Autonomous Mapping and Asset Localization using Wheel-Based Rovers

Virginia Commonwealth University
Engineering Capstone Team CS-24-311
Final Report

For the

Virginia Commonwealth University Department of Computer Science

Luke Unterman
Imanol Murillo
Mallika Lakshminarayan
Tahshon Holmes

Faculty Advisor: Dr Tamer Nadeem

Sections:

Introduction:

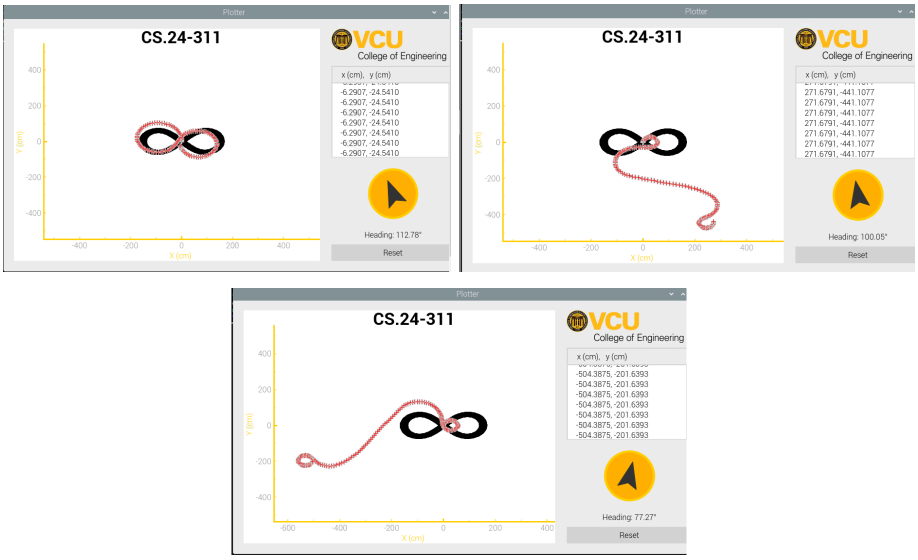
Tracking high-value assets remains a significant challenge across various industries, necessitating solutions that offer precise tracking capabilities without relying on external GPS or human intervention. Existing methods often struggle with accuracy, leading to inefficient resource utilization. Industries such as healthcare and manufacturing, with assets like wheelchairs, manufacturing equipment, and portable medical devices, require real-time tracking to enhance operational efficiency and asset utilization.

Our project presents an updated approach using rotary optical encoders and a boundary detection algorithm for accurate asset localization. By replacing gyroscope sensors with rotary optical encoders, our system achieves enhanced precision in tracking asset movement by using the rotation of the wheels rather than the motion of the sensor to calculate distance. Additionally, the implementation of a boundary detection algorithm enables the system to operate effectively in many indoor environments. The product interprets rotational velocity readings from two rotary optical encoders placed on each wheel and calculates distance traveled and relative location using trigonometric transformations and heading calculations. The boundary detection algorithm utilizes collisions with obstacles like walls to map a boundary. From this information, the system can provide the asset's location within its environment.

The goal of replacing the IMUs with a rotary optical encoder is to reduce sensor drift experienced by the system. Sensor drift is the slow changing of the output signal of the sensor independent of the measured property over time that causes the sensor reading to deviate from the ground truth. Gyroscopic drift is a common problem with gyroscopic sensors such as the IMU.

Current goal of sensor comparison:

Our project is a continuation of a previous Capstone project entitled "Real Time Indoor Wheel Based Asset Localization System." While this project was largely effective at producing an accurate wheel-based asset localization system, its system's reliance on Inertial Measurement Units (IMUs) resulted in a significant source of error when localizing assets over an extended period of time. More specifically, the previous Capstone project utilized MetaMotionS IMU sensors, which captured the angular velocity of wheel-based rovers via MEMS (microelectromechanical systems) 3-axis gyroscopes. While varying factors affect the degree with which MEMS gyroscopes experience error, all MEMS gyroscopes exhibit error depending on thermo-mechanical white noise, flicker noise, calibration errors, and fluctuations in temperature.



The above three visualizations of wheelchair movement in a figure-eight motion capture the extent to which MEMS-related error affected the previous Capstone project's localization accuracy over time. In the first visualization plot, the series of red points (tracking the wheelchair's movements) seamlessly overlaps with the black ground truth line, signifying a high-level of localization accuracy. However, after leaving the MetaMotionS IMUs initialized for 5 minutes and 10 minutes respectively, the accuracy of the previous system's localization significantly decreased as a result of the accumulation of MEMS-based gyro error. In real-world asset-localization scenarios, it is imperative for asset-tracking systems to maintain a high level of accuracy when evaluating the position of an asset.

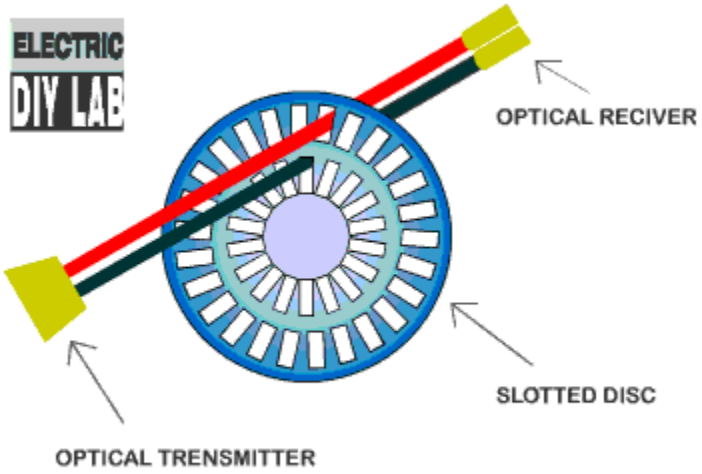
For this reason, one of the primary objectives of our Capstone project was to effectively integrate an improved sensor with minimal gyro error into the existing system developed by the previous Capstone team.

