ROSplane 2.0: A Fixed-Wing Autopilot for Research

Ian Reid, Joseph Ritchie, Jacob Moore, Brandon Sutherland, Gabe Snow, Phillip Tokumaru, Tim McLain

***Abstract*—Unmanned aerial vehicle (UAV) research requires the integration of cutting-edge technology into existing autopilot frameworks. This process can be arduous, requiring extensive resources, time, and detailed knowledge of the existing system. ROSplane is a lean, open-source fixed-wing autonomy stack built by researchers for researchers. It is designed to accelerate research by providing clearly defined interfaces with an easily modifiable framework. Powered by ROS 2, ROSplane allows for rapid integration of low or high-level control, path planning, or estimation algorithms. A focus on lean, easily understood code and extensive documentation lowers the barrier to entry for researchers. Recent developments to ROSplane improve its capacity to accelerate UAV research, including the transition from ROS 1 to ROS 2, enhanced estimation and control algorithms, increased modularity, and an improved aerodynamic modeling pipeline. This aerodynamic modeling pipeline significantly reduces the effort of transitioning from simulation to real-world testing without requiring expensive system identification or computational fluid dynamics tools. ROSplane's architecture reduces the effort required to integrate new research tools and methods, expediting hardware experimentation.**

I. Introduction

Unmanned aerial vehicles (UAVs) have gained significant popularity due to applications such as package delivery, photography, surveillance, or advanced air mobility (AAM). Research targeting UAVs often requires ready access to the inner workings of many portions of an autonomy software stack, including estimation, path planning, and high/low-level control. This access is important in all stages of research and development, both in simulation and hardware flight tests.

Researchers often face significant integration problems when it comes to conducting realistic simulations and real-world flight tests. Research software can require extensive rewriting or refactoring to be compatible with realistic simulations or hardware platforms. This increases the difficulty of conducting real-world experiments, which are essential steps in validating and proving out new research algorithms and methods.

ROSflight [1] is a lean, open-source[1] autopilot designed to mitigate these challenges and reduce the barrier to entry for UAV research. It accomplishes this by allowing the same software that runs in simulation to also control the physical vehicle with no changes. Since ROSflight is built on the robot operating system (ROS 2), it offers superior modularity and customizability.

ROSplane [2] is an open-source[2] autonomy stack for fixed-wing UAVs designed to work with ROSflight. A lean feature set means ROSplane offers not only basic functionality, but also enables better understanding for quick and seamless integration of external codebases. Extensive documentation on the algorithms used in ROSplane is available, making it a valuable resource for educational use and research [3], [2].

While ROSplane has received detailed attention in the past [2], recent advances have significantly improved ROSplane's use in advanced UAV research. These improvements reduce the barriers to practical UAV research by enhancing the usability, modularity, and extensibility of ROSplane, especially for hardware experiments. The contributions of this work are to describe the advancements of ROSplane including

- the transition from ROS 1 to ROS 2,
- updated algorithms and modularity for control and state estimation,
- an improved aerodynamic modeling pipeline that significantly reduces the simulated-to-real experiment transition effort.

The rest of this work is organized as follows. Section II discusses other available autopilots and related work. Section III describes the system architecture. The improved aerodynamic modeling pipeline is described in Section IV. Algorithm improvements are discussed in Section V. A tutorial on using ROSplane is provided in VI. Hardware test results and comparison to simulation are shown in Section VII, and then concluding remarks are offered.

II. Related Work

Many excellent autopilots and flight control software are available for hobby, commercial, and research users alike. Current open-source autopilots like PX4 [4] and ArduPilot [5] offer excellent *plug-and-play* capability. They have the advantage of years of development, large communities, and full-featured autonomy stacks. Although they offer distinct advantages, these autopilots have large code bases not designed for easy modification or integration, and can require a researcher to gain familiarity with these large code bases to implement given research. It can also lead to a *black box* environment by obfuscating core autopilot features, making integration of research code difficult and time consuming. Furthermore, code for these autopilots runs primarily on embedded microcontrollers, which makes debugging and active development more difficult and microcontroller dependent.

Even excellent autopilots like Paparazzi that are designed for researchers have limited ability to directly change underlying code. Instead, the project opts for generation of new autopilot modes through XML files. Paparazzi also lacks critical integration that may be necessary for advanced simulation [6].

