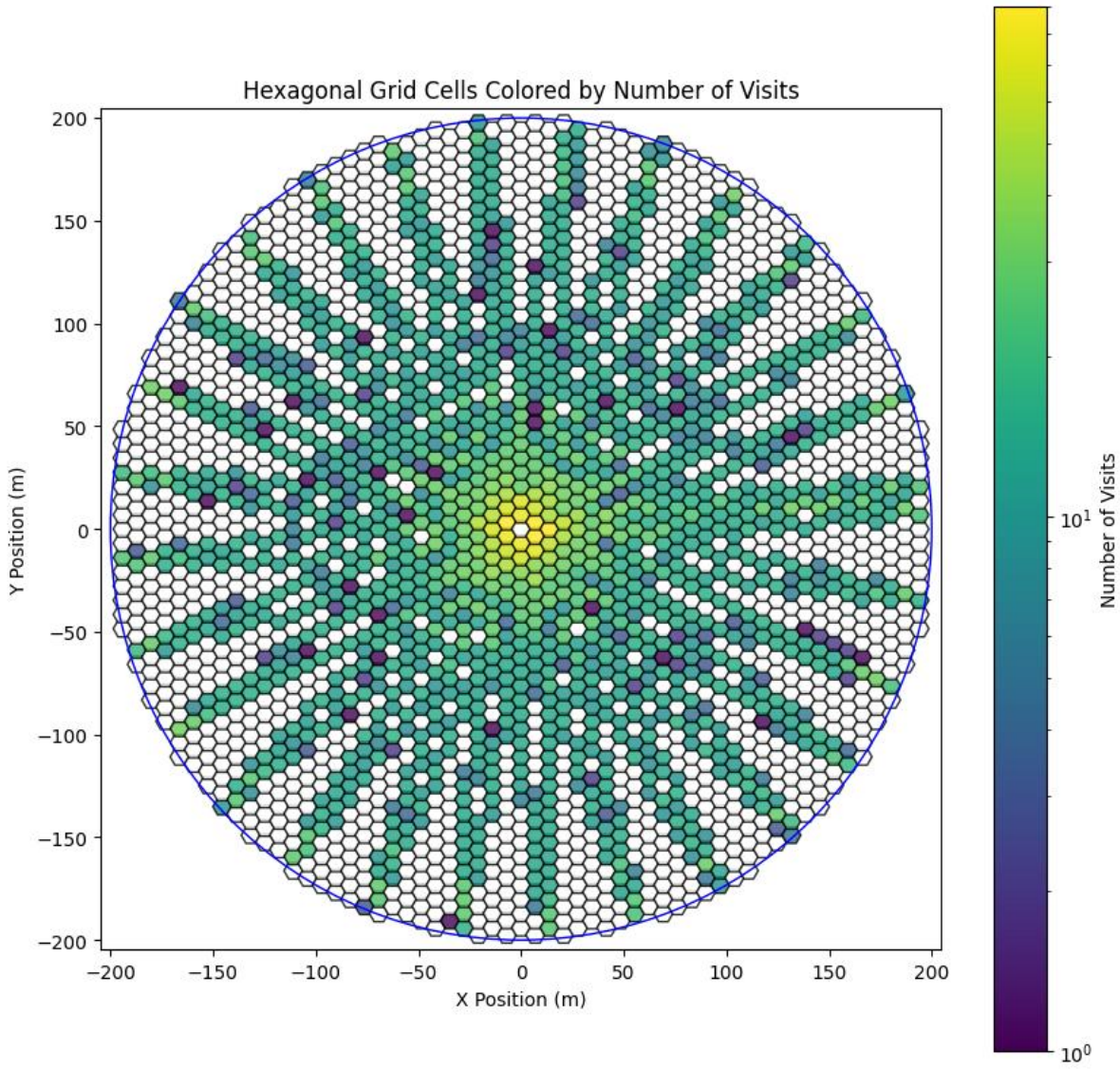


Figure 3.2. *shows the hexagonal grid with hexagons colored based on the number of visits of the standard back and fro motion*

