Real-Time UAV Obstacle Avoidance Using Nonlinear Programming

Suriya Suresh
M.Eng Robotics
University of Maryland
College Park

Apoorv Thapliyal
M.Eng Robotics
University of Maryland
College Park

***Abstract*—This paper presents the final project for the course ENAE788M: Hands On Autonomous Aerial Robotics. This project illustrates the work done in developing and implementing an obstacle avoidance algorithm incorporating non-linear programming for trajectory generation. The primary goal is to enhance the drone's autonomous navigation capabilities in obstacle rich environments. By processing the data from a stereo camera, the drone is able to dynamically compute a path around any detected obstacles and ensures it is able to reach a fixed goal while meeting a strict set of constraints. This ensures safe and quick navigation from way-point to way-point. This work also describes the various challenges faced during implementation phases and the solutions proposed to address them. The ModalAI Voxl 2 Sentinel Drone, equipped with an Intel RealSense D435i camera was used for all tests related to this project.**

I. Introduction

Unmanned Aerial Vehicles also commonly known as drones have become increasingly popular in academic as well as for commercial applications as they are versatile machines that can be integrated with different sensors to gather and process a variety of data. They can be used for a variety of applications such as mapping, delivery, inspection of structures, rescue missions and disaster management. They are used for such applications with integration of various sensors for perception such as cameras, LiDAR and depth cameras along with sensors for localization and navigation, which allow the drone to sense and map environments they are flying in.

Most often, they encounter other machines, buildings, humans or some form of obstructions that will require carefully navigating around them. Vision sensors aid in sensing such obstacles, which are integrated with a planner to traverse around the obstacle. The nature of obstacles can be static or dynamic in nature but for this project, we limit our consideration to static objects. We consider the use of a stereo camera that gives us depth images and point clouds. Stereo cameras are generally more affordable than LiDAR systems and have a compact form factor that makes them easy to mount onto drones. Unlike LiDARs, which require careful placement considerations, stereo cameras provide richer information compared to standard RGB cameras. For optimal performance, the sensors and planner must be integrated in a way that ensures real-time processing and includes a safety margin in the planning.

In this project, we focus on obstacle sensing using point clouds and employ a trajectory generator to plan a path around obstacles with appropriate velocity and acceleration profiles. This system operates in real-time and utilizes a non-linear optimizer for trajectory planning.

The paper is structured as follows: first, we present the approach used for the project. This is followed by a discussion of the simulations and hardware implementation. Finally, we present the results of our project and conclude with suggestions for future work that builds upon this project.

II. Optimal Problem Formulation

Optimal control entails minimizing a cost function, usually dependent on both state and control variables. In this implementation, a direct control over the state variables was introduced, as the objective of this approach is to define a reference path profile around the obstacle.
First, we can discretize the system by considering **X, Y, Z** as our array of decision variables we aim to optimize:

$$\mathbf{X} = [X[0], X[1], ...X[n-1]]$$

$$\mathbf{Y} = [Y[0], Y[1], ...Y[n-1]]$$

$$\mathbf{Z} = [Z[0], Z[1], ...Z[n-1]]$$

Let us now consider $(X_{initial}, Y_{initial}, Z_{initial})$ and $(X_{final}, Y_{final}, Z_{final})$ as the initial and final waypoints, with acceleration along each axis (a_x, a_y, a_z) as the inputs to the UAV.

With this, we can describe our cost function, **J**, as:

$$\mathbf{J} = min((\mathbf{Z} - Z_{final})^2)$$

$$+min((\mathbf{X} - X_{final})^2 + (\mathbf{Y} - Y_{final})^2 + (\mathbf{Z} - Z_{final})^2)$$

$$+min(a_x^2 + a_y^2 + a_z^2)$$

subject to :
Dynamics constraints: $a_{max} \geq a \geq a_{min}$
Initial and Final conditions:

$$X[0] = X_{initial}, X[n-1] = X_{final}$$

$$Y[0] = Y_{initial}, Y[n-1] = Y_{final}$$

$$Z[0] = Z_{initial}, Z[n-1] = Z_{final}$$

The first term focuses on rapidly minimizing the vertical displacement between the initial and final positions. This

leverages the drone's dynamics, simplifying the maneuver primarily to movement in the X and Y axes.
The second term guides the UAV toward the goal while penalizing deviations from the optimal straight-line path between the two points. This promotes adherence to a path with minimal deviation while meeting any additional position constraints.
The third term adds a surcharge to the acceleration input in our model, thereby encouraging the generation of a trajectory with minimal control inputs.