ROSplane 2.0 offers an autonomy stack that is designed for easy integration with fixed-wing UAV research of any kind. Its lean feature set allows the code to be easily understood, which lowers the integration time and effort for

[1]https://github.com/rosflight

[2]https://github.com/rosflight/rosplane

researchers, thus also lowering the barrier to entry and the learning curve. ROSplane also moves the entirety of the autopilot stack from the embedded microcontroller-based flight control unit (FCU) to the Linux-based companion computer. This facilitates research code development and means that no change of firmware-level code is required, the code is easily modifiable, and transition from simulation to flight is seamless. Pairing this with powerful tools like Docker [7] makes testing and development transitions from vehicle to vehicle effortless. The ROSplane architecture is described more fully in Section III.

The Robot Operating System (ROS 2) has been extensively used for research and commercial use alike [8]. Existing autopilots like PX4 have made significant efforts to support ROS 2 software, and integration with ROS 2 software and PX4 is constantly improving [6]. ROSplane takes ROS 2 support a step further by structuring all autopilot features as ROS 2 nodes, including controllers, estimators, and path planners. This tight integration with ROS 2 significantly improves customizability and modularity.

III. System Architecture

ROSplane's system architecture is designed to accelerate deployment of advanced research by allowing researchers to more easily integrate research code, test in a simulation environment, and deploy on real hardware.

A. Transition from ROS 1 to ROS 2

ROSplane 2.0 has been updated to use ROS 2 instead of ROS 1. Long-term support (LTS) versions of ROS 2 are primarily supported, including ROS 2 Humble on Ubuntu 22.04 and ROS 2 Jazzy on Ubuntu 24.04. Other non-LTS version of ROS 2 are not officially supported.

ROS 2 offers significant advantages over ROS 1, including increased modularity and flexibility for integrating new features, sensors, and code [8]. ROSplane 2.0 builds off the capabilities of ROS 2 by distributing the autonomy stack and the communication paths between independent nodes. This modular structure decouples responsibilities in the code, preventing changes in one module from unintentionally propagating through the control stack, aiding researchers integrating new algorithms. This ROS 2-defined structure also reduces the complexity of integrating novel sensors and external simulators by allowing each component to be added as an independent node. This allows researchers to test new algorithms or hardware without needing to completely restructure logic or interfaces.

B. Autopilot Structure

The default implementation of ROSplane 2.0 follows the framework of [3], as shown in Figure 1.

This architecture gives ROSplane basic waypoint-following functionality that users can use out-of-the-box. Due to the design philosophy of ROSplane, the waypoint-following functionality is neither complex, advanced, nor fully-featured.

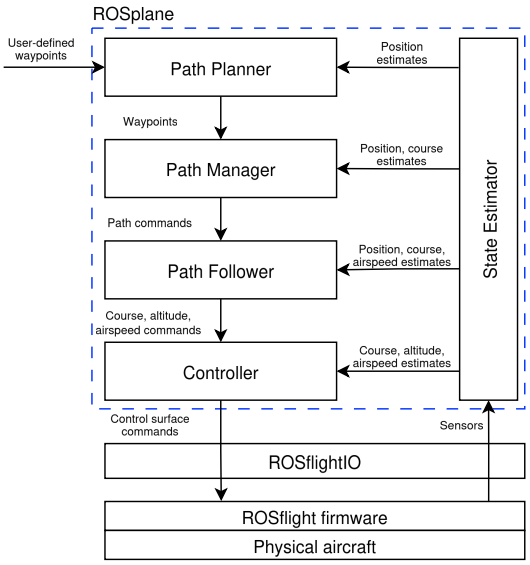

Fig. 1: Cascaded architecture of ROSplane's modules.

Instead, the structure of ROSplane facilitates easy understanding with clearly delineated responsibilities assigned to each module and lean, well-understood implementations. This makes it easier for users to identify which modules are necessary for their specific applications and which modules should be removed, replaced, or modified, thus increasing modularity and customizability while reducing time invested in deployment.

Figure 1 shows that ROSplane follows a cascaded architecture as the output of one module feeds into the next node, with the estimator providing state estimates to each. The responsibility of the path planner module is to generate high-level waypoints that are consumed by the path manager. In the default implementation of ROSplane, the path planner simply aggregates user-defined waypoints and publishes these waypoints to the path manager. The path manager is responsible for managing which waypoints are active by determining when the UAV has achieved a waypoint and what type of path (straight line or orbit) is required to fly to the next waypoint. The path manager also determines if the paths followed are Dubins paths, and calculates the correct path parameters. The path manager then sends path definitions to the path follower node, whose responsibility is to follow these path commands by generating lower-level controller commands. These lower-level commands are composed of desired airspeed, altitude, and course setpoints, which are used by the controller module to determine the desired control surface deflections and the throttle setpoint. ROSplane's estimator module receives sensor information through the ROSflightIO node (a node responsible for managing communication between the companion computer and the FCU, as discussed in [1]). It is responsible for constructing a state estimate, which is then sent to all other modules. A more detailed description of each module is found in [3].