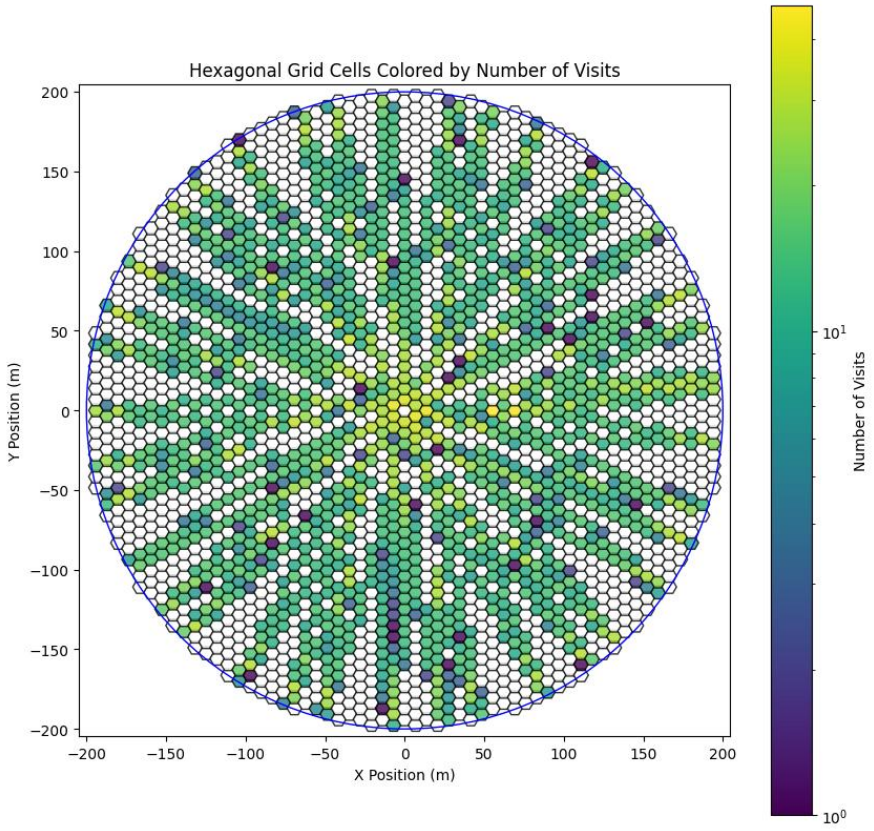


Figure 3.3. *shows the hexagonal grid with hexagons colored based on the number of visits of the optimized motion*

Results:

Based on our simulation and different iterations and tests the coverage ratio was only 54% for the standard motion and the best result we got is

Total operation time: 22044.00 seconds

Total hexagons within workspace: 2617

Number of visited hexagons: 1507

Coverage ratio: 0.5759

3. Obstacle Detection:

For us to make the robot autonomous we are using the MID360 Lidar to map the field and localize itself and detect the truss rods and pockets of the pivot.

Why did we choose livox MID360?

Mid-360 is the latest generation of Livox LiDAR for low-speed robotics. This new product inherits the cost-effectiveness of the Livox Mid series and delivers 3D perception in 360 degrees. Compact and lightweight, this LiDAR module was very easy to install, and it is optimized based on

the navigation and obstacle avoidance requirements of mobile robots. This allowed us to use the LiDAR with algorithms to deliver a wide range of functions such as SLAM [4] and obstacle avoidance.

Unlike conventional mechanical LiDAR, the Mid-360 is powered by Livox's unique rotating mirror hybrid-solid technology. And it is the first to improve the horizontal FOV to 360° and the vertical FOV to 59°. The omnidirectional ultra-large FOV coverage provided by the Mid-360 LiDAR helps mobile robots accurately perceive their surroundings in all directions. This provides the basis on which mobile robots can formulate plans and make decisions.

Methodologies:

The lidar is used along with ROS packages to detect the trusses and map the environment. Which further helps later in localization by running SLAM [4] algorithms (eg ; Fast_Lio [5]) The following are the steps involved

Ros Packages:

Inorder for us to map the environment. The lidar company has some pre-built ROS packages that publish sensory data information and visualize them using tools like Rviz and so on. The required packages are

- **Livox ROS Driver 2:**

Step 1: Cloning the GitHub repository: (Link)

After cloning the repository. Based on the ubuntu version and ROS version installed in the hardware system we use and run the respective commands given to further proceed. The system we were running was jetson Orin with Ubuntu subsystem version 20.04 and ROS Noetic.