Selecting an Improved Sensor:

A rotary encoder is an obvious candidate as a replacement for the MetaMotionS IMU sensors. Rotary encoders are sensors that convert the angular position or motion of a shaft or axle into an analog or digital signal. Because they can measure the rotation of a wheel or axis directly and do not derive their measurements from vibrations (like IMUs), they are largely unaffected by varying environmental conditions like temperature and white noise. However, while it was easy to decide to use rotary encoders instead of IMUs to measure angular motion, actually deciding which type of rotary encoder to use (incremental vs absolute) and which type of rotary encoder technology to use (optical, magnetic) was somewhat difficult.

<u>Determining Which Type of Rotary Encoder to Use:</u>

To determine which type of rotary encoder to use, we conducted preliminary research into the differences between different types of rotary encoders. For example, incremental rotary encoders differ from absolute rotary encoders fundamentally in terms of how they provide angular velocity and position information. While an absolute encoder can provide information regarding the exact position of a shaft in rotation at any given time, an incremental encoder only reports relative changes in position. Furthermore, magnetic rotary encoders measure angular rotation by detecting changes in magnetic flux fields, and are typically when a high tolerance of shock and vibration are required. Conversely, optical rotary encoders typically offer higher resolution and higher accuracy in their measurements, and measure angular rotation through the passing of an LED light through a slotted metal disk to an optical receiver.



Source: https://electricdiylab.com/how-to-connect-optical-rotary-encoder-with-arduino/

As the main priority for our project was to generate more accurate (and resistant to gyro drift) measurements while maintaining a relatively low budget, we decided to use an incremental optical rotary encoder. We did not anticipate the need for a particularly robust rotary encoder with respect to shock and vibration, and we decided that using a much more expensive absolute optical rotary encoder would be largely unnecessary.

Selecting a Wireless Optical Rotary Encoder

To be able to seamlessly integrate an optical rotary encoder into the previous Capstone's wheelchair-based asset localization system, we determined that it would be necessary to order an optical rotary encoder which could communicate with our Raspberry Pi wirelessly via Bluetooth. This significantly narrowed down our options, but we found two different wireless optical rotary

encoder solutions from GoDirect and PASCO which satisfied our project requirements. The table below details the various differences between Vernier's Go Direct® Rotary Motion Sensor and PASCO's Wireless Rotary Motion Sensor.

	Go Direct	PASCO
Price	189	199
Resolution	1° or 0.25°	0.18°
Max Speed	30 rev/s or 7.5 rev/s	30 rev/s
3-step pulley	10 mm, 29 mm and 48 mm groove diameter, 55 mm with O ring in groove	10, 29, and 48 mm diameter
Connections	Bluetooth or USB	Bluetooth or USB
Optical encoder	bidirectional, quadrature encoder	2000 divisions/rev, bidirectional
Battery	Rechargeable	Rechargeable, LiPo
Data Collection Software	Vernier Graphical Analysis	PASCO Capstone or SPARKvue
Software Price	Free for Basic, $149 for a one year site license	SPARKvue: $99; PASCO: $149
Manuals	Go Direct Sensor Manual	PASCO Sensor Manual

Ultimately, while the PASCO Wireless Rotary Motion Sensor offered higher resolution measurements than the GoDirect Rotary Motion Sensor, we opted to use GoDirect's sensor as a result of its more robust Python support. The GoDirect library is frequently updated, and it is incredibly easy to connect and receive measurements from its supported sensors.

Materials Used

Expenses	# of Units	Unit Rate ($)	Costs ($)
Built Sensor:			
Go Direct Rotary Optical Encoder	3	189	567
Pasco Wireless Rotary Motion Sensor	1	199	199
Raspberry Pi + Accessories:			
Raspberry Pi	2	140	280
Keyboard	1	20	20
Mouse	1	10	10
Misc Costs	1	100	100
Assembled Sensor Components:			

Female to Female Jumper Dupont Wire Cables	1	5	5
Male to Female Jumper Dupont Wire Cables	1	5	5
HC-05 Bluetooth Module	2	14	28
USB to TTL Adapter	1	8	8
Building Materials:			
Zip Ties	1	5	5
C-Clamps	2	16	32
Foam	2	12	24
Misc Costs	1	100	100

Creating schematic for Holding GoDirect sensor on wheelchair

To fit the sensor on the wheelchair to reproduce the results from last year's team, we had to create a fitting for the new

- Initial schematic to create fitting to hold sensor
- 

- Use of PVC pipe/fitting and velcro to initially hold GoDirect sensors

- 
- Using string, zip ties to hold GoDirect sensors in place in an attempt to reduce mechanical slippage
- 

Testing:

As a sanity check to confirm that our proposed method of replacing the MetaMotionS IMU sensors from the previous Capstone project with optical rotary encoders from GoDirect would remove gyro drift, we conducted multiple experiments to determine how the measurements of our sensors drift over time without external movement. More specifically, through three different trials each, we left both the IMUs and the optical rotary encoders to sit completely still over a period of six hours. During this time, the IMUs and optical rotary encoder sensors outputted the angular velocity (units: deg/sec) they measured every second into different log files. After each of the six hour trials had elapsed, we read from the log files and calculated how the measurements from each of the sensors varied/deviated over time via a metric called Allan variance.