Each of these modules is implemented as a ROS 2 node and all communication over the modules occurs over the ROS 2 network.

IV. Simulation to Real World Workflow

This section describes an aerodynamic modeling pipeline that significantly improves the simulation-to-real transition. The aerodynamic model used is described in [3].

Tuning a controller or attempting new research on an aircraft for the first time during a physical flight test can be dangerous because the aircraft's performance has not been validated. Unstable responses are possible and potentially catastrophic. Tuning or testing new capabilities in simulation before flight is preferable because it avoids these risks and demonstrates real-world performance; however, an accurate aerodynamic model and simulation are necessary. Unfortunately, accurate aerodynamic models of UAVs are difficult to obtain with traditional methods, as system identification is time-consuming, expensive, and often requires extensive flight data augmented with significant wind tunnel testing. Rigorous computational tools (e.g. computational fluid dynamics) are also available to help with system identification, but they are often costly, slow, and require significant researcher effort. These challenges make it difficult for researchers to test new autonomous aircraft.

The ROSplane simulation is capable of providing accurate controller tuning with a lower-fidelity aerodynamic model that can be obtained using open-source aircraft modeling software. Documentation on the aerodynamic modeling pipeline and ROSplane integration steps are available on the project website[3].

Several open-source software tools for aerodynamic modeling and analysis are publicly available. Notable examples include XFLR5 [9] and OpenVSP [10], both parametric aircraft modeling and aerodynamic analysis programs. Both tools enable users to define aircraft geometry and conduct a range of aerodynamic analyses to acquire aerodynamic parameters and stability and control derivatives [11]. The steps involved in the aerodynamic modeling pipeline are outlined in the following flowchart.

When modeling and analyzing an aircraft, users should pay careful attention to ensure that accurate aircraft dimensions, flight conditions, and trim-state conditions are used. Once the aircraft model is complete, the aerodynamic parameters and stability and control derivatives can be obtained by performing an aerodynamic analysis and a stability analysis for each set of control surfaces. The aircraft model can then be integrated into the ROSplane simulation environment by inputting all of the aerodynamic coefficients into an aerodynamic parameters file. After loading the model in simulation, the researcher can then tune both the aircraft model and the flight controller. Both XFLR5 and OpenVSP have been shown to provide sufficiently accurate aircraft models for tuning flight controllers in close to trim-state conditions; however, some errors in the model

[3]rosflight.org

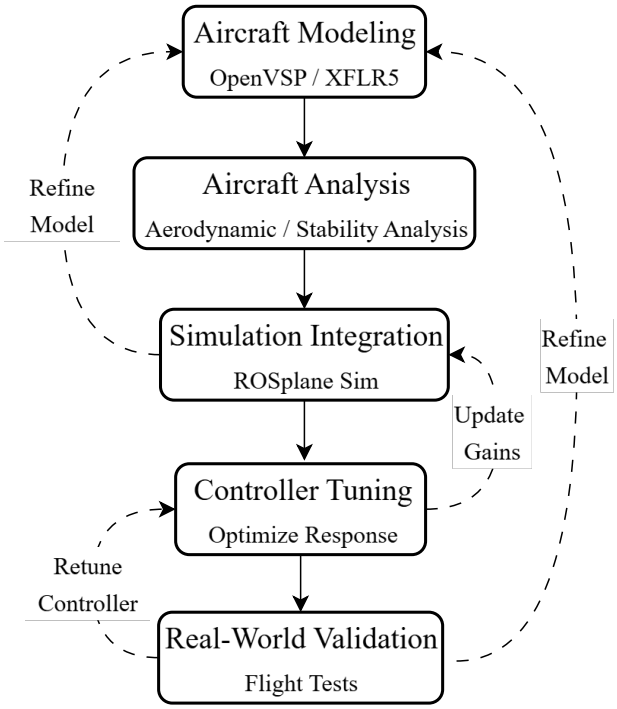

Fig. 2: Aircraft integration pipeline with iterative feedback.

can be present and will likely lead to inaccurate simulation performance [12].