Step 2: Build & install the Livox-SDK2

Follow the given GitHub link and install and make sure the Livox SDK is set up**.** So that it supports Livox ROS Driver 2.

Step 3: Build the Livox ROS Driver 2

Run the respective commands based on the ROS version. For ROS Noetic, we give the commands to build the package.

```
$  source /opt/ros/noetic/setup.sh
$ ./build.sh ROS1
```

Step 4: Change the Config and Run the ROS package

Change the MID360_config File from the src folder of the Livox_ros_driver2 folder as per your Lidar's IP Address. In our case the config file was altered as shown below.

```
{
```

```
"lidar_summary_info" : {
  "lidar_type": 8
},
  "MID360": {
    "lidar_net_info": {
      "cmd_data_port": 56100,
      "push_msg_port": 56200,
      "point_data_port": 56300,
      "imu_data_port": 56400,
      "log_data_port": 56500
    },
    "host_net_info": [
      {
        "lidar_ip": ["192.168.1.170"],
        "host_ip": "192.168.1.70",
        "cmd_data_port": 56101,
        "push_msg_port": 56201,
        "point_data_port": 56301,
        "imu_data_port": 56401,
        "log_data_port": 56501
      }
    ]
  },
  "lidar_configs": [
    {
      "ip": "192.168.1.170",
      "pcl_data_type": 1,
      "pattern_mode": 0,
      "extrinsic_parameter": {
        "roll": 0.0,
        "pitch": 0.0,
        "yaw": 0.0,
        "x": 0,
        "y": 0,
        "z": 0
```

```
        }
      }
    ]
}
```

Step 5: Check and Run Livox ROS Driver 2

Now, Run the respective commands to launch the ROS scripts to publish the messages. In our case

$ roslaunch livox_ros_driver2 msg_MID360.launch

This command starts publishing the custom Sensor msgs of pointclouds from lidar tot the node named livox/lidar.

- **Mid360_localization:**
  The GitHub repository provides a framework for LiDAR-based localization using the Livox MID360 sensor. It offers three operational modes:
  1. **FAST-LIO-only** [5] **Mode**: This mode utilizes the FAST-LIO [5] (Fast Lightweight and Optimal LiDAR Odometry) algorithm for real-time LiDAR-Inertial Odometry without backend optimization.
  2. **Online SLAM** [4]**Mode**: Incorporates loop closure mechanisms to build maps in real-time, enhancing localization accuracy by correcting drift over time.
  3. **Localization-only Mode**: Employs a pre-built map for localization, allowing the system to determine its position within an existing environment map.

FAST-LIO (Fast Lightweight and Optimal LiDAR-Inertial Odometry):

FAST-LIO [5] is a high-performance LiDAR-Inertial Odometry algorithm optimized for real-time applications. It fuses LiDAR and IMU data to estimate the system's trajectory with high precision and efficiency. Designed for lightweight computation, FAST-LIO [5] enables seamless operation in resource-constrained environments such as drones and mobile robots.

Key features:

1. **Real-time Processing:** Processes LiDAR point clouds and IMU data in milliseconds, ensuring low latency.
2. **Tightly Coupled Fusion:** Integrates LiDAR and IMU data at a fundamental level, improving robustness in dynamic and feature-sparse environments.
3. **Drift Minimization:** Maintains accurate odometry without requiring loop closure, making it suitable for scenarios where global consistency is not critical.

FAST-LIO's [5] speed and accuracy make it a popular choice for LiDAR-based localization, especially in environments where computational resources are limited.

SLAM (Simultaneous Localization and Mapping):

SLAM [4] is a computational method used by autonomous systems to construct a map of an unknown environment while simultaneously tracking the system's position within that map. It integrates data from various sensors, such as LiDAR, cameras, and IMU (Inertial Measurement Units), to achieve precise localization and mapping. SLAM is critical in robotics, augmented reality, and autonomous vehicles, enabling systems to navigate and understand complex environments.