Allan variance, also known as two-sample variance, was originally devised as a measure of ascertaining the frequency stability of clocks, oscillators, and amplifiers. However, it can also be used to measure various noise sources present in MEMS gyroscopes, including angle random

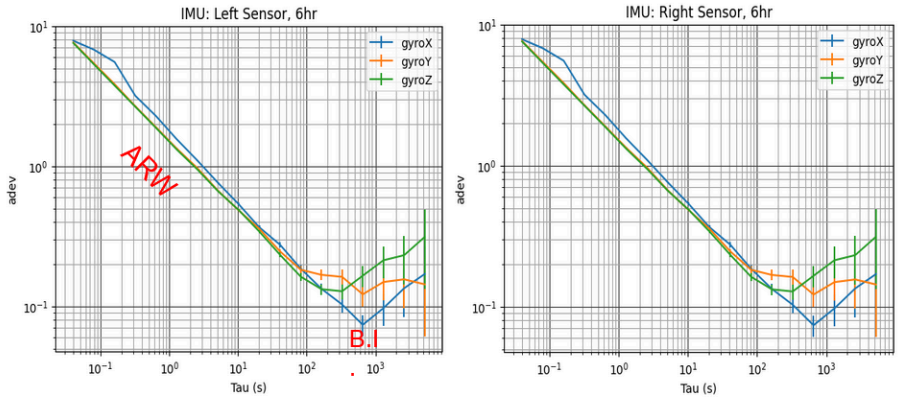
walk and bias instability. The following equation below demonstrates how Allan variance is calculated.

$$\sigma_y^2(\tau) = \left\langle \sigma_y^2(2,\tau,\tau) \right\rangle = \frac{1}{2}\left\langle (\bar{y}_{n+1} - \bar{y}_n)^2 \right\rangle = \frac{1}{2\tau^2}\left\langle (x_{n+2} - 2x_{n+1} + x_n)^2 \right\rangle$$

Thankfully, there is a Python library called **AllanTools** which can easily calculate and plot Allan **deviation** graphs, with Allan deviation being calculated by taking the square root of the Allan variance. While there a multitude of slightly varied methods of calculating Allan deviation, including Overlapping Allan deviation and Modified Allan deviation, we opted to simply calculate basic Allan deviation for our plots.

<u>IMU Allan Variance:</u>

Here is one of the plots generated from the measurements calculated by the left and right MetaMotionS IMU sensors, which were left completely still over a 6 hour period.



In order to interpret this plot, one must first understand the two most common descriptors of gyro drift/error: angle random walk and bias instability. The output of a MEMS gyroscope includes a broadband, random white noise element, and angle random walk (ARW) describes the average deviation or error of a gyroscope which occurs as a result of this noise element. As labeled on the IMU: Left Sensor plot, ARW can be visualized by Allan deviation plots as the line of constant, negative slope over time. The slope of this line can be formally calculated by dividing degrees by square seconds, but for the purposes of our experiment the ARW between the IMU and optical rotary encoders can be visually observed. Furthermore, bias instability refers to the stability of bias offset, meaning the stability of the nonzero sensor output when the input is zero. The bias instability can be identified by the lowest point on an Allan deviation plot, and generally when comparing the bias instability of multiple sensors, the lowest bias instability indicates the least inaccurate (deviating) sensor. The bias instability of the MetaMotionS IMU sensor is approximately 1e-1, which is fairly high (indicating large level of deviation in IMU measurements) for a MEMS gyroscope.

