



GoWrench Inc. Autonomous Driving Vehicle Project Final Report

SEP 799: Project in Systems and Technology Part 2

Master of Engineering in Systems and Technology

Faculty Lead: Dr. Moein Mehrtash

Course Instructor: Dr. Zhen Gao

Community Engagement Coordinators: Richard Allen & Salman Bawa

Community Partner Lead: Josh Lombardo Bottema

December 2024

Prepared by:

Matt Jing (400190117)

Sebastian Rivera (400091501)



Contents

Abstract.....	3
Introduction.....	3
Literature Review	4
Simulated GNSS Signal	11
Visual perception	16
Conclusion	24

Abstract

This project, undertaken in collaboration with GoWrench Inc., focuses on the development of a Level 4 autonomous driving vehicle with off-road capabilities. Leveraging the Robot Operating System (ROS) framework, the initiative integrates the SwiftNav Duro GNSS receiver for precise localization and the Oak-D camera for obstacle detection. This paper describes the implementation of a waypoint self-navigation system, detailing the use of simulated GNSS signals for pre-validation, configuration of Extended Kalman Filters (EKF) for accurate localization, and integration of global and local planners for path navigation. Additionally, a Google Maps API integration is presented, enabling the generation and visualization of navigation routes in real-world conditions. The study demonstrates a robust pathway for advancing autonomous navigation systems, validating the algorithms through simulations, and preparing for real-world deployment.

Introduction

Autonomous vehicles represent a transformative innovation in transportation, providing capabilities that range from enhanced safety and efficiency to the reduction of environmental impacts. The Level 4 autonomous driving functionality described in this project aims to allow vehicles to operate with minimal human intervention across diverse terrains, including off-road environments.

The project's current stage focuses on utilizing the SwiftNav Duro GNSS receiver for robust localization and the Oak-D camera for obstacle detection and avoidance. Both technologies are integrated under the ROS framework to develop sophisticated self-navigation algorithms. The use of Husky robots, which have proven to be a versatile platform for autonomous research, ensures that the proposed methodologies can be validated in controlled and real-world scenarios.

The autonomous driving vehicle project, in collaboration with GoWrench Inc., aims to develop a fully equipped service vehicle with off-road capabilities and Level 4 autonomous driving functionality. This project has been ongoing for over two years, involving three different groups of students who have contributed to various aspects of mechanical design and autonomous driving research. As the latest group tasked with advancing the autonomous driving functionality, our focus is on developing self-driving algorithms under the ROS framework,

specifically working with two key modules: the SwiftNav Duro GNSS receiver and the Oak-D high-resolution camera.

The SwiftNav Duro GNSS module plays a crucial role in outdoor localization. It is designed to be integrated with other IMU data to provide a reliable and accurate position estimate, enabling high-level self-navigation capabilities. On the other hand, the Oak-D camera offers a local solution for detecting and avoiding obstacles, ensuring safe navigation from point A to point B.

To validate our algorithms in real-world conditions, we are implementing them on a Husky sidewalk robot, which was also used by the previous group primarily for visual perception research in autonomous driving. This approach allows us to build on the foundation established by earlier teams while advancing the project's capabilities toward fully autonomous navigation.

The use of waypoint navigation systems in robotics has been a subject of extensive research and development. Companies such as Clearpath Robotics have demonstrated the potential of using ROS-based systems like OutdoorNav to create flexible and effective navigation solutions. In these systems, the fusion of GNSS and IMU data for localization provides precise spatial awareness.

Moreover, high-resolution camera systems and LiDAR are increasingly employed for enhanced environmental perception. These technologies collectively advance the field of autonomous driving by enabling robots to efficiently compute and execute navigation paths in complex environments. This section evaluates existing solutions and highlights their relevance to the development of the GoWrench autonomous system.

Literature Review

A. Waypoint Navigation

Various self-navigation software and platforms are available on the market, specifically designed for the ROS system. For example, Clearpath, a robotics company, recently released its own waypoint navigation tool, OutdoorNav, which utilizes the RViz interface to enable robotic self-navigation. A YouTube video demonstrates how to use this software on their latest autonomous vehicle.

(<https://www.youtube.com/watch?v=QJdJ7nAmhzY&t=469s>)

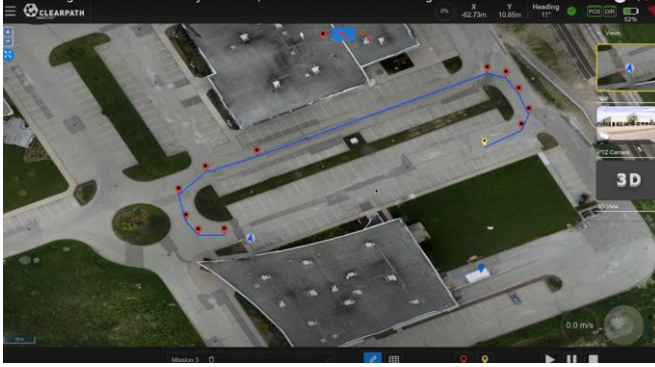


Fig 1. Video demonstration on OutdoorNav waypoint self-navigation system from clearpath

The featured vehicle is equipped with two SwiftNav Duro GNSS modules mounted on the roof for precise geometric localization. For visual perception in autonomous driving, it employs four high-resolution cameras and two high-refresh-rate LiDAR sensors to achieve accurate environmental recognition. In the video, the OutdoorNav software is used to direct the self-driving vehicle to navigate around a Starbucks and pick up a coffee from a drive-through. This process is straightforward and involves dragging and setting key waypoints along the intended route.

This example illustrates a simple yet effective approach to foundational self-driving. Using an available GNSS module, robot localization can be achieved by fusing GNSS data with IMU data to provide a reliable position estimate. To enable the robot to follow a designated route, configuring the ROS move_base node is essential. This requires setting up a global cost map and a local cost map to communicate with the robot and visualize the GNSS data in RViz.

Once the move_base node is running, it can accept navigation goals from RViz and compute a path to reach the target. After configuring these nodes, the robot's position should be accurately represented in RViz, and waypoint navigation can be achieved using the 2D Nav tool available in the RViz software.

B. Simulation Testing

Simulation testing plays a critical role in validating autonomous navigation algorithms before real-world deployment. Using static transforms and simulated sensor data ensures the system's readiness without risking physical hardware or environmental interactions. This project leverages the Jetson Orin platform to test navigation stacks, including GNSS, IMU, and path planning modules. By simulating GNSS signals and integrating fake IMU data, the system replicates real-world dynamics within a controlled environment. These

simulations allow for thorough debugging and optimization of localization algorithms, ensuring the system's robustness and reliability.

