Simulator for UAV Localization and Navigation in Various GPS-Denied Scenarios

Mihaela Orić *, Filip Novoselnik *, Vlatko Galić *
* Protostar Labs, Osijek, Croatia
mihaela.oric@protostar.ai

Abstract — **Some industries demand successful work of UAVs in areas without GPS, for example, drone operation in warehouses with metal roofs. Safe operation of UAVs includes successful "Return To Home" (RTH) protocols for retaining aircraft operation stability after losing signal and successfully finding the way back to the flight starting point. Development of a reliable UAV localization and navigation system is challenging due to the high dependence on manual work and safety concerns. To mitigate these limitations, we developed a UAV simulation pipeline that can be used for localization and navigation algorithm testing. The pipeline uses ROS and connects algorithms with a 3D environment simulator and provides a graphical interface for easy operation. Users have the ability to input waypoints for the trajectory, start the simulation, limit the GPS signal dynamically, and estimate the success of return. By changing the environment, trajectory shape, and length, various scenarios can be tested. Simulations of RTH success with realistic constraints promise to preserve resources in the testing stage of the UAVs decreasing security risks.**

Keywords - localization; navigation; unmanned aerial vehicles; ROS; simulator

I. Introduction

Unmanned aerial vehicles (UAVs) play an important role in civil and military applications, but with Industry 4.0, their use expands into many other fields. Whether they are used in open surroundings on long distances, such as in military and civil applications, or in closed environments, such as in manufacturing and warehouses, the stability of their communication and their safety protocols need to be on the highest level possible. An UAV equipped with sensors can be used for visual inspections of its surroundings, quality inspections, or inventory monitoring, whilst one equipped with special tools and grippers can do some physical work such as carrying light cargo [1]. When an UAV is working in a closed environment, there is a high chance that the GPS signal will be affected by the metal roofing and will be of very low quality or even unavailable. Moreover, if an UAV is used on an open field, the signal can be disrupted by tough atmospheric conditions, electronic interference, or by GPS jammers that interfere with the signal deliberately [2]. To avoid losing control of the aircraft and unexpected aircraft behavior, methods for sensor based navigation and localization and mapping algorithms have been developed.

To program a robot to do a specific action, for example, move through an unfamiliar environment while simultaneously localizing itself and mapping the environment, a Robot Operating System (ROS) can be used. In this paper, a robot being programmed to localize itself and navigate through the environment is a UAV. ROS is a robot software framework with architecture fitting for a wide range of domains [3]. ROS has become a software standard in robotics because of its goals which are being peer-to-peer, tools-based, multi-lingual, thin and free, and open-source.

ROS2 is the second generation of the Robot Operating System which implements production-grade algorithms and features necessary for achieving high level security and reliability [4].

II. System Overview

The framework presented in this paper is separated into two parts - one side contains the simulation software with graphical user interface, while the other side contains the odometry algorithm. A design scheme is presented on Figure 1.

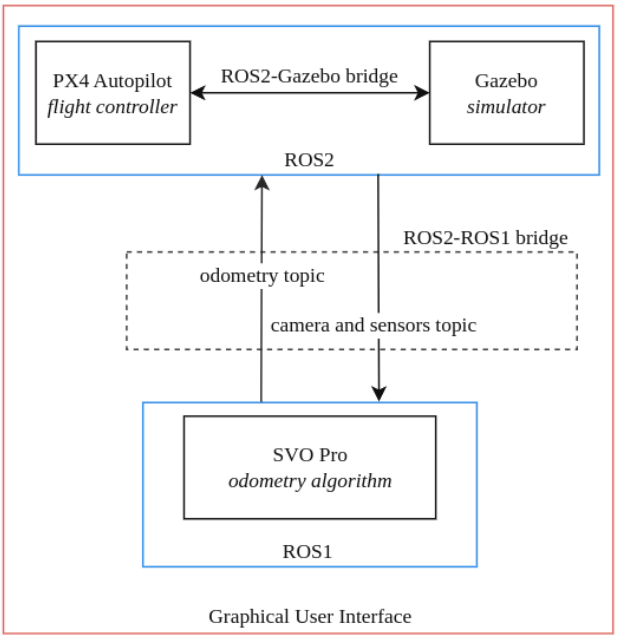

Figure 1. Design scheme of this pipeline.