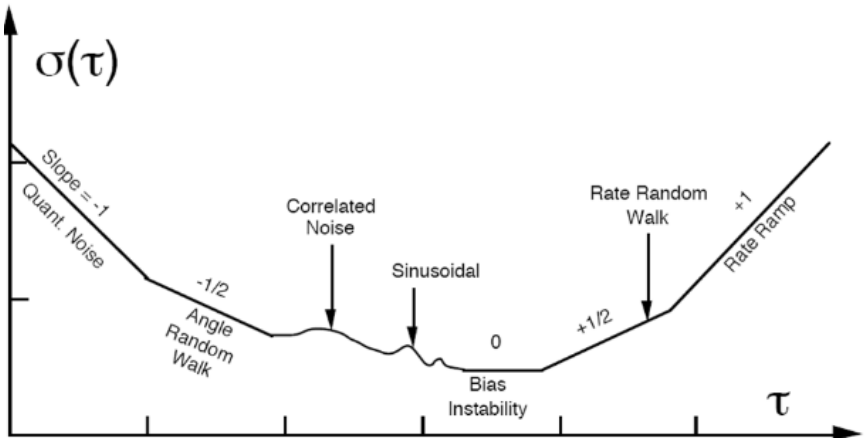
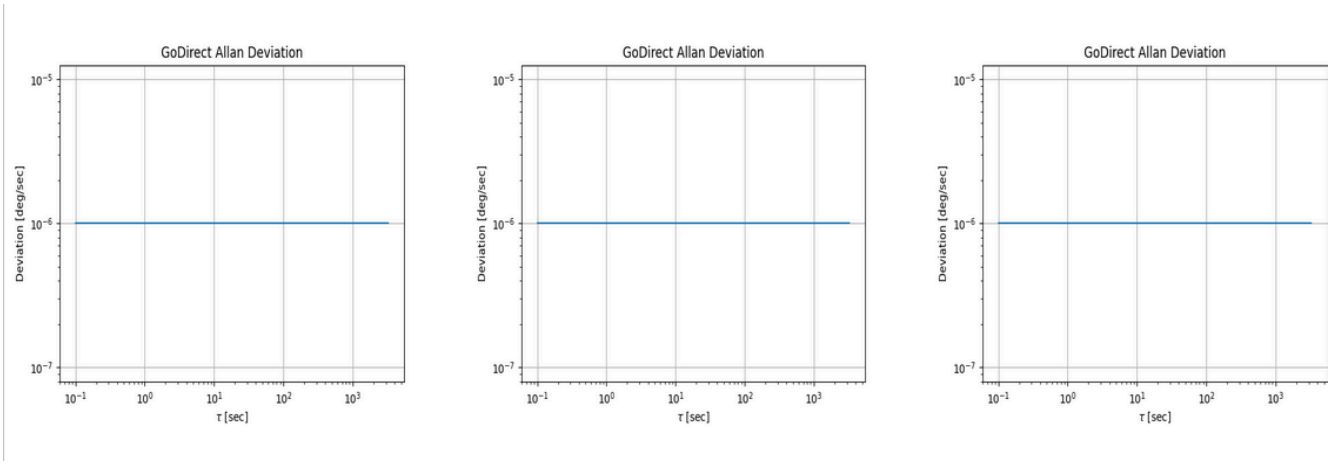


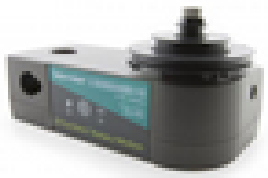
Figure: visualization of regions within Allan Variance graph (loglog plot) which demonstrate different instances of sensor drift/error

<u>GoDirect Allan Variance:</u>

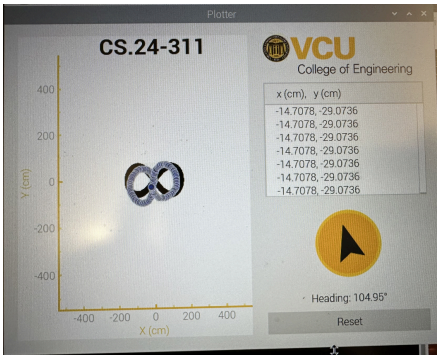


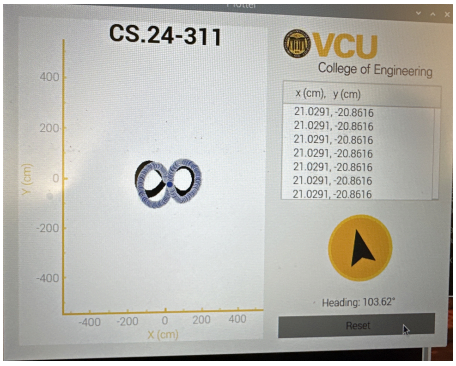
Conversely, as expected, the optical rotary encoder's measurements did not vary whatsoever during the 6 hour trial period, as its measurements are derived from physical movement of the sensor, not vibrations/other environmental conditions. In fact, because there was no variance, these plots are not entirely correct, as I had to add the deviation calculation by a very small (1e-6) constant because the number 0 cannot be plotted on a logarithmic scale. Therefore, the results of our Allan deviation testing (sanity check) demonstrate that as we suspected, the output of optical rotary encoders does not vary without external movement.

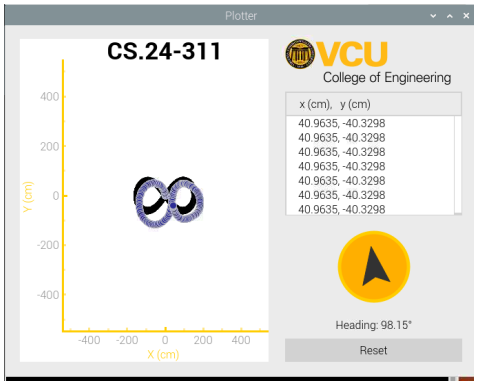
IMU vs Optical Rotary Encoder Visualization Testing:



VS.

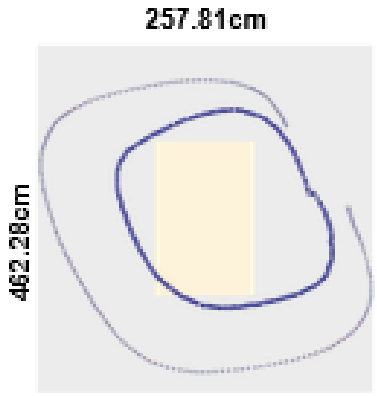






To address the difference in IMU performance vs optical rotary encoder (GDX) performance over time, the above three images represent the optical rotary encoder's visualization of the same figure-eight shape across 0, 5, and 10 minutes of initialization time respectively. Obviously, because the optical rotary encoder is no longer beholden to fluctuations in environmental factors such as temperature or white noise, it doesn't experience the same "accumulated error" effect demonstrated by the visualization of the IMUs across the same time interval.