Before testing the robot in real-world conditions, it's a good practice to first validate the waypoint navigation algorithm in a simulated environment without activating any sensors on the Jetson Orin. This allows for testing the entire navigation stack—including waypoints, path planning, and sensor integration—using simulated data such as fake GNSS signals or static transforms in ROS. The key steps involved include:

- Use Static Transforms to Simulate a Stationary Robot

Instead of using real odometry or GNSS data, the robot's position can be simulated by publishing static transforms. This will make ROS treat the robot as if it's stationary at a fixed position without actual movement.

- Simulate GNSS and Other Sensor Data (i.e. IMU)

Since there is no actual satellite data being received, sensors like GNSS, IMU, and odometry need to be simulated using a fake data publisher. This mimics the inputs that would be provided by real sensors.

- Testing in a Simulated Environment

Once the fake signals are fed to the robot, the system will interpret the environment just as it would in the real world. Rviz offers a practical and visual approach to testing the navigation algorithm, ensuring everything behaves as expected before transitioning to real-world testing.

II. Implementation

A. Waypoint Self-Navigation System

To achieve waypoint self-navigation on a robot, the robot needs to first recognize its current location and where it is pointed at. For local localization, it can be achieved by fusing the signal from a lidar or camera sensor into the IMU data. Meanwhile, in a higher level of localization, we employed an Extended Kalman Filter (EKF) to fuse GNSS module sensor streams and provide a reliable estimate of the robot's position. The following scripts show the configuration of the EKF. The configuration is set up by default according to the research of the others. Some fine-tuning needs to be done when considering the environment where the robot is working and the model of the robot. Later, an associated launch file needs to be created to load the EKF configuration and start the robot localization node. The localization topic will be updated, and the `/odometry/filtered` topic will be replaced by a fused topic that represents the position of the robot.

```

13frequency: 30
14two_d_node: true # Set to true for outdoor, flat environments
15map_frame: map
16odom_frame: odom
17base_link_frame: base_link
18world_frame: map
19
20# Inputs
21odom0: odometry/filtered
22imu0: imu/data
23gps0: /piksi_rover/navsatfix_best_fix
24
25odom0_config: [false, false, false, true, true, true, false, false, false, false, false, false, false, false, false]
26imu0_config: [false, false, false, true, true, true, true, true, true, false, false, false, false, false, false]
27gps0_config: [true, true, false, false, false, false, false, false, false, false, false, false, false, false, false]
28
29# Sensor timeouts
30odom0_queue_size: 10
31imu0_queue_size: 10
32gps0_queue_size: 10
33sensor_timeout: 0.1
34
35

```

Fig 2. Basic EKF configuration to fuse GNSS module with imu data to provide localization

There are some useful tools in Rviz to help achieve waypoint navigation by simply setting up waypoints (or navigation goals) manually, the 2D Nav Goal tool is one of the tools that works in conjunction with ROS's move_base node. To enable waypoint navigation, the move_base node needs to be running. This node will accept navigation goals from RViz and compute a path to reach the goal. The ROS costmaps need to be configured firsthand to allow the robot to recognize its environment. A costmap consists of multiple layers that stack on top of each other and combine their cost values. Each layer can have its own update frequency, data source, and parameters. The most common layers are static, obstacle, and inflation layers. The static layer represents the fixed map of the environment, such as walls and furniture. The obstacle layer detects and marks dynamic obstacles, such as people and other robots, using sensor data. The inflation layer adds a buffer zone around the obstacles to account for the robot size and uncertainty. Since the obstacles are not being considered at a high level of self-navigation, the basic global and local costmap configuration files are created.

```

global_frame: odom
robot_base_frame: base_link
update_frequency: 5.0
publish_frequency: 2.0
rolling_window: true
width: 10.0
height: 10.0
resolution: 0.5

```

```

1global_costmap:
2  global_frame: map
3  robot_base_frame: base_link
4  update_frequency: 1.0
5  publish_frequency: 1.0
6  transform_tolerance: 0.5
7  static_map: false
8  rolling_window: true
9  width: 100.0 # Large enough for outdoor environments
10 height: 100.0
11 resolution: 1.0
12

```

Fig 3. Configuration for costmap

In our case, the navfn algorithm is utilized for global path planning and dwa_local_planner is used for local path planning. navfn provides a fast interpolated navigation function that

can be used to create plans for a mobile base. The planner assumes a circular robot and operates on a costmap to find a minimum cost plan from a start point to an end point in a grid. The navigation function is computed with Dijkstra's algorithm, but support for an A* heuristic may also be added in the near future. The `dwa_local_planner` package provides an implementation of the Dynamic Window Approach to local robot navigation on a plane. Given a global plan to follow and a costmap, the local planner produces velocity commands to send to a mobile base. This package supports any robot whose footprint can be represented as a convex polygon or circle, and exposes its configuration as ROS parameters that can be set in a launch file. The parameters for this planner are also dynamically reconfigurable.

```
base_global_planner: navfn/NavfnROS # Use a simple global planner
base_local_planner: dwa_local_planner/DWAPlanerROS

controller_frequency: 10.0
planner_patience: 5.0
controller_patience: 3.0

oscillation_timeout: 10.0
oscillation_distance: 0.2
```

Fig 4. Global path planner and local path planner

After creating the needed configuration files along with the launch files to activate the nodes, we can combine all the necessary launch files for waypoint navigation into a master launch file for testing.

```
GNU nano 4.8 self_navigation.launch
<!-- Launch GNSS and IMU nodes -->
<include file="$(find piksi_multi_rtk_ros)/launch/piksi_multi_rover.launch"/>

<!-- Launch robot localization (EKF) -->
<include file="$(find self_navigation)/launch/ekf_localization.launch"/>

<!-- Launch move base -->
<include file="$(find self_navigation)/launch/move_base.launch"/>
</launch>
```