A. PX4 Autopilot

PX4, an open-source software for drone and unmanned vehicle flight control, is used [5]. Vehicle controllers, often called flight controllers, run a flight stack software that

controls the aircraft. PX4 flight stack controls different vehicle frames, implements powerful safety features and flight modes, includes various sensors and peripherals, and is integrated with ROS. PX4 flight code can be used with simulators where the PX4 vehicle gets modeled in a simulated environment. Simulators supported by the PX4 core development team are Gazebo, Gazebo Classic, jMAVSim, while the community supported simulator list is larger. Controlling the vehicle with a PX4 controller in a simulated world functions the same as controlling a physical vehicle in a real-world environment which allows users to fully test the control code and algorithms before deploying them to a real aircraft. Its integration with robotics APIs and the possibility of use in simulation made it a good choice for a pipeline presented in this paper.

PX4 allows the vehicle to be put into the offboard mode where it receives parameters such as position, velocity, and acceleration from an external source other than the flight stack. Offboard mode is used in this pipeline to control the UAV through a ROS node. Offboard ROS node publishes waypoints in the TrajectorySetpoint message format.

PX4 software and the simulator were initially planned to be run on ROS1 in this framework to make the communication with the odometry algorithm easier. This ended up being challenging and very unstable. To avoid these complications, a decision to run PX4 software on ROS2 instead of ROS1 was made.

B. Gazebo Simulator

Gazebo is a 3D robotics simulator recommended by the PX4 team. It simulates real-world physics and conditions such as gravity, wind, air pressure, etc. The role of the simulator in this pipeline is to provide the physics and the environment for the testing. It is important to use a simulator that replicates the conditions of the real environment in which the UAV will work in since those conditions could affect the success of the flight and algorithms. Other than world physics, Gazebo implements a wide range of sensors used in robotics. It is developed to fit for all simulation contexts. For successful rigid body simulation in the Gazebo simulator, each object should include model shape constructed from an SDF file, collision shape files and joints and joint types. Since it is free and open-source, the community is large and creates many third party sensor plugins and robot assets. Parameters and messages from the Gazebo simulator are published as Gazebo topics on TCP/IP sockets. The principle of sending and receiving messages in Gazebo is based on a subscriber-publisher relationship which is the base of ROS message exchangement. Publishers send messages to a named channel called topic and subscribers receive messages using callbacks. To integrate ROS2 with Gazebo, it is necessary to establish communication between them. Gazebo-ROS2 integration in this paper has been established using a network bridge called ros_gz_bridge [6]. Even though this bridge offers support to a limited number of message types, it is suitable for this case since it supports sensor message types needed for visual-inertial odometry algorithms.

After setting up this bridge for bidirectional communication, ROS2 code can read Gazebo messages (work as a subscriber) and send messages to Gazebo (work as a publisher). A drawback of using Gazebo as a 3D simulator is that it does not allow easy 3D model import that other simulation software may offer. The model should be prepared in a specific way before the import. Firstly, it is important to question the complexity of the used 3D mesh since an overly large mesh can affect the simulation performance and could cause problems for users wanting to achieve a real-time simulation. Overly complex meshes should be simplified or split into multiple parts. Secondly, scale and rotation should be checked to see if they fit Gazebo standards and edited if necessary. The center of the mesh is at (0, 0, 0) and the front of the mesh should point in the positive x-axis direction. Gazebo uses the metric system so the scale of the object should be adjusted to achieve the wanted size in meters. Thirdly, the mesh should be exported in the Collada or SDF format. Finally, the exported file should be added to the Gazebo world file.

To achieve a high level of detail in the simulated world, photogrammetry was used for 3D model creation. Photogrammetry is a method of using photographs to gather information and measurements of the photographed object or area. It stitches images together using triangulation to create a digital 3D copy of the physical object or world. 3D models created with photogrammetry and a sufficient number of photographs are highly detailed and have realistic textures. By using a photogrammetric model in this pipeline, it was possible to test the detection of features visible on a real piece of land without physically flying the drone there and doing the detection while flying. Example of a photogrammetric 3D model used in the experiments is shown on Figure 2.

Figure 2. 3D environment made using a photogrammetry tool.