Before tuning a flight controller, users should validate the accuracy of their aerodynamic model by manually flying the aircraft in the simulation. If the aircraft has unrealistic behavior, then the aircraft model may require multiple cycles of improvement. Users can often improve their aircraft model by eliminating errors in meshing, adding additional details (contours and smaller features), and increasing the precision of dimensions while comparing against experimental results. After several cycles of iteration, both XFLR5 and OpenVSP models should provide sufficient fidelity to support effective pre-flight controller tuning that is safer than tuning the controller blindly on a maiden flight.

To tune the flight controller, users can set the flight controller to control the aircraft and observe the controller responses to a variety of waypoint, trajectory, rate, and position commands. Users can then modify any controller gains during flight through ROS terminal commands, or the RQT GUI, until the controller behavior is sufficient. Once the flight controller is tuned to provide appropriate responses in simulation, it is well-prepared to safely provide control during a physical flight.

This approach reduces tuning risk, accelerates controller development, and promotes reproducible UAV research without requiring expensive wind-tunnel testing or high-end computational resources.

V. Modularity and Algorithmic Improvements

ROSplane 2.0 has improved the modularity of the system by making useful functionality more accessible to the end user. The state estimator and controller have been restructured with

clearly defined interfaces and internal structure to enable safe and effective modifications without a complete overhaul. Both the the controller and estimator have also had substantial algorithmic changes that improve performance. The algorithmic changes follow the most recent changes reflected in [3]. They represent improved accuracy and robustness in the attitude and velocity state estimates, along with improved path following.

A. State Estimation

Previously, ROSplane used a pedagogically motivated two-stage estimation scheme. It separated attitude estimation and positional states. This made it easier for students to implement [2]; however, the estimator was less accurate than desired. In ROSplane 2.0, the estimator has been replaced with a full-state extended Kalman filter (EKF). The EKF has also expanded the number of states being estimated, including heading, gyroscope biases and lateral velocity. The estimator also takes full advantage of GNSS velocity measurements and measurements from the magnetometer. Figure 3 compares the ROSplane and ROSplane 2.0's estimators on simulated data. The EKF has significantly less tracking error in attitude, velocity and position, and demonstrates that it provides consistent and reliable estimation to support the flight experiments of researchers.

The EKF has also had modularity improvements. It is designed to easily accommodate new measurement updates, allowing the researcher to easily integrate new sensors. The framework for adding new sensors can be found on the project website.

B. Control

The controller architecture for ROSplane 2.0 has also been improved. Previously, the altitude was controlled using a state machine to command a climb rate and descent rate if outside of a band around the cruising altitude. The altitude controller has been unified into a successive loop closure controller, improving altitude response. ROSplane 2.0 also integrates the improvements in orbit following. A feedforward term is added to commanded roll angles to allow for faster and tighter convergence on orbit paths. In addition, ROSplane 2.0 makes available a total energy controller for use. Section VII shows the performance of the successive loop closure controller in real flight conditions.

The controller's modularity improvements allow for safe integration of new control schemes. This includes utilizing a state machine to ensure that full autonomous control is only taken at safe altitudes. ROSplane 2.0 has a simple state machine to allow for the controller to have different control schemes based on the altitude and phase of flight. These are summarized in Figure 4. Integration with these phases of flight and the modular nature of the controller maximize flexibility for the researcher to integrate research by allowing them to take as much or as little control as they need.

VI. Tutorial

This section contains a brief tutorial on how users can expect to use ROSplane to aid their research. This is not a