Fig 5. Master launch file for testing

After launching the self-navigation launch file, the output is shown as follows. The `move_base` node is automatically shut down because the robot is tested inside with no GNSS signal received so the localization cannot be fulfilled. But all the necessary nodes are activated successfully so that the robot is ready to be tested outside or we can send a fake GNSS message with the `gps_umd` package to be tested indoors.


```

administrator@pr-a200-0985:~/catkin_ws/src/self_navigation/launch$ roslaunch self_navigation.launch
... logging to /home/administrator/.ros/log/05041eac-48b2-11e5-912d-48b02de646a8/roslaunch-cpr-a200-0985-35689.log
Checking log directory for disk usage. This may take a while.
Press Ctrl-C to interrupt.
Now checking log file disk usage. Usage is <1GB.

started roslaunch server http://192.168.0.117:44365/

SUMMARY
=====
PARAMETERS
* /ekf_localization_node/config_file: self_navigation/c...
* /move_base/base_local_planner: navfn/NavfnROS
* /move_base/base_local_planner: dwa_local_planner...
* /move_base/controller_frequency: 10.0
* /move_base/controller_patience: 3.0
* /move_base/global_costmap/global_costmap/global_frame: map
* /move_base/global_costmap/global_costmap/height: 100.0
* /move_base/global_costmap/global_costmap/publish_frequency: 1.0
* /move_base/global_costmap/global_costmap/resolution: 1.0
* /move_base/global_costmap/global_costmap/robot_base_frame: base_link
* /move_base/global_costmap/global_costmap/rolling_window: true
* /move_base/global_costmap/global_costmap/static_map: false
* /move_base/global_costmap/global_costmap/transform_tolerance: 0.5
* /move_base/global_costmap/global_costmap/update_frequency: 1.0
* /move_base/global_costmap/global_costmap/width: 100.0
* /move_base/local_costmap/global_frame: odom
* /move_base/local_costmap/height: 10.0
* /move_base/local_costmap/publish_frequency: 2.0
* /move_base/local_costmap/resolution: 0.5
* /move_base/local_costmap/robot_base_frame: base_link
* /move_base/local_costmap/rolling_window: true
* /move_base/local_costmap/update_frequency: 5.0
* /move_base/local_costmap/width: 10.0
* /move_base/oscillation_distance: 0.2
* /move_base/oscillation_timeout: 10.0
* /move_base/planner_patience: 5.0
* /piksi_rover/altitude: 565.7
* /piksi_rover/base_station_ip_for_latency_estimation: 192.168.131.1
* /piksi_rover/base_station_mode: false
* /piksi_rover/baud_rate: 230400
* /piksi_rover/broadcast_addr: 192.168.0.255
* /piksi_rover/broadcast_port: 26078
* /piksi_rover/debug_mode: false
* /piksi_rover/driver_verbose: true

* /piksi_rover/interface: tcp
* /piksi_rover/latitude_deg: 47.81
* /piksi_rover/longitude_deg: 8.51
* /piksi_rover/navsatfix_frame_id: swiftmax_link
* /piksi_rover/publish_om_and_map: false
* /piksi_rover/serial_port: /dev/ttyUSB0
* /piksi_rover/tcp_port: 192.168.0.131.31
* /piksi_rover/tcp_port: 55555
* /piksi_rover/use_gps_time: false
* /piksi_rover/var_rtk_fix: [0.0049, 0.0049, ...]
* /piksi_rover/var_rtk_float: [25.0, 25.0, 64.0]
* /piksi_rover/var_spp: [25.0, 25.0, 64.0]
* /piksi_rover/var_spp_sbas: [1.0, 1.0, 1.0]
* /roslaunch: none
* /rosversion: 1.10.0

NODES
/
  ekf_localization_node (robot_localization/ekf_localization_node)
  move_base (move_base/move_base)
  piksi_rover (piksi_multirx_ros/piksi_multirx)

ROS_MASTER_URI=http://cpr-a200-0985:11311

process[piksi_rover-1]: started with pid [35730]
process[ekf_localization_node-2]: started with pid [35731]
process[move_base-3]: started with pid [35732]
log file: /home/administrator/.ros/log/05041eac-48b2-11e5-912d-48b02de646a8/ekf_localization_node-2*.log
[WARN] [172722478.024507437]: Reason given for shutdown: [[/ekf_localization_node] Reason: new node registered with same name]
[ekf_localization_node-2] process has finished cleanly
log file: /home/administrator/.ros/log/05041eac-48b2-11e5-912d-48b02de646a8/ekf_localization_node-2*.log
[INFO] [172722480.170963]: /piksi_rover start
[INFO] [172722481.357963]: libnav version currently used: 3.4.10
[INFO] [172722481.138817]: Swift driver started in normal mode.
[INFO] [172722481.323981]: Origin ENI frame set to: 42.810000, 0.510000, 565.70
[INFO] [172722481.376378]: Origin ENI frame set by rosparam.
[INFO] [172722481.376378]: Starting device in Rover Mode
[WARN] [172722483.00951193]: Timed out waiting for transform from base_link to map to become available before running costmap, tf error: can
transform: target frame map does not exist.. canTransform returned after 0.100457 timeout was 0.1.
[WARN] [172722486.94330233]: Timed out waiting for transform from base_link to map to become available before running costmap, tf error: can
transform: target frame map does not exist.. canTransform returned after 0.101044 timeout was 0.1.
[move_base-3] killing on exit
[piksi_rover-1] killing on exit
shutting down processing monitor...
... shutting down processing monitor complete
done

```

Fig 6. ROS output after launching the master launch file

The integration of simulations into this project provides a foundation for understanding and improving navigation algorithms. Static transforms, which maintain a fixed positional reference, enable consistent testing conditions. Simulated GNSS and IMU data ensure that localization and navigation nodes can operate in a variety of scenarios. This approach minimizes dependencies on real-world environmental conditions during the development phase. Furthermore, dynamic path generation and visualization using tools like RViz and Google Maps API provide insights into the system's operational capabilities.

1. Static Transform

Since a fixed position signal is constantly sent to the robot, a launch file for simulating the robot is always at a fixed position is required to publish a static transform between the odom and base_link frames.