C. Semi-direct Visual Odometry

Odometry algorithm estimates the position of the aircraft relative to the starting point. While the simulator provides the environment for the testing, the odometry algorithm takes physics and sensor readings from the simulator as its inputs and calculates the position of the aircraft as the output. Visual odometry is a technique that uses a stream of images captured with a camera mounted

to a vehicle to estimate change in position over time. Camera sensor is a cheaper alternative to some other more expensive sensors used for odometry, such as lidar sensors, optical flow, and GNSS, which made it an area of interest for computer vision and robotics engineers. The base principle of visual odometry is calculation of pixel displacement between multiple images [7]. Basic steps of visual odometry are image acquisition, finding distinct features on images, matching these features on multiple images, and finally calculating the displacement of the features on multiple images.

Semi-direct visual odometry (SVO) is a visual odometry method that tracks and triangulates pixels by relying on proven optimization methods [8]. Its speed and competitive accuracy made it a suitable choice for the odometry algorithm used in this pipeline. It uses two parallel threads for estimating the position and mapping the area. By combining visual input with the readings from other sensors, the resulting odometry output is more robust and less prone to drift. Camera and an inertial sensor (IMU) are used as inputs into visual-inertial odometry algorithms. Such a visual-inertial odometry algorithm used in this paper is SVO Pro which is an extension of the previously published SVO algorithm [9].

SVO Pro ROS node subscribes to the camera topic message which is originally published by Gazebo and then sent to ROS2 using a Gazebo ROS bridge. It also publishes calculations to the odometry ROS1 topic.

D. ROS Bridge

PX4 and Gazebo simulator are working in ROS2, while the SVO algorithm works in ROS1. Those two sides of the pipeline are communicating through a bridge which exchanges messages between ROS2 and ROS1. Bridge is available as a ROS2 package [10]. It provides options in selecting the direction of the communication.

It was possible to avoid using this bridge by using previous versions of PX4 and Gazebo Classic simulator which are integrated with ROS1. However, as ROS2 becomes the standard in robotics software, it is useful to have a pipeline that is compatible with it. In this way, long term usability of this pipeline is achieved. Furthermore, setting up a ROS1-ROS2 bridge allows users to implement a wide range of odometry algorithms which were developed prior to ROS2 release.

E. Graphical User Interface

Starting every part of this system or any other similar system which uses flight control, simulator, and odometry algorithm, can be time-consuming and complex since nodes need to be started in the correct order. Moreover, setting the trajectory waypoints for the vehicle needs to be done programmatically prior to starting the simulation, which requires programmer knowledge, or specific tools need to be used, which complicates the setup. This pipeline avoids all those problems and simplifies the running by connecting every step of the pipeline and integrating basic options inside a graphical interface. This way, users can quickly start the testing process by clicking on a few buttons. To make trajectory input faster, easier, more intuitive and more transparent, an interactive map of the simulator world is available. Interface loads the selected 3D model of the world, calculates its top view plan and shows it as an image which then represents the map of the world. The conversion of the 3D model to a map proved to be more challenging than it was initially thought because processing point clouds and calculating a top view can be computationally demanding. However, this step increases the quality of the interface since the user only needs to change the 3D model but does not have to provide the top view of the used model.

To start collecting waypoints, the user has to press the "Start collecting waypoints" button. Waypoints are selected by clicking on the map with the ability to delete waypoints. Conversion from pixel locations of waypoints selected on the map to a 3D location in the simulated world is done in the background. This way, users can get a better feeling of how waypoints are positioned than if the waypoints had to be set numerically in the code. Furthermore, there is a slider which can be used to add rotations to each waypoint. To save the selected waypoints, the user has to press the "Save waypoints" button. Waypoints are saved locally before the start of the simulation which allows them to be loaded again and provide repeatability of tests. The interface also provides options for GPS signal availability, which can be turned on or turned off prior to the mission start, and also turned on or turned off mid-flight.

Graphical User Interface was made in Python using Qt tools [11]. Qt is a free open-source platform for creating graphical user interfaces and applications that can be run on multiple platforms, such as Windows and Linux. In this paper, PySide6 module was used for Qt tools integration with Python. Figure 3. and 4. show the graphical interface made for this pipeline.