Key SLAM components:

1. **Localization:** Determining the current position and orientation of the system.
2. **Mapping:** Building a representation of the environment using sensor data.
3. **Loop Closure:** Correcting positional drift by identifying previously visited locations to refine the map and trajectory.

SLAM operates in real-time, balancing computational efficiency and accuracy, with algorithms tailored to specific hardware and environmental conditions.

Steps to run the SLAM algorithms

Step 1: Cloning the GitHub repository: (Link)

After cloning the repository. There are 3 algorithms that can be run and tested. They are FAST-LIO [5], Online SLAM and Localization.

Step 2: Run the launch Files from the packages

To start the mapping of environment. Run the following commands. They atrr

$ roslaunch livox_ros_driver2 msg_MID360.launch (If this is not already running. Launch it so that is instantiated

$ roslaunch fast_lio_sam_qn run.launch lidar:=livox

These commands start the mapping of the environment and shows the visualization in RVIZ. On config file and launch file we can change the RVIZ config to have different visualizations.

Once mapping we can terminate the scripts, the respective scanned maps are saved as PDC scans and the ros bags are saved as result.bag. Chack the github for the location of the save files

Step 3: Localization

To run the localization algorithm, we have to source the recorded ros bag file's path in the config file. Refer to GitHub for the process.

After souring the path. Run the following command to launch the localization

$ roslaunch fast_lio_localization_qn run.launch lidar:=livox

Results:

After running all the above steps. We obtain the map of the environment based on our filtering and so on. The below images are obtained while mapping the environment.