This file creates a static transform where the robot stays at position (0, 0, 0) in the odom frame and it is oriented with no rotation (0, 0, 0 for roll, pitch, yaw).

```

GNU nano 4.8 static_transform.launch
<launch>
<!-- Publish a static transform between odom and base_link -->
<node pkg="tf2_ros" type="static_transform_publisher" name="static_transform_publisher" args="0 0 0 0 0 odom base_link" />
</launch>

```

Fig. 7 Configuration for static transform

2. Simulate GNSS Data

To fake sending the GNSS data constantly, a fake GNSS publisher script is created to simulate GNSS data on the `/piksi/navsatfix_best_fix`.

```

GNU nano 4.8 fake_gnss_publisher.py
#!/usr/bin/env python3
import rospy
from sensor_msgs.msg import NavSatFix
from sensor_msgs.msg import NavSatStatus

def fake_gnss_publisher():
    rospy.init_node('fake_gnss_publisher', anonymous=True)
    # Publish to the specific ETHZ package topic for GNSS data
    pub = rospy.Publisher('/piksi/navsatfix_best_fix', NavSatFix, queue_size=10)
    rate = rospy.Rate(1) # 1 Hz

    while not rospy.is_shutdown():
        msg = NavSatFix()

        # Set a fake, static position (e.g., latitude and longitude for a specific location)
        msg.latitude = 51.7245 # Replace with desired test latitude
        msg.longitude = -122.4194 # Replace with desired test longitude
        msg.altitude = 10.0 # Replace with desired altitude

        # Simulate a valid GNSS fix
        msg.status.status = NavSatStatus.STATUS_FIX
        msg.status.service = NavSatStatus.SERVICE_GPS

        # Optionally set the covariance (to simulate the GPS accuracy)
        msg.position_covariance_type = NavSatFix.COVARIANCE_TYPE_APPROXIMATED
        msg.position_covariance = [10.0001, 0, 0,
                                   0, 0.0001, 0,
                                   0, 0, 0.0001]

        # Publish the fake GNSS data
        pub.publish(msg)
        rospy.loginfo('Publishing fake GNSS data: Lat: {msg.latitude}, Lon: {msg.longitude}, Alt: {msg.altitude}')
        rate.sleep()

if __name__ == '__main__':
    try:
        fake_gnss_publisher()
    except rospy.ROSInterruptException:
        pass

```

Fig. 8 Simulated GNSS signal publisher

3. Simulate IMU Data

A similar Python node can be used to simulate IMU data, which publishes fake orientation and angular velocity data on the `/imu/data` topic to measure and report the robot's specific force, angular rate, and sometimes the orientation of the robot.

With all the necessary nodes for waypoint navigation and signal publishing, the waypoint navigation system can be tested in Rviz with the static transform and fake data publishers running.

```

GNU nano 4.8 fake_imu_publisher.py
#!/usr/bin/env python3
import rospy
from sensor_msgs.msg import Imu
from geometry_msgs.msg import Quaternion
import tf.transformations

def fake_imu_publisher():
    rospy.init_node('fake_imu_publisher', anonymous=True)
    pub = rospy.Publisher('/imu/data', Imu, queue_size=10)
    rate = rospy.Rate(10) # 10 Hz

    while not rospy.is_shutdown():
        imu_msg = Imu()

        # Simulate an orientation in quaternion (no rotation)
        quaternion = tf.transformations.quaternion_from_euler(0, 0, 0)
        imu_msg.orientation = Quaternion(*quaternion)

        # Simulate no angular velocity and no linear acceleration
        imu_msg.angular_velocity.x = 0.0
        imu_msg.angular_velocity.y = 0.0
        imu_msg.angular_velocity.z = 0.0
        imu_msg.linear_acceleration.x = 0.0
        imu_msg.linear_acceleration.y = 0.0
        imu_msg.linear_acceleration.z = 0.0

        # Publish the fake IMU data
        pub.publish(imu_msg)
        rospy.loginfo('Publishing fake IMU data')
        rate.sleep()

if __name__ == '__main__':
    try:
        fake_imu_publisher()
    except rospy.ROSInterruptException:
        pass

```

Fig 9. Simulated IMU signal publisher

4. Tf2 Tools

tf2 is a ROS library that manages coordinate transforms in a distributed and dynamic way. It allows different parts of a robot's software system to communicate spatial information (e.g., the position, orientation, and movement of objects relative to one another) efficiently and in real-time. To ensure the move_base node is working, it requires the necessary transformation between the base_link and map frames. This transform typically comes from a localization system such as robot_localization, which fuses data from GNSS, IMU, and odometry to provide the map -> odom -> base_link transforms. There is a static transform publisher available in the tf2 library that publishes the map -> odom transform, which will simulate the odom frame is aligned with the map frame, allowing move_base to function.

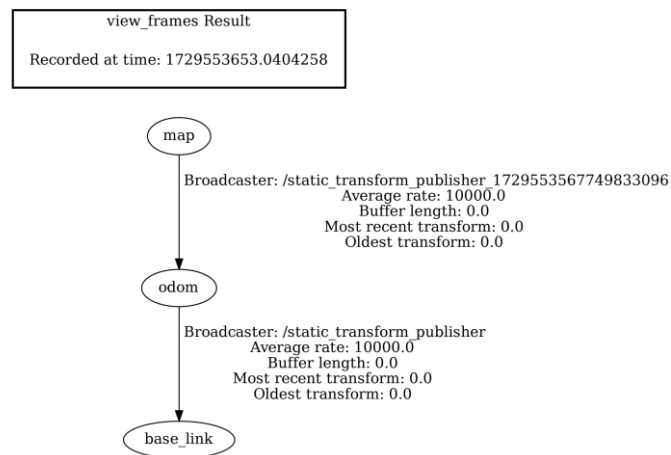


Fig. 10 Structure of a self-navigation system demonstrated by frame map

Simulated GNSS Signal

B. Simulated GNSS Coordinates Visualization

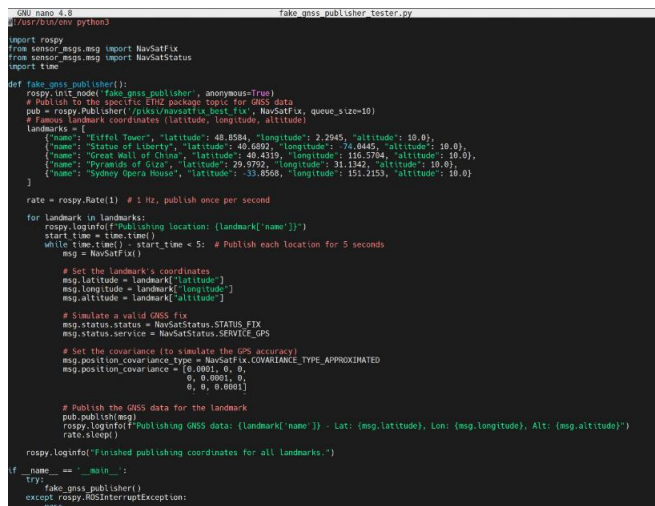
The first and foundational step in developing a waypoint navigation system involves validating the functionality of the GNSS signal publisher to ensure accurate localization. The use of simulated GNSS signals provides a controlled and flexible testing environment, crucial for early-stage algorithm validation without the dependency on real-world conditions or satellite connectivity.