In the situation that an obstacle has been detected at position $(X_{obs}, Y_{obs}, Z_{obs})$, we may add a Path constraint defined as:

$$\sqrt{(\mathbf{X} - X_{obs})^2 + (\mathbf{Y} - Y_{obs})^2 + (\mathbf{Z} - Z_{obs})^2} \geq d$$

Essentially this constraint states that every point in the path must be a minimum safety distance, d, away from the obstacle position.

III. Approach

The code operates on a flag-based architecture, where specific flags initiate corresponding actions. For instance, when ready to execute the mission, the drone awaits the generation of waypoints before proceeding with any further tasks. Until then, it remains in standby mode, concurrently running all other processes.
The **ready** flag determines whether the drone is oriented towards the goal position. Once the drone is aligned with the goal, this flag is set to True.
The **waypoints** flag holds the generated waypoints from the Optimal Trajectory Generator. When set to False, it indicates that there are no waypoints to track. Otherwise, it contains the x, y, z points to track.
The **reached** flag informs the controller that the drone has reached the desired goal once the final waypoint is reached. Upon setting this flag to True, the **ready** and **waypoints** flags are set to False, signaling the end of the mission and transitioning to standby mode.

A. Obstacle Position Estimation

From the Intel RealSense, we derive estimations of obstacle positions surrounding the UAV. By applying appropriate thresholds to the points acquired from the RealSense, we can filter out interference from the ground plane in our obstacle position estimates. The processing of point cloud data occurs concurrently with the Master Controller on a distinct ROS node. This segmentation of tasks aids in compartmentalizing our operations and facilitates the identification of potential points of failure.
The position data from the point cloud is published to the *obstacle_point* topic, which is subscribed to by the ROS node responsible for controlling the drone. Specifically, the *obstacle_position_callback()* function continuously updates the obstacle position, ensuring that the drone has access to the latest obstacle position data.

The pipeline for position estimations is outlined in Fig 1 and Fig 2.

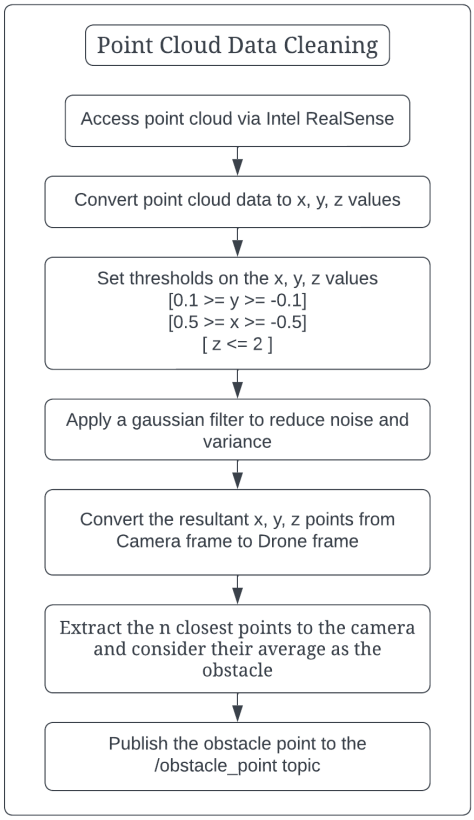

Fig. 1. Pointcloud Processing

B. Optimal Trajectory Generation

The trajectory generation initiates only when the **ready** flag is set to True; until then, trajectory generation remains on standby.
The drone's initial position is determined by the local vehicle position, with predefined velocity and acceleration constraints. Utilizing our approximations, we employ the point mass model to generate the trajectory. The trajectory generation pipeline is outlined in Fig 3

C. Master Controller

The Master Controller orchestrates the switching of flag states, which in turn dictate the actions executed by the drone. The flow of control is illustrated in Fig 4.