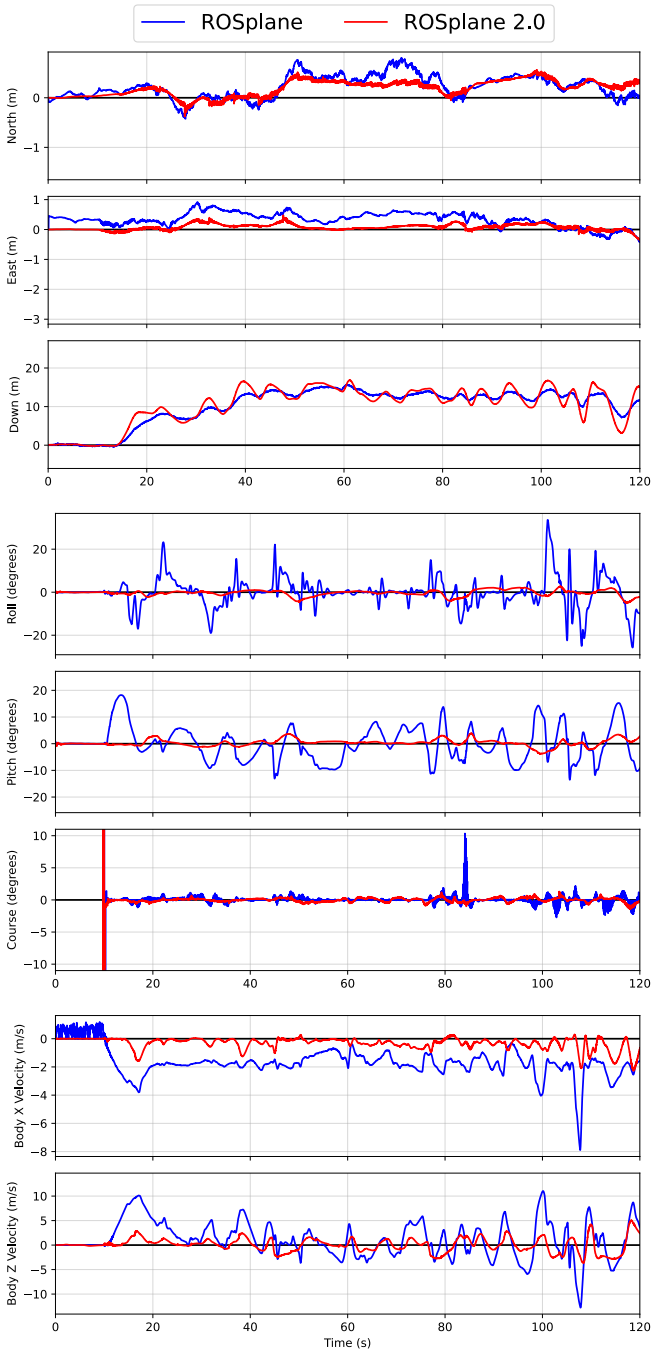

Fig. 3: Comparison of ROSplane and ROSplane 2.0 estimated states on a takeoff and path following sequence in simulation.

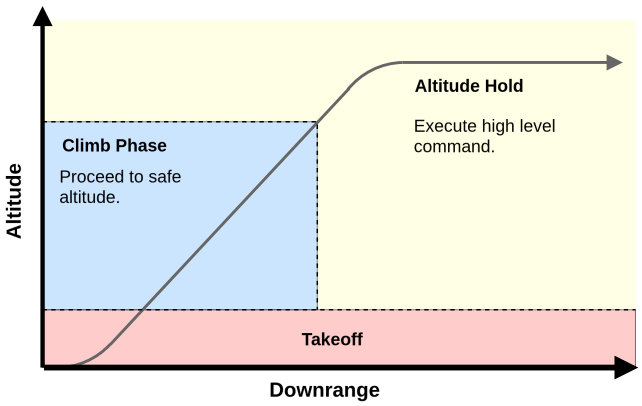

Fig. 4: The state machine for different phases of flight.

comprehensive tutorial, meaning that the ideas listed here serve as a starting point. For a more detailed description please see the project website. We will first describe ROSplane's default functionality, and will then give examples of how to customize ROSplane to fit individual needs. Finally, we will discuss the intended workflow for researchers to use ROSplane in their research.

A. Default High-level Functionality

This section describes ROSplane's basic functionality that can be used out of the box. By default, ROSplane enables an aircraft to fly waypoint missions. These waypoints are simply 3D coordinates, and can be specified in latitude, longitude, and altitude (LLA), or in meters north, east, and down (NED) from the initial position of the aircraft. These waypoint types can be mixed and matched—for example, one waypoint can be specified in LLA format and the next can be specified in NED format. Waypoints are loaded using service calls to ROSplane's path planner module, as shown in Figure 1, and can be loaded one at a time, or all at once as specified in a YAML file.

ROSplane's default path planner module can be used by high-level planning or coordination algorithms. For example, a high-level exploration module for search-and-rescue might specify waypoints that an aircraft needs to fly, or a multi-agent coordination algorithm might specify a rendezvous point as a single waypoint. These algorithms may not need more complex path-following than the straight lines and orbits in ROSplane. In this case, the default ROSplane functionality could effectively sit underneath a researcher's high-level algorithm.

B. Customizing ROSplane

ROSplane's basic waypoint-following capability will not satisfy all researchers' needs. Because of this, ROSplane has been designed so that each module has a single, specific responsibility, as shown in Figure 1. This separation of responsibility makes it easy to determine which modules can be removed or modified and which modules should be kept for a given application. Additionally, ROSplane exploits the modular nature of ROS 2 to allow these modules to be easily modified without affecting the rest of the autonomy stack.

