CS 467 - Fall 2021

Greenhouse HyperRail Research: Midpoint Report

Project Owner/Mentors

Jorian Bruslind
Chet Udell

Team Members

Kelley Reynolds, Eric Sanchez, Lando Shepherd, Peter Wright

Introduction

We are working on building the interface, data collection, and motion control modules for the HyperRail project, a cross-disciplinary project being conducted by the Openly Published Environmental Sensing (OPEnS) Lab, The OSU Department of Electrical Engineering and Computer Science, and the OSU Department of Biological and Ecological Engineering. The HyperRail is a modular linear motion system designed for low-cost, precision data collection in a variety of plant growing environments.

The HyperRail is an aluminum-framed linear motion device with a gantry-mounted end effector. The gantry moves along the base frame in the Y-direction and the end effector moves along the upper frame in the X-direction. The system is powered by three stepper motors controlled by an ESP-32 microcontroller connected via USB to the control computer [1]. Environmental data will be collected using an SHT-30 temperature/humidity sensor. Light data collection may be added in a later iteration of the system. Images of the test plots will be collected by a MicaSense Red Edge MX multi-spectral camera. The system is controlled by an Nvidia Jetson Nano single board computer running Robot Operating System and hosting the web user interface for the user to input destinations, specify data collection actions, report on machine status, and review prior collected data. Data which has been collected by the system is stored in a SQLite database on the Jetson Nano and stores the data collection events as a series of waypoints represented by an X-Y coordinate, a type of data to be collected, a timestamp, and a unique identifier. A series of waypoints collected together constitutes a program. The user may create a program which they want to run for data collection, run a previously saved program to collect data from the same locations in the test plot over time, or review historical data.

Current Status

To date, we have been focused on on-boarding with the project objectives, the languages involved, and the concepts of ROS. During the first weeks of this term we have focused on the design of the system. At this stage, the program flow has been laid out, message formats for the services and topics have been established, and the basis for nodes has been established. The database schema has been established and in the process of being built. The user interface has been wireframed, UI development has started, and the message protocol for publishing to ROS topics from the Ruby front end has been tested. The framework for the nodes which will handle motion control and interacting with the sensor nodes has been established and the logic for triggering events through services has been started.

Design Discussion

User Interface

Ultimately, the user will send instructions to the HyperRail through a web user interface. A rough wireframe of the UI was created. It includes three main pages. The first page allows the user to move the end effector to a distinct x and y coordinate. It also will show current sensor (temp, co2, etc) data. (also, this might show the current NDVI data from the pictures taken) This display data page enables the user to enter x and y coordinates to move the end effector. It is also used for testing purposes. For the second page, (Path planning) the user may designate a series of waypoints, each with an associated action. Actions at each waypoint can be to collect an image and/or a set of environmental readings. The user may also send individual instructions to the HyperRail to manually move the end effector about the test plot and collect data. When the user enters a series of waypoints, they are saved in a SQLite database stored on the host computer for reference when the program is being executed and data is being collected. The third page is a current display of the location of the end effector. As of right now, simple data such as x and y position of the end effector will be displayed.

ROS Components

The HyperRail control system is built on ROS, a publish/subscribe programing paradigm which consists of a set of concurrently running nodes. In a ROS program, each node is responsible for a specific action. This may be a logical operation on the host computer or interacting with a microcontroller. ROS provides two primary interprocess communication channels for nodes. Services, which allow for one-to-one communication between two nodes in a request/response format and topics which allow for one-to-many communication between multiple nodes in a publish/subscribe format. The HyperRail system will include nodes for data collection, motion control, and location tracking.

Motion control will be handled by a program node, which receives requests from the user interface containing the unique identifier of the program that the user has selected to run. It will then query the database for the waypoints which constitute the program entered by the user. The program node will send the destination to the motion node which will split up the distance between the end effector's current location and the waypoint destination to generate a series of G-codes and publish them to the G-code topic. The G-code topic is subscribed to by the ESP32controller node which ultimately sends the series of codes over a serial connection to the ESP-32 microcontroller which controls the stepper motors which move the end effector along the tracks. As the end effector is moving, the location node will poll the location of the end effector until the destination is reached. When the destination is reached, the motion node will notify the program node that the destination has been reached. The program node will then send a request to the sensor node or camera node to collect a set of environmental readings or an image at the designated waypoint. In the case of the sensor node, once the data has been collected, it will respond to the program node with the environmental readings for the location to be stored by the program node. In the case of the camera node, the image will be stored to the file system by the camera node directly. Once all responses to data collection requests have been received by the program node, the next waypoint is sent to the motion node. This process

will repeat until each waypoint in the program has been visited and the specified data has been collected.

Environmental data will be collected by the sensor node which interacts with an SHT-30 temperature/humidity sensor. This sensor will be connected to the Jetson GPIO and readings collected when the node receives a request from the program node. The sensor service request consists of the unique id for the datapoint and the type of reading to be collected. The response consists of the unique id for the datapoint, a status code, and the results of the readings. When the program node sends a request through the service, the handler function will execute the data collection and send the response. The program node will be responsible for storing the datapoint in the database.