D. Hardware and Software Used

This project utilizes a ModalAI Voxl 2 Sentinel Drone equipped with an Intel RealSense D435i camera mounted on the front. The ModalAI Voxl Drone is powered by a Qualcomm-based ARM chip running a Linux OS, providing

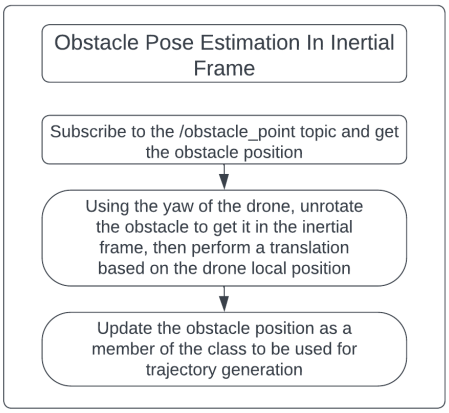

Fig. 2. Obstacle Pose Estimation

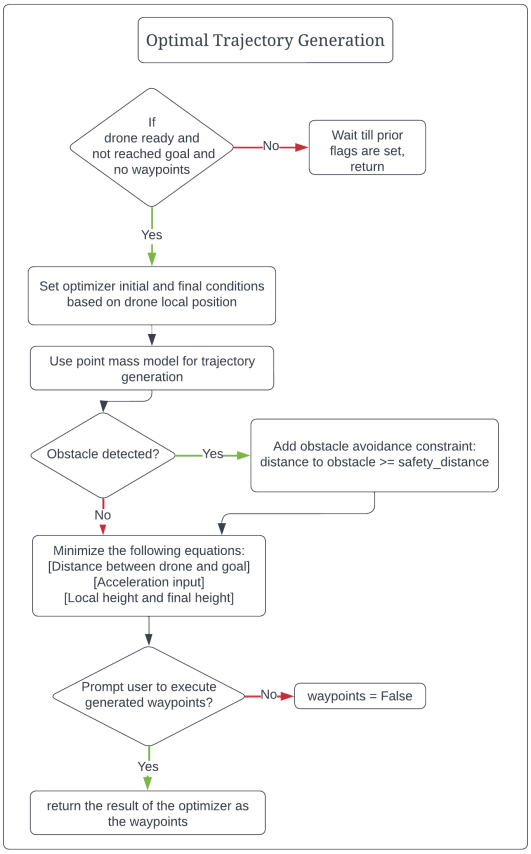

Fig. 3. Optimal Trajectory Generation

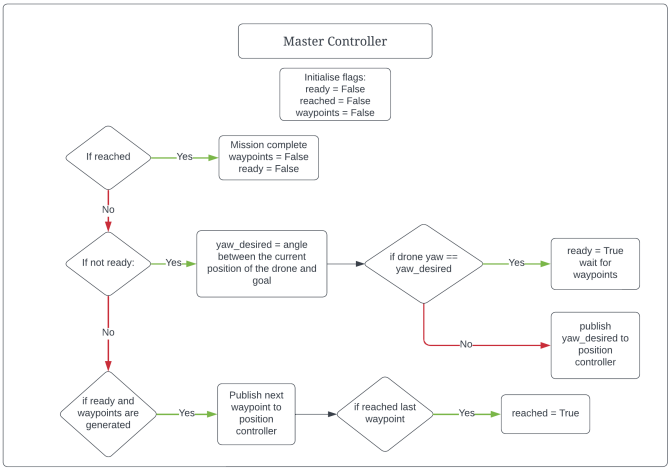

Fig. 4. Master Controller

easy adaptability to various use cases. The D435i is a stereo camera featuring RGB sensors, IR sensors, and a pair of stereo cameras, delivering RGB, depth, and point cloud outputs. The IR module enables the camera to operate independently of lighting conditions and effectively distinguish objects in featureless backgrounds. Outputs can be viewed using the Intel RealSense viewer or the ROS2 Wrapper provided by Intel .