Each module in ROSplane uses an interface class to define each module's ROS 2 interfaces, as well as the required functions that a derived node must implement. If a given derived class successfully inherits from the base interface class, it will integrate seamlessly with the rest of ROSplane. For example, the ROSplane path follower module uses a vector field approach from [3] to follow straight-line or orbit paths that are specified by the path manager module. Changing the way ROSplane follows these straight-line and orbit paths can be done by simply creating a node that has the same ROS 2 interfaces as the original path follower. This can be done by either inheriting from the base class in the node, or reimplementing the ROS 2 interfaces if inheriting is not possible (e.g. if using Python instead of C++).

Many times, users will desire to customize ROSplane in a way that crosses the boundaries of ROSplane's default structure. For example, a researcher studying different spline-following methods would need to modify ROSplane so that it follows splines instead of straight lines and orbits. Thus, both the path manager and path follower modules would need to be replaced. The linear flow of information through ROSplane makes this easy—as long as the replacement module has the same input ROS 2 interfaces (subscribers and service servers) as the path manager and the same output ROS 2 interfaces (publishers and service clients) as the path follower, the rest of ROSplane will seamlessly work with the new application code.

ROSplane has been designed to be as flexible as possible so that modules can be replaced, removed, or modified as needed for a particular application.

C. Intended Workflow for ROSplane

This section describes the intended workflow for users interested in ROSplane. The first step is for a researcher to develop their application-specific code and integrate it into the ROSplane autonomy stack. Then, the aircraft used in hardware flight tests should be modeled as described in Section IV. If hardware tests will not be performed, then the default aerodynamic model can be used. It is recommended that users then use the ROSflight simulation to test and further refine the application code. The project website has detailed information on setting up the simulation environment. Once the application code works well in simulation, hardware tests can be performed. Because ROSflight and ROSplane run the exact same way in simulation and in hardware [1], these hardware tests can be performed in the exact same manner as the simulation tests were performed.

VII. Results

This section presents the results of hardware and simulation experiments demonstrating the basic functionality ROSplane offers to users out-of-the-box. These results also demonstrate that ROSplane closes the simulation-to-real gap for research.

In each test, the aircraft was directed to fly to a series of four waypoints in an hourglass formation on a continuous circuit.

A. Hardware

To validate the performance of ROSplane on standard UAV hardware, several maneuvers were flown that tested the performance of the controller and estimator. The desired path and the performance can be seen in Figure 5. The path following algorithm overshoots the turns of the path because the commanded path assumes that the aircraft is only acceleration-limited and can achieve its maximum bank angle instantaneously. Figure 6 shows that the aircraft was able to respond to given commands effectively. Times of poor response in the pitch loop are seen to be coincident to periods of high roll command. It is important to note that the oscillations seen in the roll response have a frequency on the order of four seconds and are in response to deviations of the course from the commanded and are not due to instability in the roll controller. The amplitude of the roll response oscillations is small, typically less than 15 degrees. Similarly, the course oscillations have amplitudes less than 10 degrees.

The altitude tracking often dips well below and above the actual target. This is because as the aircraft banks, the effective lift is decreased, and the pitch control loop takes time to compensate. The aircraft overshoots its target altitude as well. This error is likely contributed to by pressure effects from the barometer measuring cabin pressure rather than true atmospheric, since the FCU is housed inside the fuselage of the aircraft. This is manifest as a decreasing error between the RTK and estimated down position as the aircraft decreases in altitude.

The estimator accurately estimates the position of the aircraft and yields an RMSE from RTK in the north east position estimation of 1.93 meters. The total RMSE error for the estimated path is 3.64 meters. Though no ground truth was available for the other states, the estimator gives reasonable and expected outputs. The estimation of these states can be seen in Figures 5 and 6.

These results show that ROSplane is an effective platform for research. Researchers can implement desired algorithms easily to evaluate different control, estimation, trajectory planning and following algorithms. Full discussion and analysis of the controller and rationale for its performance can be found in [3].

B. Simulation to Real Flight

The simulation capabilities of ROSplane and the aerodynamic modeling pipeline were tested according to the aircraft integration process outlined in Section IV. The responses of the controller in simulation were then compared to those recorded during physical flight, given the same trajectory commands.