Images will be collected by the camera node which interacts with the MicaSense camera. Similar to the sensor node, the camera node service passes a request from the program node to the camera node with an instruction to collect an image. When the image is collected, the camera node sends a response containing the identifier for the data point, a status code, and the file path which is stored in the database by the program node.

Work to be completed

Over the remainder of the term, we have to fill in the skeleton of the program we have started. In the program node, we still need to write the database queries to retrieve programs from the database and store data collected. The motion and location nodes need to have their services and topics revised to fit a modification to the control flow. The motion node will need to have functionality built to split up the distance between the end effector's current location and the destination into short segments. The sensor and camera nodes still need to be built. Before this can happen, we will need to discuss the sensor types and connection with the project owners. Once we have this information, we can begin setting up the functions which will collect and return that data to the program node. We still need to get a better understanding of how to interact with the camera in ROS. To do this, we will download sample data from the Micasense website, create experimental ROS nodes that accept the same data type and run some tests. We will also need to understand how to store files to the file system for retrieval later.

The web UI will need to have additional work done to facilitate program creation, display real time machine status, and display/download images take from the Micasense camera. There will also need to be some additional work to send one-off Gcode to the ROS node responsible for sending Gcode to the ESP32 controller. Once it is determined how to stitch images that the Micasense camera generates, there will need to be a script program which can generate, stitch, and use those images in the program editing interface on an ongoing basis. Ideally this entire code path will be tested directly on the Jetson Nano as well as this is a non-standard ARM architecture and there may be additional challenges ensuring all software runs properly on that architecture.

We still need to investigate the Micasense camera in more depth. We need to determine which data we need to retrieve from the micasense and how we can process the images. Also, the photo-stitching needs has not yet started implementation. Both of these features seem to have a scope that is still undefined.

Level of Effort

The complexity of this project warrants a high level of research prior to any experimentation work we have put forth thus far. Initially, it was necessary to learn about the ROS library general as it was new to most if not all of us. We then completed a few tutorials in order to solidify our understanding of the way this library works before we could apply it to the needs of the HyperRail project. Another challenge involved not having the ability to experiment with the hardware as each member of the CS capstone group is remote. With our time spent researching, conducting experiments and having frequent meetings with the project owners to ensure we are on the right path, we thoroughly believe that each member of the group has met or exceeded the 10 hour/ week level of effort.

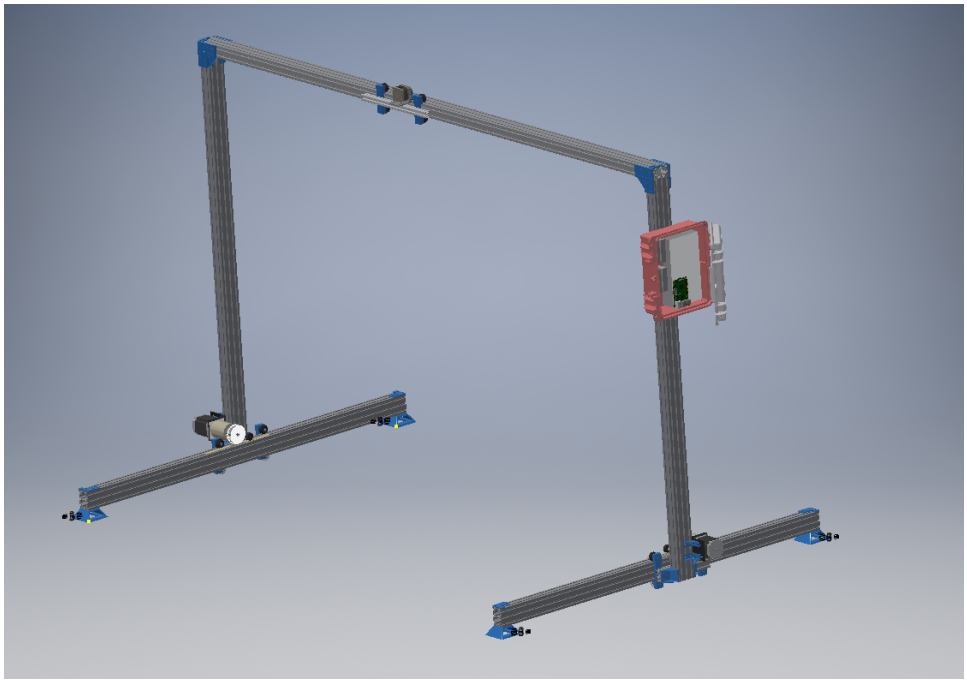

Figure 1 - Solid Works of Gantry [2]

Diagrams and Research Code

Attached in zipfile:
flowchart of nodes - node_flowchart.pdf
wireframe of UI - ui_wireframe.pdf
UML diagram of database - database_schema.pdf

Links:
Testing websocket that connects Interface with ROS nodes - link
Javascript ROSBridge connection example - link
Ruby on Rails UI - link

ROS Nodes - LocationTracker, PathPlanner, tester nodes. link

References:
[1] J. M. Lopez Alcala, M. Haagsma, C. J. Udell, and J. S. Selker, “HyperRail: Modular, 3D printed, 1–100 m, programmable, and low-cost linear motion control system for imaging and Sensor Suites,” HardwareX, vol. 6, 2019.
[2] J. Bruslind, https://github.com/Jbruslind/HyperRail/tree/main/Inventor%20Files