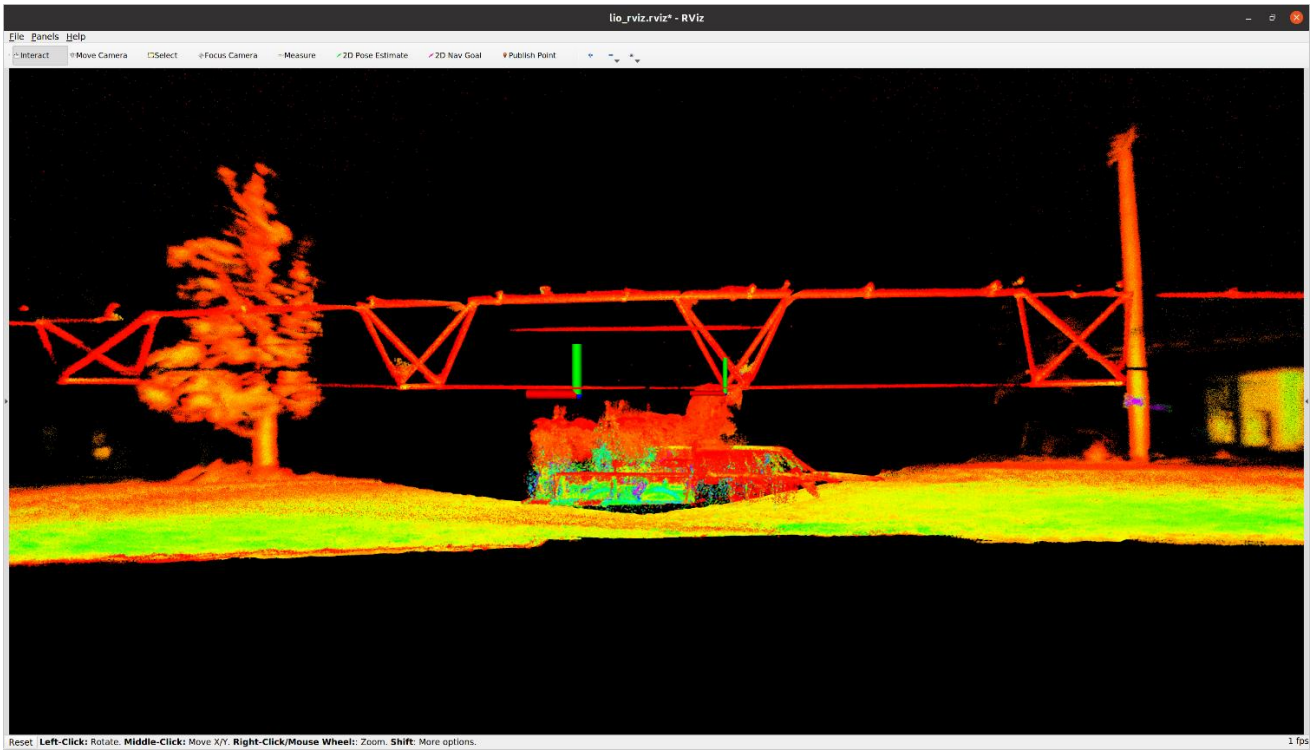


Figure 4.1. *shows the Rviz visualization while mapping the environment*

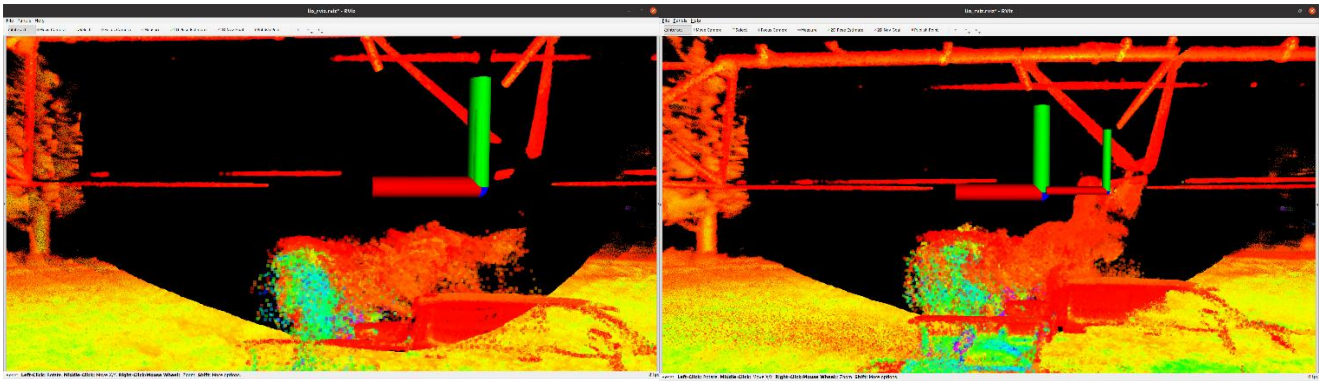


Figure 4.2. *shows the Rviz visualization of the robot's pose and localization from initial position while mapping the environment*

After Mapping the whole field. The saved result(pcd) is visualized in a tool called cloud compare to analyze and measure the distance between the robot and trusses. The images are attached below