A model of an RMRC Anaconda UAV aircraft was created in both XFLR5 and OpenVSP, and a variety of aerodynamic and stability analyses were performed to obtain the aerodynamic coefficients and stability and control derivatives for the

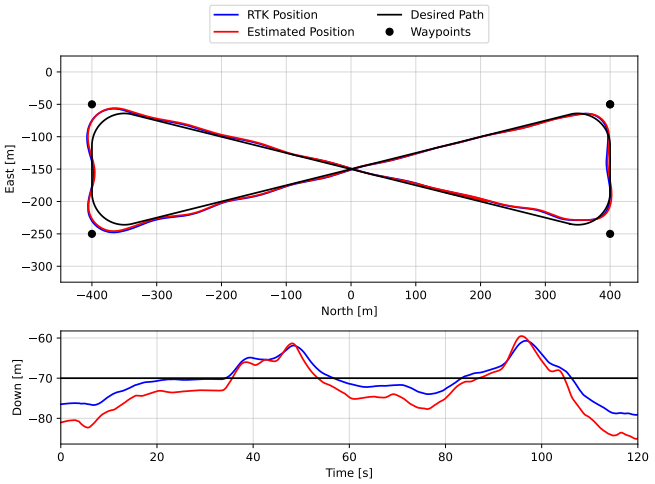

Fig. 5: Comparison of ROSplane estimated position and RTK GNSS recorded position, alongside commanded path.

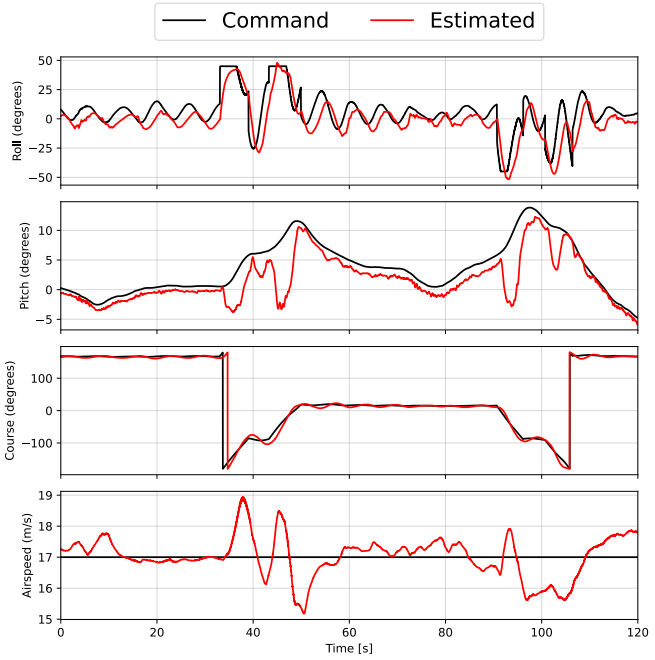

Fig. 6: The controller response to given flight path input in the airspeed and attitude angles.

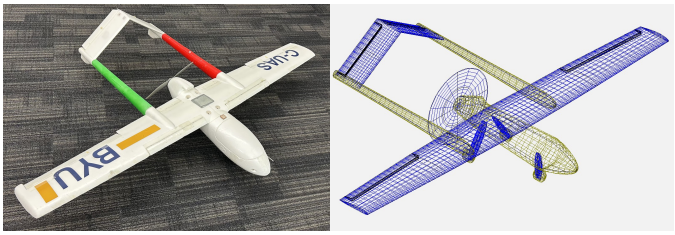

Fig. 7: RMRC Anaconda UAV (left) with digital OpenVSP model used for aerodynamic analysis (right).

aircraft. XFLR5 and OpenVSP both produced aerodynamic models that were accurate enough for controller tuning, though each tool excelled in different areas. XFLR5's built-in trimming capability made stability analyses straightforward and yielded more consistent stability derivatives. OpenVSP, by modeling the full aircraft geometry, provided more reliable drag estimates and control derivatives. For both the Xflr5 and OpenVSP models, several cycles of iteration, as outlined in Section IV, were required for the aircraft model to converge on realistic performance. The final performance of the simulation and physical flight was comparable. Note that the same controller gains were used in both flights, meaning that there was no difference in ROSplane's controllers for this comparison. Figure 8 shows the OpenVSP simulation results compared with several physical flight test results. The Xflr5 results were similar.