We use the ROS2 wrapper and its SDK inside a Docker container running on the drone. This setup is necessary due to Intel's lack of support for the ROS and Ubuntu versions on the drone, as well as the need to install specific Python packages that require a newer version of Python 3. The Docker image is based on Ubuntu 22.04 running ROS2 Humble.

We used GEKKO, a Python package for optimizing algebraic equations, which supports non-linear programming (NLP) essential for this project. Due to platform constraints on the drone, the supported solvers do not allow NLP to be solved locally [5], necessitating the use of a remote solver. Consequently, an internet connection is required to run the code on the drone.

We utilized standard ROS Python packages for message definitions, quality of service settings, and other ROS-related functions. For point cloud processing and filtering to clean the camera output, we employed Open3D.

IV. Simulation

We conducted our simulations using the PX4 SITL based on Gazebo Ignition within a simulated Bay-Lands world, as illustrated in Figure 5. The x500 Quadrotor, equipped with a depth camera, was used for these simulations. The depth camera, modeled on the OAK-D, publishes RGB, depth, and point cloud data as ROS topics, facilitating easy access and visualization of the output.

The Gazebo simulator publishes camera topics as gz topics, which require the use of the ros_gz bridge to convert them into ROS2 topics. RViz2 can then be used to visualize the RGB and depth images. The point clouds generated by the depth

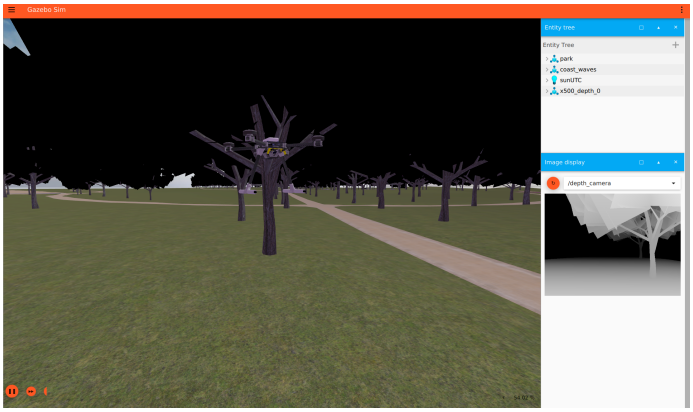

Fig. 5. x500 in Bay-Lands World

camera are also available as ROS topics. Figure 6 shows the output of the point clouds in RViz. These point clouds cover a fixed range and provide high-quality output. However, they also capture part of the ground, which can introduce challenges when used for obstacle avoidance.

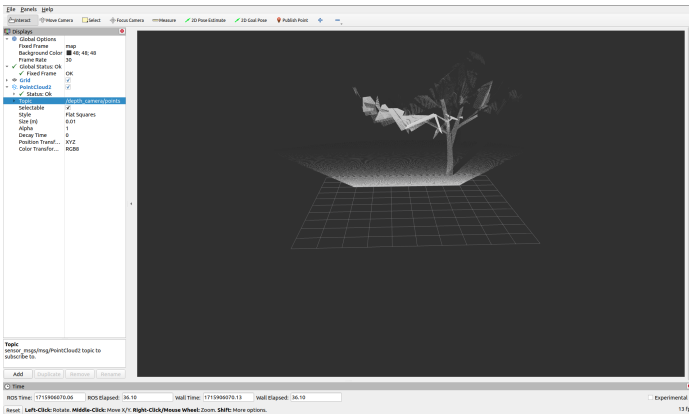

Fig. 6. PointCloud in Rviz

For converting ROS2 pointcloud2 message type to xyz pointcloud array code from here was used. Messages from any topic publishing the same type of message can be used to extract the XYZ points in the form of a *[n,3]* array along with the associated colors is possible. The benefit of this approach is that it allows the point cloud data to be directly processed by the Open3D Python package without requiring any additional conversions.