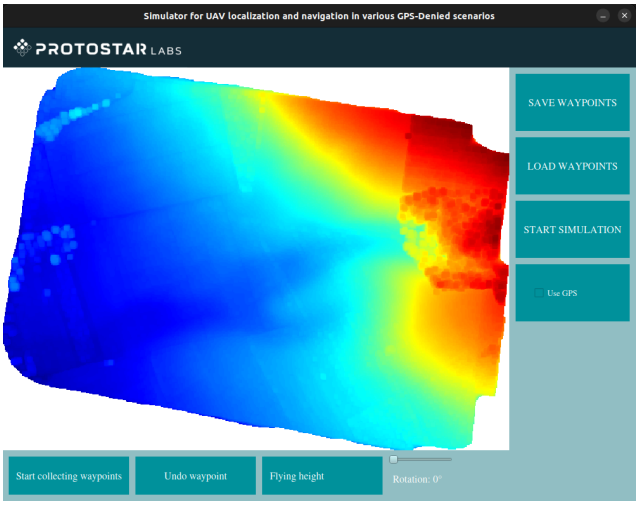

Figure 3. Graphical user interface of the simulator for UAV localization and navigation.

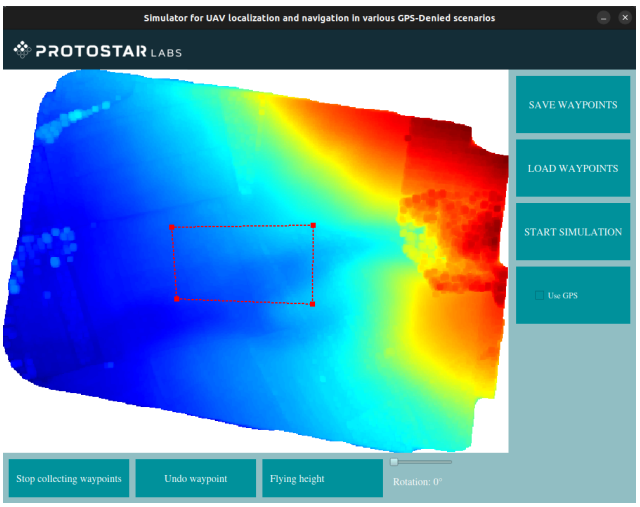

Figure 4. Graphical user interface of the simulator for UAV localization and navigation with waypoints selected.

III. EXPERIMENTS

Aerial photographs of a vineyard have been acquired using a drone flying in a grid pattern. Those photgraphs were processed using the photogrammetry software WebODM [12]. The 3D model calculated with WebODM was converted to an STF format which is suitable for Gazebo and the converted model was put into a world file. SVO Pro visual-inertial odometry algorithm was used and topic names were adjusted so that they are named the same in ROS1 and ROS2 side of the framework. Drone used in the simulation was a Gazebo "gz_x500" drone model and is shown on figure 5. Sensors added to the drone were IMU, magnetometer, barometer and RGB camera sensor.

Figure 5. Gazebo "gz_x500" drone model.

The simulation was run multiple times with the GPS signal turned off. Some of the more important settings used in odometry algorithm configuration for this set of tests include setting the pipeline to mono option, setting grid size to 30 which can be lowered when processing power is limited, and enabling loop closure. Three different trajectory shapes were selected for testing purposes with the goal of testing specific cases of navigation. One trajectory consisted of a straight flight in one direction and a return back to the starting position. The point of this test was to see how well the algorithm would perform over a long distance and how big the drift would be. The second case was a flight with a rectangular trajectory. This trajectory shape has multiple 90 degree rotations which show the accumulation of the rotation estimation error. Lastly, a trajectory with a circular shape was set. This circular trajectory shows how well the odometry algorithm works when the trajectory is not straight. The results of these three test cases are shown on figures 6., 7., and 8.

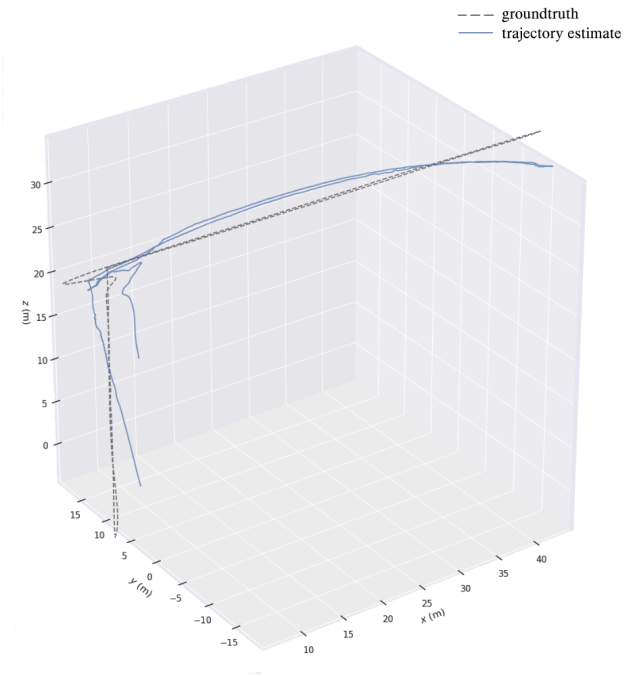

Figure 6. Straight trajectory estimation.