A modified version of the fake_gnss_publisher node dynamically generates GNSS

coordinates over time, representing specific geographical locations. These coordinates are published to the `/piksi/navsatfix_best_fix` topic, which acts as a simulated feed for the robot's localization system. This step verifies the ability of the system to interpret and process GNSS data effectively.

For demonstration and validation purposes, the node generates coordinates corresponding to globally recognized landmarks:

- **Eiffel Tower** (Paris, France)
- **Statue of Liberty** (New York, USA)
- **Great Wall of China** (China)
- **Pyramids of Giza** (Egypt)
- **Sydney Opera House** (Sydney, Australia)



```

GNU nano 4.8                                     fake_gnss_publisher_tester.py
/usr/bin/env python3
import rospy
from sensor_msgs.msg import NavSatFix
from sensor_msgs.msg import NavSatStatus
import time

def fake_gnss_publisher():
    rospy.init_node('fake_gnss_publisher', anonymous=True)
    # Publish to the specific ENU package topic for GNSS data
    pub = rospy.Publisher('piksi/navsatfix_best_fix', NavSatFix, queue_size=10)
    # Famous landmark coordinates (latitude, longitude, altitude)
    landmarks = [
        {'name': 'Eiffel Tower', 'latitude': 48.8584, 'longitude': 2.2945, 'altitude': 10.0},
        {'name': 'Statue of Liberty', 'latitude': 40.6892, 'longitude': 74.0445, 'altitude': 10.0},
        {'name': 'Great Wall of China', 'latitude': 40.4319, 'longitude': 116.5704, 'altitude': 10.0},
        {'name': 'Pyramids of Giza', 'latitude': 29.9792, 'longitude': 31.1342, 'altitude': 10.0},
        {'name': 'Sydney Opera House', 'latitude': -33.8568, 'longitude': 151.2153, 'altitude': 10.0}
    ]

    rate = rospy.Rate(1) # 1 Hz, publish once per second

    for landmark in landmarks:
        rospy.loginfo('Publishing location: {landmark["name"]}\'')
        start_time = time.time()
        while (time.time() - start_time) < 5: # Publish each location for 5 seconds
            msg = NavSatFix()

            # Set the landmark's coordinates
            msg.latitude = landmark['latitude']
            msg.longitude = landmark['longitude']
            msg.altitude = landmark['altitude']

            # Simulate a valid GNSS fix
            msg.status.status = NavSatStatus.STATUS_FIX
            msg.status.service = NavSatStatus.SERVICE_GPS

            # Set the covariance (to simulate the GPS accuracy)
            msg.position_covariance_type = NavSatFix.COVARIANCE_TYPE_APPROXIMATED
            msg.position_covariance = [0.0001, 0, 0,
                                      0, 0.0001, 0,
                                      0, 0, 0.0001]

            # Publish the GNSS data for the landmark
            pub.publish(msg)
            rospy.loginfo('Publishing GNSS data: {landmark["name"]} - Lat: (msg.latitude), Lon: (msg.longitude), Alt: (msg.altitude)')
            rate.sleep()

    rospy.loginfo('Finished publishing coordinates for all landmarks.')

if __name__ == '__main__':
    try:
        fake_gnss_publisher()
    except rospy.ROSInterruptException:
        pass

```

Fig 11. Simulated GNSS signal publisher to stably publish coordinates of landmarks

Once the node is launched, the `/piksi/navsatfix_best_fix` topic is updated dynamically with the simulated GNSS data. These messages are recorded and converted into a Keyhole Markup Language (KML) format for visualization using Google Earth. The resulting visualizations confirm the GNSS signal publisher's ability to generate accurate and consistent coordinates.

C. Generated Path Visualization

After validating individual GNSS coordinates, the next logical step is to generate and visualize a simulated navigation path. This process assesses the system's capacity to create coherent routes between multiple waypoints and allows for the refinement of path-planning algorithms.

The `fake_gnss_publisher` node is extended to generate dynamic paths by sequentially connecting a series of simulated GNSS coordinates. These paths emulate a robot's movement from one location to another, such as navigating between predefined waypoints within a test environment or virtual map.

The generated paths are published to the `/gnss/path` topic. Each message includes metadata, such as:

- **Frame ID:** Specifies the reference coordinate system (e.g., map frame).
- **Timestamps:** Records the time at which each coordinate is generated, aiding in playback and analysis.
- **Path Segments:** Defines the trajectory between waypoints.

```
path = Path()
path.header.frame_id = "map" # Set to 'map' for global visualization

for coord in coordinates:
    # Publish GNSS as NavSatFix message
    msg = NavSatFix()
    msg.latitude = coord["latitude"]
    msg.longitude = coord["longitude"]
    msg.altitude = coord["altitude"]
    msg.status.status = 0
    msg.status.service = 2
    gnss_pub.publish(msg)

    # Create PoseStamped for the Path
    pose = PoseStamped()
    pose.header.frame_id = "map"
    pose.header.stamp = rospy.Time.now()
    pose.pose.position.x = coord["longitude"]
    pose.pose.position.y = coord["latitude"]
    pose.pose.position.z = coord["altitude"]

    path.poses.append(pose)
    path_pub.publish(path) # Continuously publish path with added points
    rospy.loginfo(f"Publishing GNSS data: Lat: {msg.latitude}, Lon: {msg.longitude}, Alt: {msg.altitude}")
    rate.sleep()

rospy.loginfo("Finished publishing GNSS path.")

if __name__ == '__main__':
    try:
        fake_gnss_publisher()
    except rospy.ROSInterruptException:
        pass
```

Fig 14. Simulated path generator

For real-time analysis, the paths are visualized using the `rviz_satellite` plugin within RViz. This plugin overlays the generated path on satellite imagery, providing an intuitive and realistic representation of the navigation route. Furthermore, these paths are recorded in a .bag or KML file format, enabling post-test review and validation on platforms such as Google Maps or Google Earth.