V. Hardware Implementation

A. Integrating RealSense with the Sentinel

The Realsense 435i was intially tested on a laptop using Realsense Viewer which allowed us to understand the depth map generated.Fig 7 shows the ouput obtained when visualized. This also allows you to change settings of the camera and enable different filters and tweak the depth visualization.

For integrating the Realsense Camera onto the Sentinel, a mount had to be designed to attach the camera to the drone.

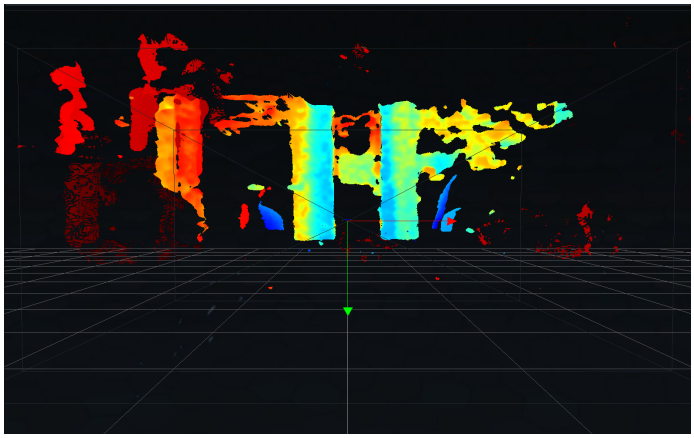

Fig. 7. RealSense Ouput

A 3D printed mount was used and attached to the front of the drone as shown in Fig 8. The camera mount was positioned almost level with the landing skids, which poses a risk of damage to the camera if the drone were to land hard during flight. A USB 2.1 hub was attached to the drone to connect

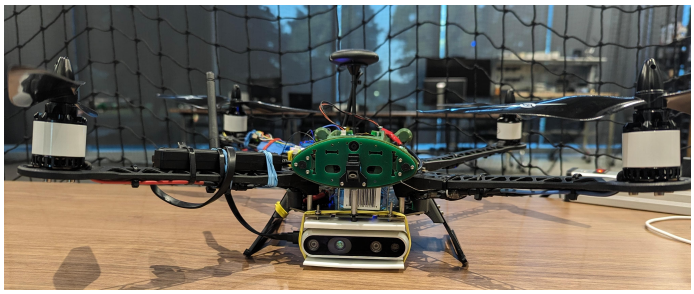

Fig. 8. Camera Mounted on Drone

both the camera and WiFi to the onboard computer. The USB hub can be seen attached to the motor mount of the drone in Fig 8.

The camera is accessible in Linux using the Realsense ROS2 Wrapper and the Realsense SDK. To facilitate this, a docker container was utilized to run both the wrapper and the SDK. The setup of the docker container followed the guidelines provided by modalAI documentation [6]. Additionally, the ROS2 core and ROS build tools were installed as per the instructions outlined on the ROS wiki [1]. Furthermore, docker containers running ROS2 Humble were employed for simulating the drone on a PC as well.

The ROS2 wrapper facilitates the publishing of camera topics as ROS2 topics. By executing the command `ros2 launch realsense2_camera rs_launch.py pointcloud.enable:=true pointcloud.ordered_pc:=true` , the realsense2_camera_node is launched, enabling RGB, depth image, and ordered pointcloud topics such as */camera/color/image_raw*, */camera/depth/image_rect_raw*, and */camera/depth/color/points*. The comprehensive range

of topics published by running the realsense2_camera_node is depicted in Fig 9. Additionally, compressed image topics are automatically enabled when the *image-transport-plugin package* is installed. It's worth noting that the Realsense camera should ideally be connected to a USB3.0 port. During node launch, the camera automatically adjusts its maximum supported resolution to 640x480p streaming at 15 FPS.