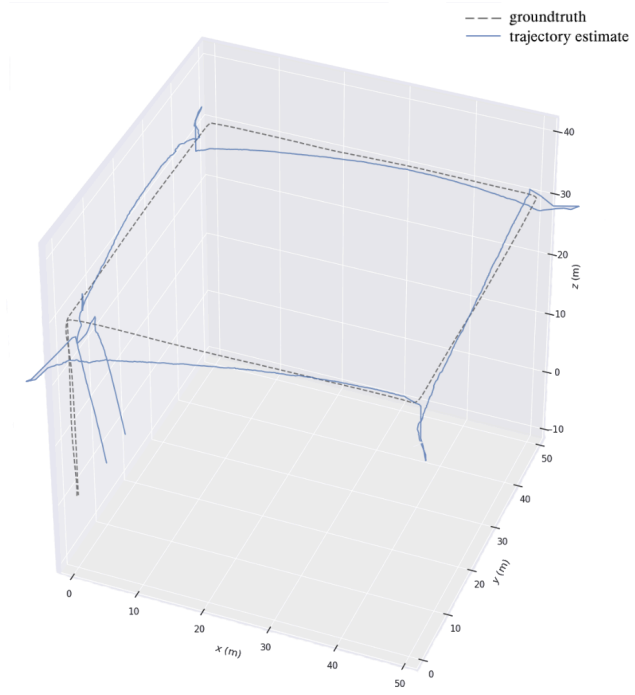

Figure 7. Rectangular trajectory estimation.

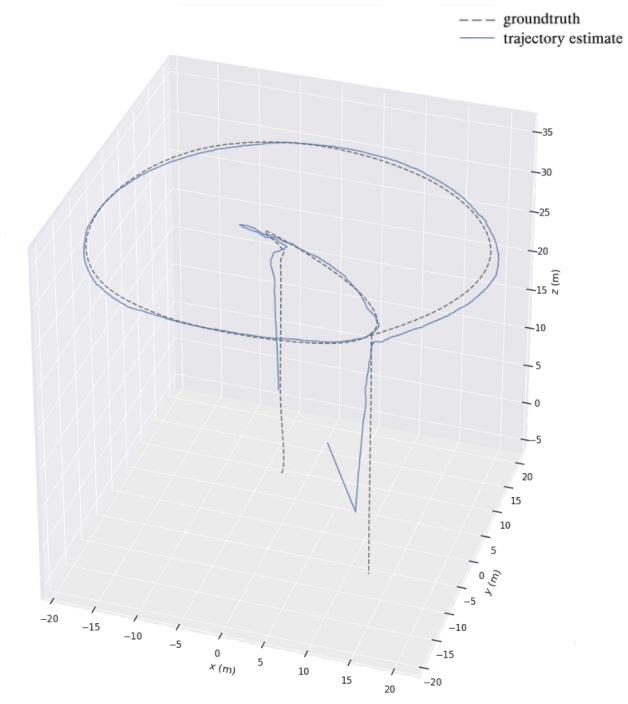

Figure 8. Circular trajectory estimation.

The appearance of the area above which the UAV will fly can highly affect the success rate of the visual odometry algorithm. An area with a high level of detail and complex scenery allows the algorithm to be precise and confident when matching the features since the number of detected features would most likely be high. On the other hand, land with a uniform appearance can cause problems because there are not enough distinct features that can be matched. For example, a visual odometry algorithm running on a drone flying above a residential area would be more successful than the same algorithm running when the drone is flying above a large grass field.

Absolute position error (APE) metrics in meters for each trajectory shown on figures 6., 7., and 8. are visible on table 1.