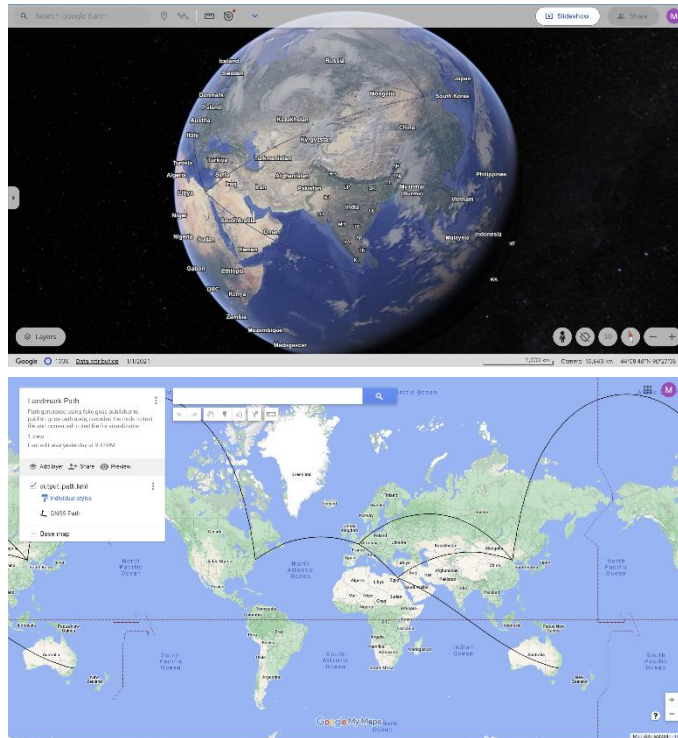


Fig 15. Simulated path visualization on Google Maps and Google Earth

By providing dynamic path generation and visualization, this phase ensures that the navigation stack can correctly interpret, generate, and refine simulated routes. The insights gained here inform critical adjustments to the algorithms and configurations before transitioning to real-world testing.

D. Google Maps Path Generator

The integration of the Google Maps API introduces a layer of realism, connecting the simulation environment with real-world geographic data. The API serves as a powerful tool for generating precise and contextually relevant navigation routes between user-defined locations.

Using the Google Maps Directions API, latitude and longitude coordinates are extracted to create a series of waypoints that define a navigable path. This process involves:

1. **Start and Destination Input:** Users specify the start point and destination.
2. **Route Calculation:** The API computes the most efficient route based on road networks, terrain, and navigation preferences (e.g., walking, driving).
3. **Coordinate Extraction:** The computed route is broken down into intermediate waypoints, each represented by latitude and longitude values.

These coordinates are then fed into the `fake_gnss_publisher` node to simulate a continuous GNSS feed along the planned route. The published data is visualized in real time using RViz and can be exported for further validation on Google Maps or Google Earth.

For instance, a route from the McMaster Automotive Resource Centre (MARC) to the McMaster Student Center demonstrates the integration's effectiveness. The visualized path provides a direct overlay on a map, offering a clear representation of the robot's intended navigation route.

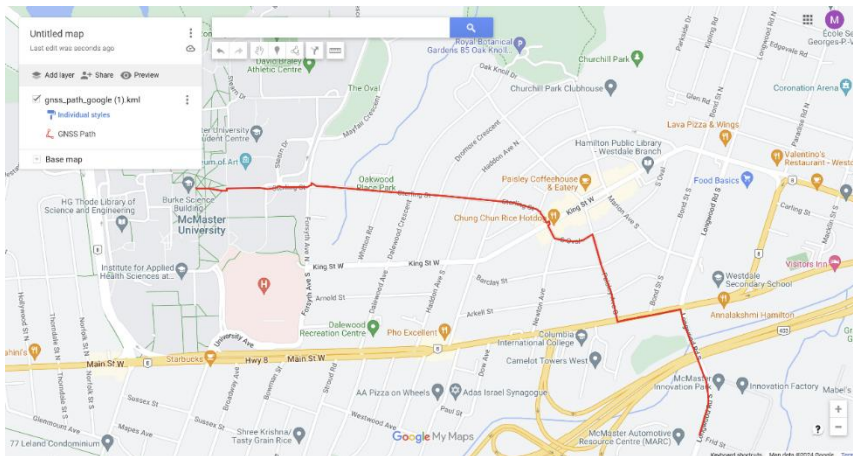


Fig. 16 Navigated path generated by Google Maps API and visualization on Google Maps

This final step bridges the gap between simulation and real-world application. By leveraging the Google Maps API, the system gains the ability to plan and simulate realistic navigation scenarios, offering valuable insights into the robot's performance in a geographically accurate context.

Visual perception

Oak D Camera

The Oak D camera is a very capable camera that can be utilized with its on-board processing power of 1.4 TOPs for artificial intelligence algorithms or on-board GPU on the computer. Enabling the GPU on the camera will provide real time object identification or any other functions available via other libraries such as open computer vision and YOLO. Another advantage of this camera is that it contains two stereo cameras that can record the depth of objects.

For the purposes of this project the YOLOv8 model was installed and ran within the docker for compatibility purposes.

The YOLOv8 model was installed along with the depth ai libraries.

To be able to do this the Lab.md from <https://github.com/Sep783Lab/macbot-labs-/blob/main/lab7.md> was used from the third step as they will be recited here below:

To create virtual environment for pip and python3 following the following steps,

3. Virtual environments for pip and python3

Virtual environments provide a sandboxed environment where you can install project-specific dependencies without affecting other Python projects or the system-wide Python installation. This isolation prevents conflicts between different versions of packages and ensures that each project has its own set of dependencies. How Virtual Environment Works

- 1. Isolation:** It isolates the Python interpreter, dependencies, libraries, and environment variables used within a specific project from the global interpreter.
- 2. Dependency Management:** It enables you to install Python libraries in the scope of the project rather than system-wide, thus avoiding version conflicts.
- 3. Switching Projects:** Virtual environments allow you to switch between different projects by activating or deactivating the corresponding environments, each with its own set of dependencies.

install and set up virtual environment (install two Python packages, virtualenv and virtualenvwrapper):

```
sudo -H pip3 install virtualenv virtualenvwrapper
```

The `.bashrc` file is a script file that contains commands and configurations for the Bash shell, which is the default shell for most Unix-like operating systems, including Linux. When you open a terminal session or start a new shell session, Bash reads and executes commands from the `.bashrc` file to set up the environment according to your preferences. In summary, the `.bashrc` file is a script file used to customize the behavior and environment of the Bash shell for individual users on Unix-like operating systems. adding virtual environment activation to the `.bashrc` file improves convenience, consistency, productivity, and adherence to best practices in Python development. It's a simple yet effective way to enhance your development workflow and ensure that you're always working within the appropriate environment for your projects.

open your `.bashrc` file located in your home directory with `vi` (you can open/edit the `.bashrc` in various way)

```
sudo vi ~/.bashrc
```

add following lines to the bash script:

```
# Virtual Env Wrapper Configuration
export WORKON_HOME=$HOME/.virtualenvs
export VIRTUALENVWRAPPER_PYTHON=/usr/bin/python3
source /usr/local/bin/virtualenvwrapper.sh
```