Fig 10 shows the view the camera sees when the obstacle is kept in front of it.The camera is mounted on the drone in the figure.

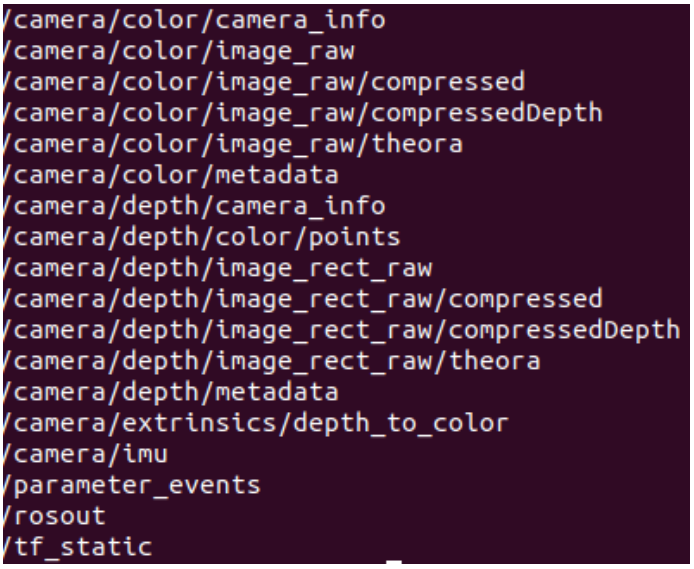

Fig. 9. Topics Published by Realsense Camera

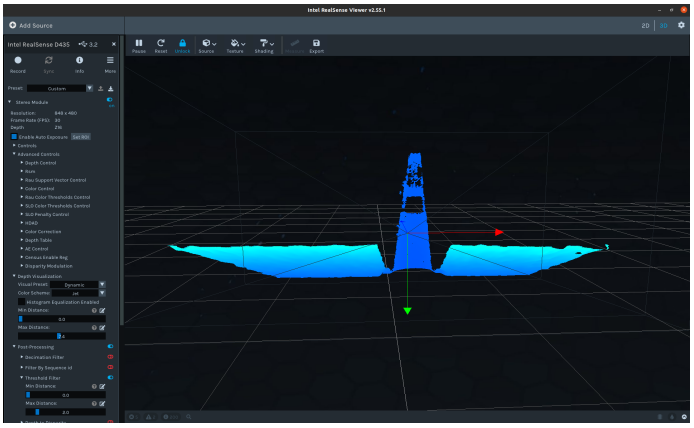

Fig. 10. Depth Camera Output in front of Obstacle

VI. Challenges Faced

This section describes the major challenges we faced during the course of this project and have listed how we overcame some of them.

A. Publishing PX4 Topics and Camera Topics under Same ROS Domain

The drone initiates the publication of topics immediately upon booting if the *voxl-configure-microdds* is enabled. By default, ROS2 publishes topics under the domain id 0. However, when the camera node is executed, it encounters difficulty publishing topics under the same domain, even if the exported environment variable **ROS_DOMAIN_ID** is set to zero.

This obstacle can be addressed by assigning different domain IDs to the PX4 topics published by the MicroXRCE Agent and the camera topics, avoiding the values 0 or 1. It's essential to export the environment variables **ROS_DOMAIN_ID** for ROS nodes and **XRCE_DOMAIN_ID_OVERRIDE** for the agent, both on the drone and the docker container, ideally via the *.bashrc* file. This configuration ensures that the MicroXRCE agent is initialized with the specified domain ID upon drone boot-up. Note that exporting the flag in a terminal and restarting the MicroXRCE does not achieve the desired effect; it must be set as part of the system initialization process. This setup allows the docker container to access both the PX4 and camera topics seamlessly.

B. Viewing Image Output in RViz

To view the camera feed in RViz, it's essential to publish a static coordinate transform to tf2, linking the base frame of the camera to the map frame. RViz relies on this transformation to interpret messages from the camera's coordinate frame within the context of the world frame, as outlined in the ROS documentation. Failure to publish this static transform can lead to RViz being unable to properly process incoming messages from the camera topics, potentially causing a buffer overflow.