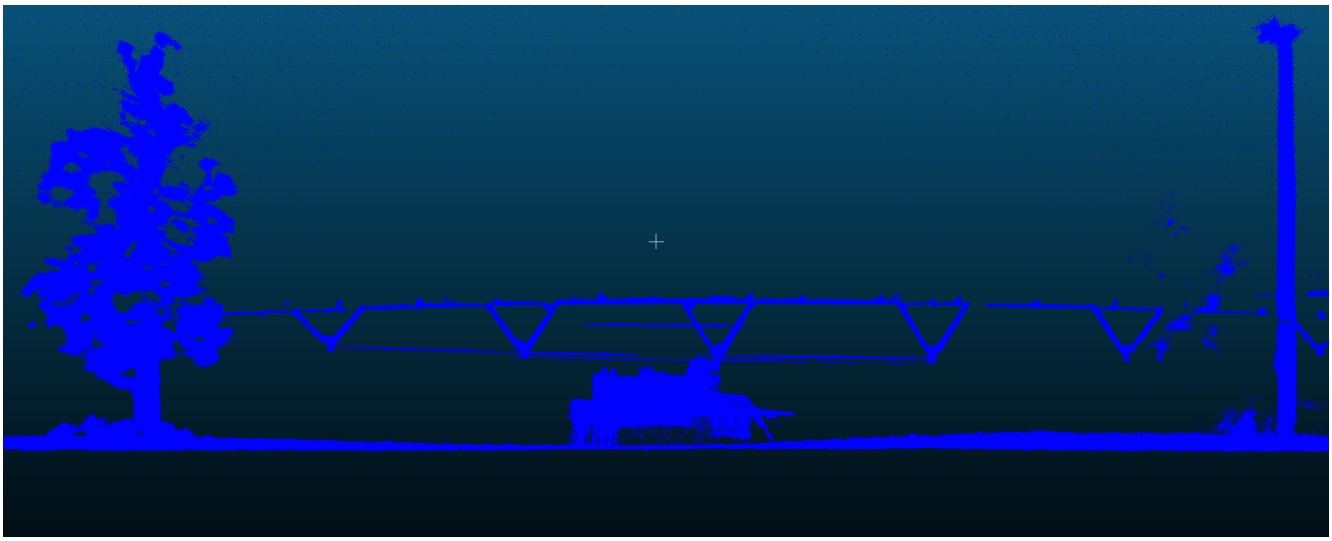


Figure 4.3. *shows the Map of the environment in front view from Cloud Compare tool*

Experiments and its results:

We conducted experiments to measure the distance between the robot and truss pockets, which can be validated using the generated map. This approach aids in accurately detecting trusses and enhancing the robot's localization precision within the mapped environment. During each iteration, the robot was positioned stationary at various points along the truss rod, maintaining different distances from the truss pocket. The actual distance was measured manually, while the sensor's odometry data was collected from the /odom node. The collected data was cross-verified and analyzed to calculate errors, along with metrics such as the mean error and RMSE, providing insights into the system's accuracy and performance.

S.No	Distance from Truss Pocket (cm)		Error
	Actual Distance (Ground Truth)	Sensor Odometry Data	
1	0	-2.7	2.7
2	23.01875	20.968	2.05075
3	46.99	45.93	1.06
4	75.565	73.864	1.701
5	108.585	106.565	2.02
		Mean:	1.906

Table 1.1 Illustrates the distance between robots and truss pockets and error

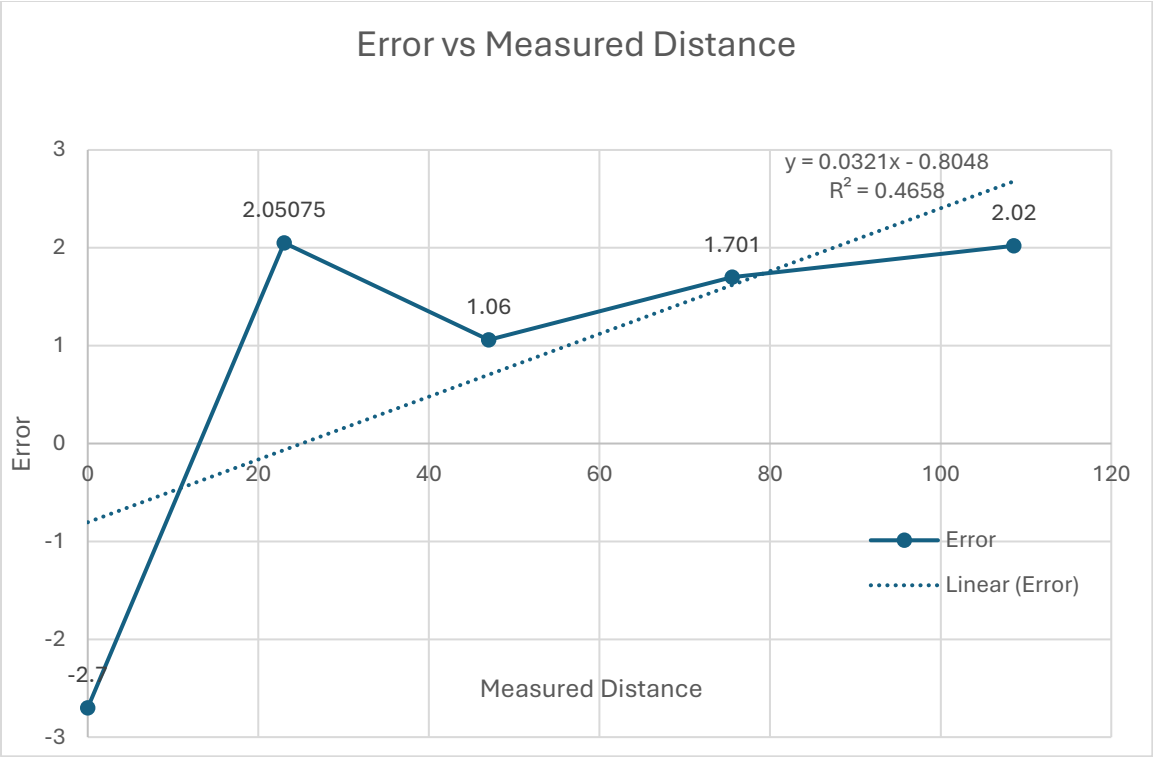


Figure 4.4. *shows the plot between error and measured distance along with the R-square value and linear trendline* [5]