Save and reload the script by running the command `source ~/.bashrc`.

```
source ~/.bashrc
```

then create a virtual environment (in this example it's called `depthAI`).

```
mkvirtualenv depthAI_ve -p python3
```

to check that you are in virtual environment:

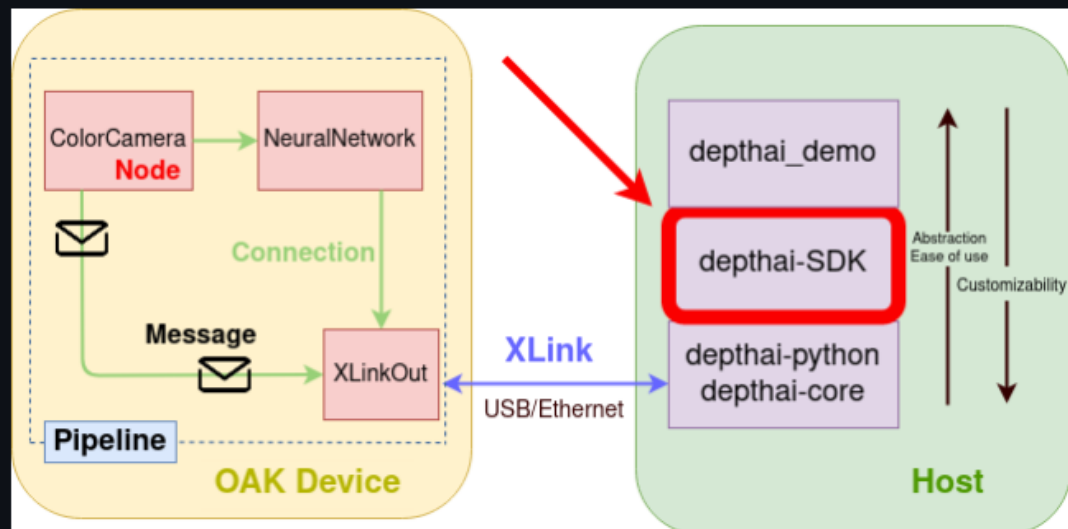
```
echo $VIRTUAL_ENV
```

it should return the following:

```
(depthAI_ve) jnano@macbot21:~$ echo $VIRTUAL_ENV
/home/jnano/.virtualenvs/depthAI_ve
(depthAI_ve) jnano@macbot21:~$
```

4. Installing Depth_AI

The DepthAI library from Luxonis is a software library designed to work with Luxonis DepthAI hardware modules. DepthAI is an embedded platform that integrates multiple capabilities for depth perception, object detection, and spatial AI tasks into a compact and low-power device. The platform is specifically designed for applications that require real-time processing of depth and visual data, such as robotics, augmented reality, autonomous vehicles, and more.



The DepthAI library provides a set of APIs and tools that enable developers to interact with DepthAI hardware and leverage its capabilities. Some of the key features and functionalities of the DepthAI library may include:

- **Depth Sensing:** The DepthAI hardware modules include stereo cameras and depth sensors, which allow for accurate depth perception and spatial mapping of the environment.
- **Object Detection:** The library may include pre-trained models and algorithms for detecting objects in the environment based on visual and depth data.
- **Neural Inference:** DepthAI hardware may be equipped with AI accelerators to perform real-time neural network inference for tasks such as object recognition, classification, and tracking.
- **Spatial AI:** The platform may support advanced spatial AI tasks, such as simultaneous localization and mapping (SLAM), gesture recognition, and scene understanding.
- **Integration with Python:** The DepthAI library may provide Python bindings and APIs, making it easy for developers to interact with DepthAI hardware and integrate it into their Python-based projects.

first download the updated Depth_AI dependencies from Luxonis:

```
#Download and install the dependency package
sudo wget -qO- https://docs.luxonis.com/install_dependencies.sh | bash
```



The command `wget | bash` is a combination of two commands chained together using the pipe (`|`) operator in a Unix-like shell environment. A dependency package in a shell script refers to an external software package or library that the script requires to function correctly. Ensuring that the necessary dependencies are installed on the system is crucial for the successful execution of the script.

then download the `depthai-python` from github repository:

```
#Clone github repository
git clone https://github.com/luxonis/depthai-python.git
```

In the context of OpenBLAS, which is an open-source implementation of the Basic Linear Algebra Subprograms (BLAS) library optimized for various CPU architectures, including ARM-based processors, setting the `OPENBLAS_CORETYPE` environment variable allows you to specify the target CPU architecture for optimization. By setting `OPENBLAS_CORETYPE` to `ARMV8`, you are instructing OpenBLAS to optimize its performance specifically for ARMv8-based CPUs (Jetson Nano is powered by an ARM Cortex-A57 CPU, which is an ARMv8-A architecture). Last step is to edit `.bashrc` with the line:

```
echo "export OPENBLAS_CORETYPE=ARMV8" >> ~/.bashrc
```

The `libcanberra-gtk-module` provides a bridge between GTK-based applications and the `libcanberra` library, allowing GTK applications to play sounds in response to events using `libcanberra`'s capabilities. It ensures that sounds are played consistently across GTK-based applications in a desktop environment

```
sudo apt-get install --reinstall libcanberra-gtk-module
```

Up to this point this resource explains and helps the user guide to the proper configuration

To activate this environment,

workon depthAI_ve

Once all of this is done and you find yourself in the environment,

cd /depthai-python/examples/Yolo to navigate to the proper python file to run the stereo cameras, if you want to see what python files you can run you can press `tb` twice to list the following

```
(depthAI_ve) sidewalk@ubuntu:~/depthai-python/examples/Yolo$ ls
tiny_yolo.py  yolov8_nano.py  yolov8_nano_with_xyz.py
```

(depthAI_ve) sidewalk@ubuntu:~/depthai-python/examples/Yolo\$ python
yolov8_nano_with_xyz.py

The underlined command will provide the proper script to run.

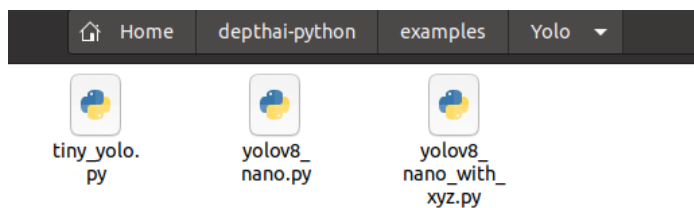
nano yolov8_nano_with_xyz.py ,

Will allow you to modify the script if need arises, Once in the script, The following commands takes care of assigning the proper hardware connection to access to the camera, this ensure the cameras are recognized by the Jetson Orin.