Also, in order to test the validity of the optical rotary encoder (GDX) visualizations, we decided to conduct several tests to visually compare the performance of the IMU sensors (immediately after initialization) with the performance of our GDX sensors.

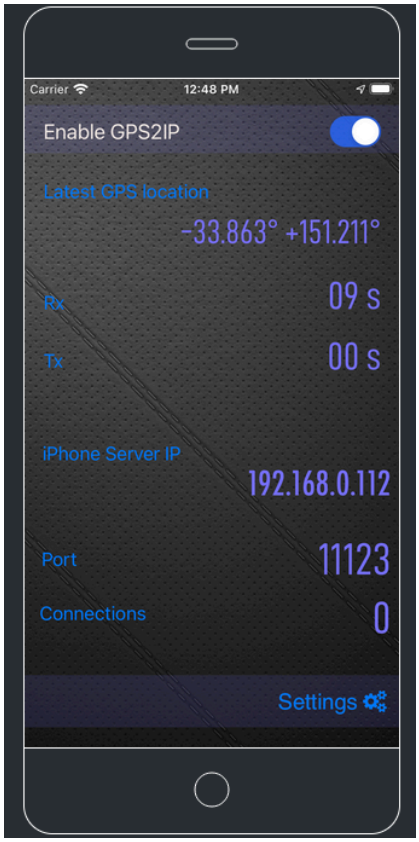


Example: Image of IMU (dotted blue) vs optical encoder (solid blue) moving around table. In this case, the IMU struggles to close the loop, unlike the optical encoder.

We first started with easy shapes like circles, triangles, squares, and straight lines. We initially started with this approach to see if the simplest shapes could be made without any adjustments before we moved on. Complex shapes were composed of figure eight (infinity) shapes, figure m shapes, and random shapes. These shapes contained several components from our easy shapes such as turns, straight lines, and start/stop locations. For the most part the mapping for these easy and complex shapes were fairly accurate. We then tested how the system would perform when it is constantly moving to see if it could continue to handle it. After a couple of turns the system would start mapping very incorrectly from the ground truth. We believe the custom fittings had a part to play in the reason why it did not map correctly. It made the shapes correctly but messed up on the direction heading. We then tested what the output would be after you go on a flat surface vs a ramp/bump area. As expected we saw a straight line with the flat test but when going over a bump we did see small deviation in our results. The ramps results were very similar to the straight line test. As predicated it was not able to detect that the wheelchair was moving up elevation.

Ground Truth Testing vs Optical Rotary Encoder:

In addition to directly comparing the visualization outputs of the IMUs and optical rotary encoder, we performed a test outdoors to evaluate the accuracy of our system's visualization. To do this, we utilized an iOS app called GPS2IP which allows you to share the GPS coordinates in latitude and longitude of your phone in real time. To replicate this, 1) install the GPS2IP iOS app on your smart device, 2) initialize a personal hotspot on your smartphone so that both the Raspberry Pi and smartphone are connected to the same network, 3) ensure that the receive_gps.py program is specified to connect to your designated IP and port, and 4) initialize the receive_gps.py app in src/app/GPS/ to begin writing GPS coordinates to a log file in real time. Note: messages are sent via RMC format, and you can configure delay between sending positions in the GPS2IP app.



While we collected GPS coordinates on the raspberry pi, the asset localization system was also running concurrently. We simply collected the log files produced from this system, and extracted the list of x, y coordinates (relative) from the log file. Then, given the GPS coordinates and relative x, y coordinates from our asset localization system, we converted the x, y coordinates into latitude and longitude coordinates. This is possible because the (x, y) coordinates generated by our system is an approximation of distance in centimeters.

Given the radius of the earth (approximately 6378 kilometers) and the equations for the conversion rates of number of kilometers per degree of longitude and latitude, these x, y values can be converted into GPS coordinates. In order of operations, this is done by 1) converting cm values into km values, where 1 cm = 1e-5 km, 2) calculating new_latitude from x coordinates, and 3) calculating new_longitude from y coordinates.

- new_latitude = starting_lat + (x / r_earth) * (180 / pi)
- new_longitude = starting_lon + (y / r_earth) * (180 / pi) / cos (starting_lat * pi/180)

Then, after saving the converted GPS coordinates and real GPS coordinates as CSVs, we can use the gmplot Python library (given an oauth token) to plot each of the paths in an interactable .html file. Interestingly, we found that as a result of the low GPS connectivity within the courtyard where we performed the outdoor visualization, our asset localization system (white) more closely mapped to the ground truth line (yellow) than the real GPS coordinates did (red).