Error Metrics and results:

- ***Root Mean Square Error (RMSE):***

  Measures the square root of the average squared differences between actual and predicted values, emphasizing larger errors.

$$\text{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}$$

  Where:

  - y_i = Actual Distance (Ground Truth)
  - $\hat{y}_i$ = Sensor Odometry Data y y-hat
  - n = Total number of observations

  Therefore, the RMSE = 1.979

- ***Mean Absolute Error (MAE):***

  Computes the average of absolute differences between actual and predicted values.

$$MAE = \frac{\sum_{i=1}^{n} |x_i - \hat{x}_i|}{n}$$

Note: For distances close to zero, ensure no division by zero occurs.

Therefore, MAE = 1.9063

- ***Mean Absolute Percentage Error (MAPE):***

  Calculates the average percentage error, relative to the actual values.

$$MAPE = \frac{\sum_{i=1}^{n} \frac{|x_i - \hat{x}_i|}{x_i} \times 100}{n}$$

  Therefore, MAPE = 3.052%

- ***Lin's Concordance Correlation Coefficient (CCC):***

$$\rho_c = \frac{2\rho\sigma_y\sigma_{\hat{y}}}{\sigma_y^2 + \sigma_{\hat{y}}^2 + (\mu_y - \mu_{\hat{y}})^2}$$

  Where:

  - ρ = Pearson correlation coefficient between y_i and $\hat{y}_i$
  - σy, σŷ = Standard deviations of y_i and $\hat{y}_i$
  - μy, μŷ = Means of y_i and $\hat{y}_i$

  Therefore $\rho_c = 0.999$

Future Works:

In future work, we aim to automate the existing gantry robot setup [2] used in center pivot irrigation systems [1] by implementing a Finite State Machine (FSM) for autonomous operation. Building on the successful integration of manual control and LiDAR systems, we plan to utilize LiDAR data for real-time truss detection. Advanced techniques such as RANSAC and other point cloud processing libraries will be explored to enhance detection accuracy and localization within the mapped environment. Additionally, we will focus on optimizing the SLAM algorithm to further improve the system's overall performance and reliability.

File Attachments:

CODE:

1. **Manual Control:** (Rivulet_Main) – Code file that contains FSM implementation for manual control
2. **Motion Planning:** (Path_Planning) – Code file that contains motion planning simulation

3. **ROS Packages:** **(Livox ROS Driver 2 and MID360 Mapping, Localization)** – The required ROS packages to map the environment and localize.

Conclusions:

In conclusion, this course has equipped me with essential skills and knowledge in real-time robotic control systems and also enhanced my critical thinking and problem-solving abilities by addressing unique challenges in Agriculture Field conditions.

References and Citations:

[1] Splinter, William E. "Center-pivot irrigation." *Scientific American* 234.6 (1976): 90-99

[2] Ahlers, Corey. "Development Of an Adaptive Control Mechanism to Enable Automated Irrigation on Gantry Robots." (2022).

[3] Shafi, Uferah, et al. "Precision agriculture techniques and practices: From considerations to applications." *Sensors* 19.17 (2019): 3796.

[4] G. Q. Huang, A. B. Rad and Y. K. Wong, "Online SLAM in dynamic environments," *ICAR '05. Proceedings., 12th International Conference on Advanced Robotics, 2005.*, Seattle, WA, USA, 2005, pp. 262-267, doi: 10.1109/ICAR.2005.1507422.

[5] Xu, W., & Zhang, F. (2021). Fast-lio: A fast, robust lidar-inertial odometry package by tightly coupled iterated kalman filter. *IEEE Robotics and Automation Letters*, *6*(2), 3317-3324.