TABLE I. ABSOLUTE POSITION ERROR METRICS

APE metric	Trajectory shape		
	Straight	***Rectangular***	***Circular***
Max APE	0.5214	4.8739	1.6818
Mean APE	0.1683	0.1908	0.1472
Median APE	0.1706	0.1183	0.1234
Min APE	0.0187	0.0177	0.0481
Root-mean-square deviation	0.1935	0.3122	0.1787
Sum of square error	51.1318	39.0381	119.2799
Standard deviation	0.0957	0.1013	0.2471

Table 1. And figures 6., 7., and 8. show that the proposed setup has some imperfections and that the odometry algorithm chosen does not work perfectly. Rectangular trajectory had the biggest error metrics, while the algorithm worked the best on the straight trajectory which shows that rotations propose a challenge for visual odometry algorithms. From the estimated trajectories visible on the figures above, it is evident that there are some height estimation errors on some parts of the trajectories. This can be corrected by using a barometer sensor for height estimation. Regardless of the estimation errors, the proposed setup would still be adequate for the "Return To Home" use case.

These use cases and different scenarios can easily be tested in the proposed pipeline by switching the photogrammetric 3D model used.

IV. CONCLUSION

This paper presented implementation of a connected pipeline for simulating GPS-denied scenarios and testing UAV localization and navigation algorithms. Proposed system uses ROS2 and ROS1 with a communication bridge between them. Parts of the pipeline are connected in a closed system that is started through a graphical interface. Users have the possibility of changing the odometry algorithm used and providing different 3D models used in simulation. Experiments showed the usability of the pipeline for testing UAV navigation in a GPS-denied environment. This framework can be improved by adding the option of changing algorithm parameters through the graphical interface. Adding the ability to change world physics settings of the simulator through this framework would allow for faster testing of different flying scenarios. Proposed setup uses the ROS1 and ROS2 bridge for communication between the simulator and the odometry algorithm. However, if the user chooses an odometry algorithm that uses ROS2, the structure of this setup should be modified to avoid using a communication bridge.

ACKNOWLEDGMENT

This work has been supported by European Union's Horizon Europe research program Widening participation and spreading excellence, through project Strengthening Research and Innovation Excellence in Autonomous Aerial Systems (AeroSTREAM) - Grant agreement ID: 101071270.

REFERENCES

[1] D. Mourtzis, J. Angelopoulos, N. Panopoulos, "UAVs for Industrial Applications: Identifying Challenges and Opportunities from the Implementation Point of View", *Procedia Manufacturing*, vol. 55, pp. 183-190, Nov 2021.

[2] "Overcoming Signal Interference in GPS Land Surveying", Global GPS Systems. Accessed: Jan. 20, 2024. [Online.] Available: https://globalgpssystems.com/gnss/overcoming-signal-interference-in-gps-land-surveying/

[3] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, A. Ng, "ROS: an open-source Robot Operating System", ICRA Workshop on Open Source Software, Kobe, China, 2009.

[4] S. Macenski et al. “Robot Operating System 2: Design, architecture, and uses in the wild”, *Science Robotics*, vol. 7, May 2022. doi:10.1126/scirobotics.abm6074

[5] “Open Source Autopilot”, PX4 Autopilot. Accessed: Jan. 20, 2024. [Online.] Available: https://px4.io/

[6] “ros_gz”, gazebosim. Accessed: Dec. 20, 2023. [Online.] GitHub repository, Available: https://github.com/gazebosim/ros_gz

[7] M.O.A. Aqel, M.H. Marhaban, M.I. Saripan, et al. “Review of visual odometry: types, approaches, challenges, and applications”, *SpringerPlus*, vol. 5, Oct 2016. doi: https://doi.org/10.1186/s40064-016-3573-7

[8] C. Forster, M. Pizzoli, D. Scaramuzza, “SVO: Fast Semi-Direct Monocular Visual Odometry”, 2014 IEEE International Conference on Robotics and Automation (ICRA), Hong Kong, China, 2014, pp. 15-22, doi: 10.1109/ICRA.2014.6906584.

[9] Robotics and Perception Group, “SVO Pro: Semi-direct Visual-Inertial Odometry and SLAM for Monocular, Stereo, and Wide Angle Cameras”, University of Zurich. Accessed Jan. 20, 2024. [Online.] Available: https://rpg.ifi.uzh.ch/svo_pro.html

[10] “ros2”, ros1_bridge, Accessed: Dec. 20, 2023. [Online.] GitHub repository, Available: https://github.com/ros2/ros1_bridge

[11] Q. Group, “The Future of Digital Experiences”, Qt Group. Accessed: Jan. 20, 2024. [Online]. Available: https://www.qt.io/

[12] “WebODM”, OpenDroneMap. Accessed: Dec. 20, 2023. [Online.] Available: https://www.opendronemap.org/webodm/.