Implementation

Integrating from IMU to GDX

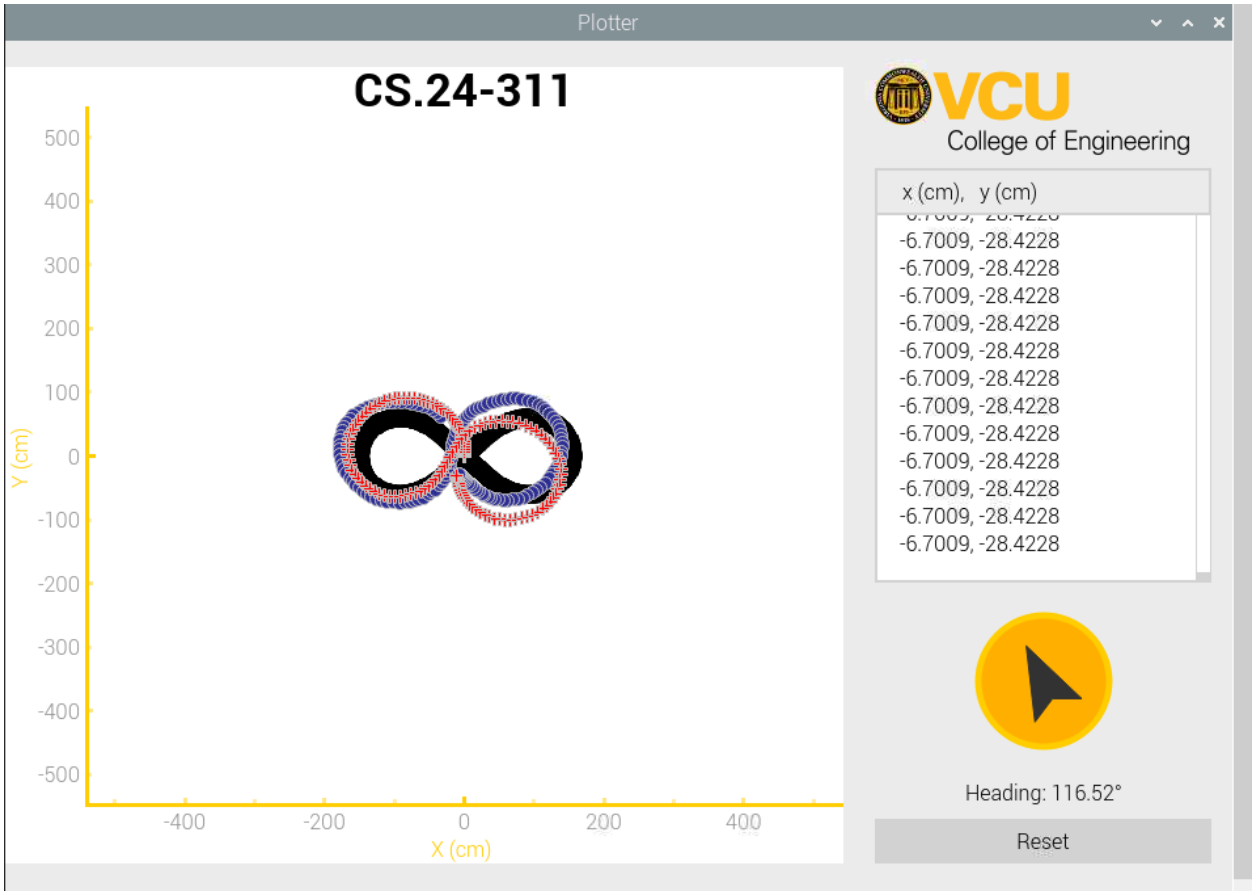
It was somewhat straightforward to alter the Aggregation layer to accept sensor readings from the GoDirect optical rotary encoder sensors instead of Metawear's MetaMotionS IMU sensors. We altered the sensor_to_raw_msg_handler.py to initiate two GoDirect sensor instances at the same time (instead of the Metawear sensors), as we mapped Threads to their start() method (which begins collecting data). We also had to alter the Aggregation layer's sensors.py class, which was slightly more difficult. The Metawear library's functionality was heavily specialized, so while we were able to still maintain the method headers/order of method operation, we changed most of the inner logic. Each of the GoDirect sensors now continuously unload messages in an infinite loop until a termination signal calls the shutdown() method. Thankfully, we were able to utilize most of the synchronization measures previously specified during the previous Capstone project.

However, during the integration process, one challenge that we faced was that we encountered discrepancies in the readings. After troubleshooting, we identified that the sensors were oriented in opposite directions, significantly impacting our project setup. To address this, for the sensor rotating counterclockwise, we inverted the readings to simulate a clockwise direction. This adjustment proved beneficial, sparing us from extensive reconfiguration of other critical components in our mapping algorithm.

Upon configuring these parameters, we implemented individual threads for each sensor to gather data from the most recent sampling. Depending on the synchronization between thread timing and sampling, there could be one or more readings queued for processing. Subsequently, we transmitted these readings to the server. Thankfully, the GDX outputs data as a list of the radians measured in the specified sensor speed setting, so we trivially could convert the radians to degrees by multiplying them by (180/pi).

It is ultimately quite fortunate that the GDX sensors closely resembled the setup of the IMU sensors. They operate via Bluetooth connectivity but also offer USB connectivity options. The GDX is governed by specific parameters dictating the frequency of sampling readings, presently set at 20 milliseconds.

Visualization of Two Sensors



Example of both IMU (red) and GoDirect optical rotary encoding visualizations on same plot

Solution:

To compare the performance of both GDX and IMU sensors, we upgraded the previous Capstone's visualization module to 1) be able to plot points from different sensors on the same graph and 2) visually differentiate between the two sensors. Instead of managing both of the sensors simultaneously (which could be a potential upgrade to the system in the future), you can run one of the programs for their respective sensor, stop that program, and then run the opposite program (of other sensor) afterwards to see both of the visualizations. The way we differentiate messages (in the same format) for each of the different sensors is by adding a "sensor_type" parameter to the system's Message Broker "Message.py" class. If the message is from an IMU, it will have sensor_type = "IMU", and if the message is from a GoDirect sensor, then the sensor_type = "GoDirect". To avoid having any messages from the opposing sensor_type being intercepted by the programs, each of the IMU and GoDirect systems have different message broker endpoints (sub and pub destinations) for the Aggregation, Logging, and Transformation layers. The two sensors share the Visualization endpoint (each have the same pub destination) for the MQTT message broker.