C. GEKKO Permissions Issue

Running GEKKO on linux hosts has permissions issue to run the exectuable of the package when the solve method is called. This was observed on the drone as well as the docker container. The solution was to set permissions of the folder containting the package as *chmod 755 -R path_of_file*.

D. Inconsistent Point Clouds

The intel real-sense camera gave inconsistent number of point each time it published to the topic.This in turn would affect quality of the detection of obstacles, consequently affecting the identification of obstacle location.

VII. Results

The videos of our testing can be found here here.We were successful in achieving total autonomy of the drone being able to detect a obstacle and plan an trajectory around it. We only require to confirm the rough location of the obstacle when the node is running to ensure the drone does not accidentally crash into a obstacle.

We can see that the drone is able to avoid the obstacle when it detects and flies in a straight line if no obstacle is detected in the videos given above.

Integration of the Planner and obstacle detection using point-clouds in simulation was not done.

The instructions to run the code are given in the README of the project submission. We have included code to run the demo on the drone as well as the code to visualize the point-clouds in RViz. The code can be found here.

VIII. Contributions

In this project, the contributions of the authors are as follows. Apoorv Thapliyal designed and implemented the master controller, trajectory generator, and obstacle avoidance algorithms, which collectively formed the core of the drone's autonomous navigation system. This involved the development of a flag-based architecture to govern drone actions, the creation of an optimal trajectory generation algorithm to guide the drone towards its destination, and the integration of obstacle detection and avoidance strategies to ensure safe navigation in dynamic environments.Jayasuriya Suresh focused on implementing the code for interfacing with the Intel RealSense camera, including configuring camera settings, collecting and processing point cloud data, and developing algorithms for obstacle detection. Additionally, both authors collaborated on refining the overall system architecture and conducting experiments to evaluate its performance. The collective efforts of the authors contributed to the successful development of an autonomous drone navigation system capable of avoiding obstacles in real time.

IX. Conclusion and Future Work

We have the potential to expand our project to include real-time dynamic obstacle avoidance. However, one significant obstacle we encountered was the slow update times of the topic publishing obstacle points from the Realsense camera. Enhancing this aspect or considering the use of a Time-of-Flight (ToF) camera could significantly improve the performance of obstacle detection.

Acknowledgment

We extend our sincerest appreciation to Professor Joseph Conroy for his exemplary mentorship throughout the duration of this research project. His insightful guidance, unwavering support, and profound expertise have significantly influenced the trajectory of our study and contributed to its scholarly rigor. We are profoundly grateful for his invaluable contributions, which have been pivotal in shaping the direction of our research endeavors and overcoming various challenges encountered along the way. Additionally, we would like to thank our friends and other faculty members for their support and encouragement throughout this endeavor.

References

[1] "Documentation - ROS Wiki." https://wiki.ros.org/

[2] Intel RealSense, "Depth Camera D435 – Intel® RealSenseTM depth and tracking cameras," Intel® RealSenseTM Depth and Tracking Cameras, Dec. 05, 2022. https://www.intelrealsense.com/depth-camera-d435/

[3] "ROS2," Intel® RealSenseTM Developer Documentation. https://dev.intelrealsense.com/docs/ros2-wrapper

[4] "GEKKO Optimization Suite — GEKKO 1.1.1 documentation." https://gekko.readthedocs.io/en/latest/

[5] "solving a large size NLP with linear constraint using Gekko," Stack Overflow. https://stackoverflow.com/questions/72925962/solving-a-large-size-nlp-with-linear-constraint-using-gekko

[6] "Setup voxl-cross," ModalAI Technical Docs. https://docs.modalai.com/voxl-docker-and-cross-installation/

[7] Aldao E, González-deSantos LM, Michinel H, González-Jorge H. UAV Obstacle Avoidance Algorithm to Navigate in Dynamic Building Environments. Drones. 2022; 6(1):16. https://doi.org/10.3390/drones6010016