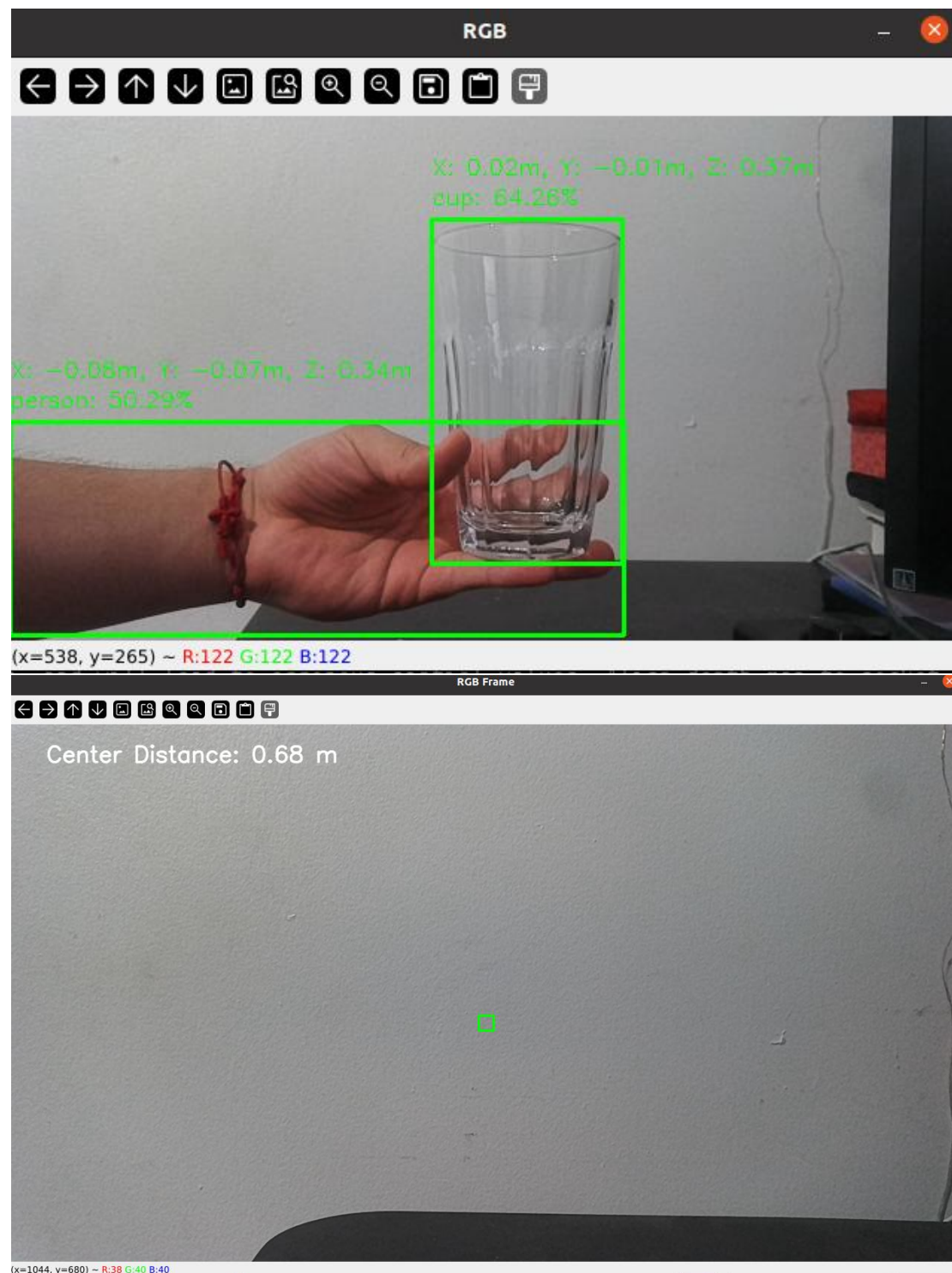
Mono camera settings

camLeft.setBoardSocket(dai.CameraBoardSocket.CAM_B)

camRight.setBoardSocket(dai.CameraBoardSocket.CAM_C)



The image above shows the following results of the x,y,z integration.



The y distance was tested through a script with a bounding box encasing the points were the script looks for the y distance to the most proximate object captures in the frame.

This docker below allows for the x,y,z coordinate systems to be integrated with yolov8 within the yolop container.


```

sudo docker run --rm -it \
  --runtime nvidia \
  --gpus all \
  --privileged \
  --mount type=bind,source=/dev/bus/usb,target=/dev/bus/usb \
  --mount type=bind,source=/home/sidewalk/depthai-
python,target=/home/sidewalk/depthai-python \
  -e DISPLAY=$DISPLAY \
  -v /tmp/.X11-unix:/tmp/.X11-unix \
  ping6508/gpucam:latest

```

Results



The xyz script yielded the coordinates above, now the xyz coordinates are being read off a display with a fixed y distance, for proper testing the camera should be placed in a vehicle while its moving. Before this the camera should also be calibrated. the Oak-d cameras boasts the following accuracy.

Depth accuracy (See [800P, 75mm baseline distance OAKs for details](#)):

- below 4m: below 2% absolute depth error

- 4m - 7m: below 4% absolute depth error
- 7m - 10m: below 6% absolute depth error

This could be tested with various known lengths to a wall or object.

The following link has various links to properly calibrate the camera

<https://docs.luxonis.com/hardware/platform/depth/calibration/>

Overall for the depth perception aspect of this project the cameras can identify the road, pedestrians, vehicles and estimate the distance to them.

Conclusion

The GoWrench autonomous vehicle project successfully integrates ROS-based modules and technologies to achieve waypoint self-navigation. By simulating GNSS and IMU data, the system ensures accurate localization and seamless navigation in pre-deployment testing environments. The integration of global and local planners with costmap configurations allows efficient path planning, while Google Maps API enhances route generation and visualization, bridging simulated environments and real-world applications. This study not only highlights the feasibility of achieving autonomous navigation through modular ROS frameworks but also provides a foundation for future advancements in off-road autonomous vehicles. Future work will focus on enhancing obstacle detection and integrating real-time sensor feedback for dynamic environments.