Distinguishing between the two, we represented GDX readings as blue 'o's and IMU readings as red '+'s. To ensure proper functionality, they needed to run independently or sequentially.

Initial Challenge:

A significant challenge we faced at the beginning of the mutual visualization task was that both sensors couldn't operate simultaneously. During our work on this issue, we observed a notable slowdown in packet reception. The visualizer would intermittently alternate between displaying IMU and GDX readings for brief periods of 3-5 seconds. Through analysis and testing, we discovered that running the sensors sequentially yielded better results. This approach allowed the runs to be displayed overlapped, facilitating comparison. Our reason for this setup stemmed from the desire to objectively test the accuracy and performance of each sensor along the same path.

Convex Hull Boundary Mapping

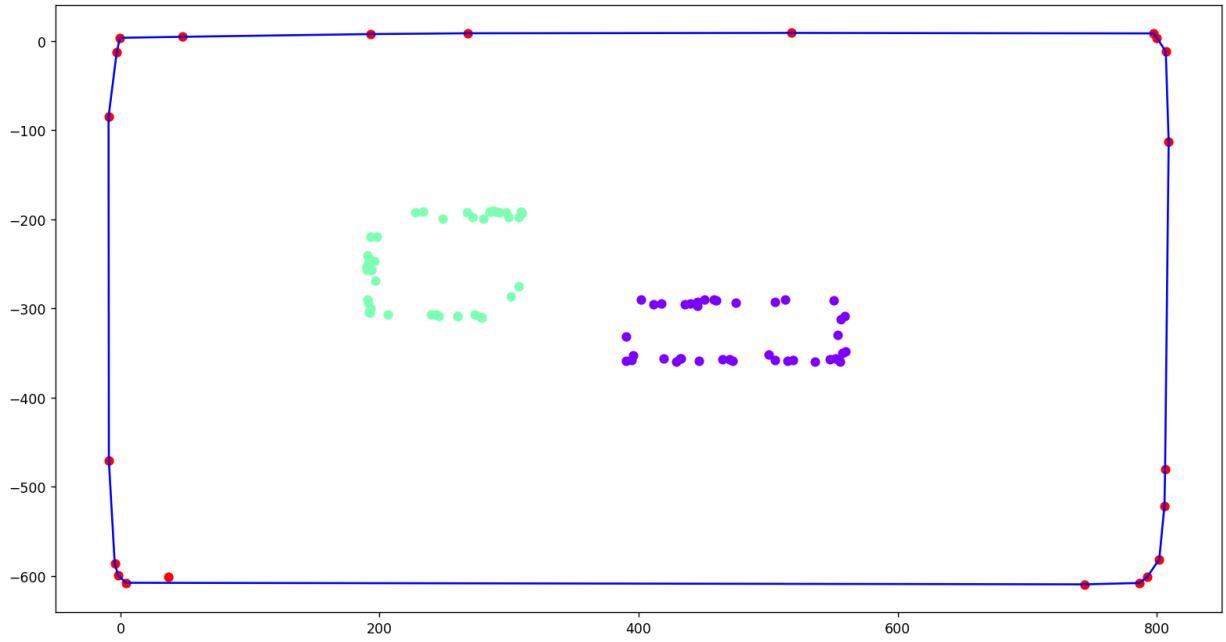
The second portion of our solution was the autonomous mapping of an environment using only the rotary optical encoder's input. Essentially we needed to use our dead reckoning algorithm and the positional data it gave us to map the two-dimensional boundary of our environment and the boundaries of the objects contained within the environment. Our boundary mapping algorithm handled these two portions separately.

Outer Boundary

To obtain the outer boundary, our algorithm computes the convex hull of the points the rover traverses throughout its journey. The formal definition of a convex hull is the set of points S in n dimensions is the intersection of all convex sets containing S. We used a Jarvis March algorithm to compute the convex hull of the points. This algorithm starts at the leftmost point and finds the point that is most counterclockwise to its position and selects this as the next point in the hull. It repeats this process with the point chosen until the first point is reached again. We draw a line through the convex hull points as our suggested outer boundary. To eliminate other points that don't indicate objects, we created a threshold and excluded those in our calculations of inner object boundaries.

Inner Object Boundaries

To distinguish points that belong to discrete objects within the environment, we used a clustering algorithm. This idea rests on the assumption that separate objects have a reasonable gap between each other. We used the DBSCAN algorithm from the scikitlearn library to identify the clusters. They are labeled with different colors as seen in the figure below. DBSCAN has two parameters: epsilon, which has to do with the radius of the region, and the minimum points required to form a region. These parameters do not always produce the expected results with different environments, so we would need to adjust this algorithm to produce consistent results.



Challenges

Our primary challenge in this project arose from our sensor attachment methods, resulting in inaccurate readings. Specifically, movements such as sharp turns or continuous turning triggered erroneous responses. Upon investigation, we determined that these discrepancies were largely attributable to user error induced by sensor attachment techniques. Notably, tracking accuracy decreased notably when maneuvering the wheelchair backward. While forward and backward motions individually presented few issues, the transition from backward to forward movement introduced deviation from the intended path. This consistent pattern led to trajectory departure. Despite aiming for straight-line movements, we consistently observed a leftward drift during both forward and backward motions. We attribute this to a combination of sensor sensitivity to all movements and coding intricacies. Additionally, we aimed to have both IMU and GDX readings simultaneously, but encountered challenges due to the Mosquitto server becoming overloaded with messages. While the server did not crash, it experienced significant slowdown.