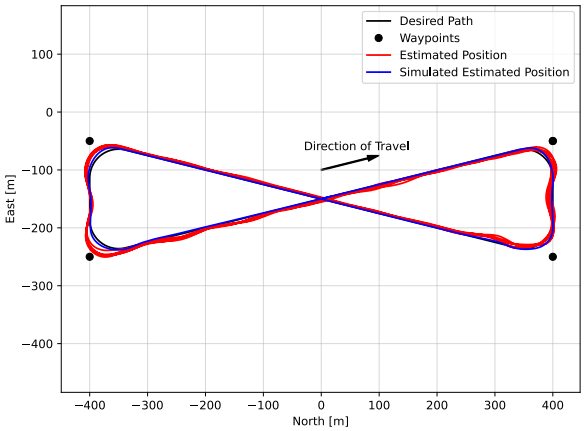

Fig. 8: Comparison of a commanded flight path with OpenVSP VLM-derived simulation and several full flight paths of physical comparison results.

These comparisons validate that new UAVs can be modeled and analyzed using open-source software, integrated into the ROSplane simulation, and used for controller tuning, mission planning, and research testing. Iterative testing and improvement of these models provided confidence that real-life system performance can be accurately predicted within a safe simulation environment.

VIII. Conclusion

ROSplane's structure, capabilities, and modularity provide researchers with more flexibility and lower the barriers to entry to perform UAV autonomy research. The improved code structure and architecture provide greater customizability and opportunities for integrating external hardware and code. ROSplane 2.0 architecture allows researchers to quickly and safely tune flight controllers for new fixed-wing UAVs in simulation to prepare for physical flight tests. This functionality of ROSplane 2.0 was demonstrated by modeling a UAV aircraft using open-source aircraft modeling and analysis software, integrating this aircraft model into the ROSplane simulation, and validating that the closed-loop controller performance of the simulation and physical flight are comparable. This enables researchers to perform accurate controller tuning in simulation before flight, thereby reducing risk and iteration time. The universal applicability of the ROS 2 structure, paired with proven functionality and an effective simulation platform, makes ROSplane a powerful tool for researchers valuing customizability and direct control in cutting-edge autonomous research applications.

References

[1] J. Jackson, D. Koch, T. Henrichsen, and T. McLain, "ROSflight: A lean open-source research autopilot," in *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2020, pp. 1173–1179.

[2] G. Ellingson and T. McLain, "ROSplane: Fixed-wing autopilot for education and research," in *2017 International Conference on Unmanned Aircraft Systems (ICUAS)*, 2017, pp. 1503–1507.

[3] R. W. Beard and T. W. McLain, *Small Unmanned Aircraft: Theory and Practice*, 2012. [Online]. Available: uavbook.byu.edu

[4] L. Meier, D. Honegger, and M. Pollefeys, "PX4: A node-based multithreaded open source robotics framework for deeply embedded platforms," in *2015 IEEE International Conference on Robotics and Automation (ICRA)*, 2015, pp. 6235–6240.

[5] ArduPilot. (2025) Ardupilot. [Online]. Available: https://ardupilot.org/

[6] N. Aliane, "A survey of open-source UAV autopilots," *Electronics*, vol. 13, no. 23, 2024. [Online]. Available: https://www.mdpi.com/2079-9292/13/23/4785

[7] D. Merkel, "Docker: lightweight linux containers for consistent development and deployment," *Linux J.*, vol. 2014, no. 239, Mar. 2014.

[8] S. Macenski, T. Foote, B. Gerkey, C. Lalancette, and W. Woodall, "Robot operating system 2: Design, architecture, and uses in the wild," *Science Robotics*, vol. 7, no. 66, p. eabm6074, 2022. [Online]. Available: https://www.science.org/doi/abs/10.1126/scirobotics.abm6074

[9] Xflr5. (2025) Xflr5. [Online]. Available: https://www.xflr5.tech/

[10] OpenVSP. (2025) OpenVSP. [Online]. Available: https://openvsp.org/

[11] D. F. K. Mustafa Özdemir, "Model based aircraft design and optimization: a case study withcessna 172n aircraft," *Engineering Research Express*, vol. 5, pp. 1–22, 2023. [Online]. Available: https://www.researchgate.net/publication/373635436

[12] S. H. Fumiaki Hiraharaa, Kandai Uchidab, "Influences of aerodynamic parameter uncertainties for UFSC based flight trajectory generation," *SICE Journal of Control, Measurement, and System Integration*, vol. 16, no. 1, pp. 363–372, 2023. [Online]. Available: https://www.tandfonline.com/doi/full/10.1080/18824889.2023.2273608