Future Work

Some of the forthcoming tasks for this project involve refining the fitting mechanism for attaching GDX sensors to the wheel's center, optimizing our solution for enhanced movement accuracy, and finalizing boundary mapping.

Our current approach for attaching GDX sensors was designed for easy swapping between GDX and IMU sensors during testing. However, our aim is to establish a more permanent solution for the GDX sensors. We envision a setup where the GDX body remains stationary, with only the

rotary disk in motion. This permanent fitting should minimize user error and facilitate more precise movement, particularly aiding in straight-line motion and sharp turns by minimizing sensor movements.The configuration aspect of our future work will closely align with improving the sensor fitting.

Furthermore, completing the boundary mapping component of the project. Currently, our boundary mapping reads from a log file to determine the produced boundary. Ideally, we seek to eliminate the log file from the process and instead implement a system where data packets are sent to the server, allowing the boundary mapping to draw and update in real-time. This real-time updating capability will allow us to see what is the current boundary and clusters as it runs.

- Sensor Fusion
- LiDAR sensors
- 3D mapping

Appendix:

How to Use:

- Ensure that GoDirect sensors BLE LED's are both flashing red. This is the pairing mode for the GoDirect sensors, and the sensors must be in this state to avoid any issues when initializing initialize.py. Once GoDirect sensors are both flashing red, ensure that they are in close proximity to raspberry pi before initializing the program. Once they are paired, they have a large connection distance before disconnecting, but they need to be in close proximity to the raspberry pi during initialization. After both of these steps are completed, you can now run initialize.py in the "src" folder of our repository.
    - Correct folder for modern implementation of asset localization system:
        - Begins with "GoDirect"
    - Correct folder for implementation of asset localization system (IMU)
        - Begins with "CS-23…"

Installation/Dependencies:

Hardware:

- Raspberry Pi 3B/3B+/4
  - The visualization application does not require a Raspberry Pi and can be run on most OS environments. See the needed dependencies listed below and reference your OS package installation instructions.
- 16GB+ MicroSD card (32GB recommended)

Software:

- RaspberryPi Imager
- Raspberry Pi OS (32bit) with Raspberry Pi Desktop (can be downloaded within RaspberryPi Imager)

Directions

1. Flash Raspberry Pi OS to an SD card
   a. Follow the instructions in RaspberryPi Imager to download "Raspberry Pi OS (32bit)"
   b. Flash to SD card
2. Boot into Raspberry Pi OS
   a. Set username and password (Recommended name = server or client)
   b. VCU SafeNet Wireless
      i. Enable the alternative network manager (to connect to enterprise WiFi)
         1. `sudo raspi-config` → Advanced → Network Config → Network Manager
      ii. reboot
      iii. Click the Network Management Settings on the Panel, select "edit connections" and make a new WiFi connection
         1. SSID: VCU SafeNet Wireless
         2. Device: wlan0 (Mac addr)
         3. Security: WPA & WPA2 Enterprise
         4. Auth: PEAP
         5. CA cert: (select from file) /etc/ssl/certs/ca-certificates.crt
         6. PEAPv: Auto
         7. Inner Auth: MACHAPv2
   c. Allow system to do initial updates
   d. Reboot
3. Check bluetooth installation
   a. `hcitool dev`
      i. Make sure the adapter is recognized by Raspberry Pi OS
      ii. Results should look like:
      Devices:
      hci0 xx:xx:xx:xx:xx:xx

Install universal dependencies

Paho for MQTT

`python3 -m pip install paho-mqtt`

Note: You can also use `python3 -m pip install -r requirements.txt` [in the src directory of our code] to verify the list of dependencies is installed, but that includes dependencies for client as well which is a large download.

Install dependencies for Server

PyqtGraph for visualization

python3 -m pip install pyqtgraph (includes numpy upgrade, which takes a long time)

Install dependencies for Client

Dependencies and other software

- sudo apt install -y python3 python3-pip bluetooth libbluetooth-dev libudev-dev libboost-all-dev (large, 300MB+, 130+ packages)

Note: Do not upgrade using pip as this will break dependencies: https://github.com/pypa/pip/issues/5599

GPS2IP:

- Download on iOS or Android: http://www.capsicumdreams.com/gps2ip/
- Use personal hotspot to connect between Raspberry Pi and Laptop

MetaWear and Warble

- python3 -m pip install metawear

Mosquitto MQTT

https://mosquitto.org/blog/2013/01/mosquitto-debian-repository/

Note: you will likely be using Debian Bullseye as the base for your instance of Raspberry Pi OS. When adding the .list, choose the correct name.

GoDirect:

- For GDX functionality: manually install GDX library:
  - https://vernierst.github.io/godirect-examples/python/gdx/
- python3 -m pip install godirect

GMPlot:

- Receive oauth token first to allow for mapping of path onto Google Maps plot
  - https://developers.google.com/maps/documentation/datasets/oauth-token
- python3 -m pip install gmplot

For ease in testing:

- Install RealVNC Viewer on personal machine (laptop) and connect to Raspberry Pi on personal network to view Raspberry Pi screen on laptop
  - https://www.realvnc.com/en/connect/download/viewer/