ROSflight 2.0: Lean ROS 2-Based Autopilot for Unmanned Aerial Vehicles

Jacob Moore[1], Phil Tokumaru[2], Ian Reid[1], Brandon Sutherland[1], Joseph Ritchie[1], Gabe Snow[1], Tim McLain[1]

***Abstract*— ROSflight is a lean, open-source autopilot ecosystem for unmanned aerial vehicles (UAVs). Designed by researchers for researchers, it is built to lower the barrier to entry to UAV research and accelerate the transition from simulation to hardware experiments by maintaining a lean (not full-featured), well-documented, and modular codebase. This publication builds on previous treatments and describes significant additions to the architecture that improve the modularity and usability of ROSflight, including the transition from ROS 1 to ROS 2, supported hardware, low-level actuator mixing, and the simulation environment. We believe that these changes improve the usability of ROSflight and enable ROSflight to accelerate research in areas like advanced-air mobility. Hardware results are provided, showing that ROSflight is able to control a multirotor over a serial connection at 400 Hz while closing all control loops on the companion computer.**

I. Introduction

In recent years, interest in unmanned aerial vehicles (UAVs) has increased significantly. Technological advances have enabled numerous applications of UAVs, including package delivery, photography, search-and-rescue, firefighting, as well as military applications.

Advanced air mobility (AAM), a category broadly referring to increasing autonomy in urban areas for civilian use, is also currently an area of high interest. AAM aircraft often take the form of eVTOL aircraft, and specialized autopilots, algorithms, and hardware are needed to effectively conduct research in this field. Because of this, researchers often need access to the inner workings of an autopilot (e.g., the state estimator or inner loop controller), which makes commercial closed-source autopilots or many open-source autopilots difficult to use. Additionally, simulation and hardware experiments are critical in AAM research to ensure proposed systems and methodologies are safe and function as intended.

ROSflight is a lean, open-source autopilot for UAVs built for research. Because it is designed to be lean, ROSflight is not full-featured and does not boast many of the state-of-the-art functions available in other popular open-source autopilots [1], [2]. Instead, ROSflight offers only the basic functionality needed to support UAV research, prioritizing understandability. While this places more responsibility on the end user to develop application-specific code, we believe a lean architecture reduces the *black-box* nature of the autopilot, thus reducing the total effort to implement a user's application code. Additionally, the ROSflight project has a strong emphasis on clear code and complete documentation, which improves accessibility and lowers the barrier to entry for researchers. Documentation can be found on the project website, rosflight.org.

Built on the robot operating system (ROS 2) [3], ROSflight is designed with a modular architecture to fit the needs of varying airframes and applications. ROSflight is designed to enable true software in the loop (SIL) simulation. When using ROSflight, the same code that runs the autopilot in simulation also flies the vehicle in hardware, with no changes—significantly enhancing the transition from simulation to hardware experiments.

ROSflight has received detailed treatment in [4], [5]. We rely on [5] for an excellent description of the overall architecture and design goals of the ROSflight project. The contributions of this work are to describe significant improvements to ROSflight in the release of ROSflight 2.0 including

- the transition from ROS 1 to ROS 2,
- improved modularity and accuracy in the actuator mixing,
- new supported hardware for faster and more reliable operation,
- a restructured and modular simulation environment to support diverse simulation needs, and
- flight test results demonstrating these improvements in hardware.

Due to these advancements, ROSflight 2.0 improves the modularity and usability of the software, enabling ROSflight to accelerate research in areas like advanced air mobility.

The rest of the paper is organized as follows: Section II describes work similar to ROSflight. A brief overview of ROSflight is described in Section III Improvements to the ROSflight architecture are described in detail in Section IV, and supported hardware and improvements to the simulation environment are discussed in Section V. Hardware results demonstrating these improvements to ROSflight are described in Section VI and we conclude in Section VII.

II. Related Work

Advances in UAV technology have given rise to many excellent and mature open-source autopilots like PX4 [1] and ArduPilot [2]. Both of these autopilots offer impressive feature sets, state-of-the-art performance, extensive community support and adoption, and *plug-and-play* functionality. Additionally, both projects have good support for the Robot Operating System (ROS 2), thus improving integration of external code libraries into an autopilot [6]. However, the large code bases, complexity, and steep learning curve of these autopilots decrease understandability, which increases

[1]Brigham Young University
[2]AeroVironment Inc.

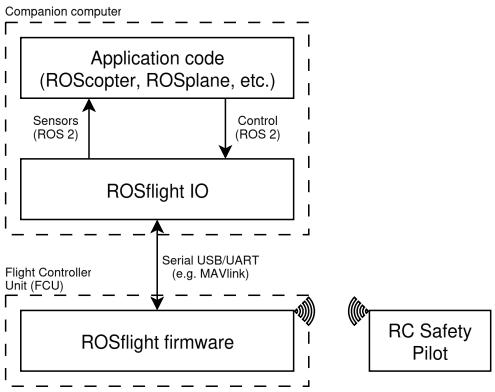

Fig. 1: Overview of the ROSflight archtecture.

the time and effort required to integrate external code and conduct simulation and hardware tests. ROSflight is primarily designed to be understandable with clean, lean code and complete documentation. Additionally, ROSflight moves most of the autopilot stack to a Linux-based companion computer which has more compute power and is easier to develop on than an embedded microcontroller. ROSflight also emphasizes true software-in-the-loop simulation [5]. Other open-source autopilots [7], [8] are less mature, have limited flexibility and ROS 2 support, or are designed primarily for hobbyist use. ROSflight is built around ROS 2 and is explicitly designed to fill researchers' needs.

III. ROSFLIGHT OVERVIEW

An overview of the ROSflight architecture is shown in Figure 1. At a high level, it is comprised of two onboard systems: a low-level embedded microcontroller (called the flight control unit) and a Linux-based companion computer (also called the flight computer).

Flight control unit (FCU): The FCU interfaces directly with the physical aircraft and communicates with a Linux companion computer using a high-speed UART or USB CDC ACM serial interface. This FCU runs the ROSflight firmware, and is responsible for collecting sensor measurements (a *sensor aggregator*), controlling actuators, streaming information to the companion computer, and monitoring the state of the software (e.g. armed or disarmed). The FCU also receives radio control (RC) commands directly from a safety pilot via an RC receiver. An estimator and controller are present on the firmware to manage the fast inner loops of the control architecture (e.g., angle and rate loops for a multirotor). Note that while the vehicle is fully controllable by the safety pilot without the companion computer, the majority of the autonomy stack is found on the companion computer, so only limited autonomy is available without it.

Companion computer: The companion computer, running Linux, receives sensor measurements and sends actuator commands to the FCU over the serial interface. The serial protocol is abstracted in such a way that different serial protocols can be used, and MAVlink is used as the default serial communication protocol. Most of the autonomy stack, including high-level state estimation and control algorithms, resides on this computer. All modules on the companion computer are written as ROS 2 nodes, enhancing the modularity of the system. When connected on the same network using standard WiFi or another network, the companion computer can communicate with other computers like a ground-station laptop so users can easily monitor the system.

The subject of this work is to describe the significant improvements to both the hardware and software associated with the FCU. Examples of out-of-the-box high-level autonomy stacks using ROSflight are available with ROSplane[1] [9] and ROScopter[2].

IV. DESIGN UPDATES

A. ROS 1 to ROS 2

Previous versions of ROSflight used ROS 1, which is no longer supported by the maintainers of ROS. ROSflight now uses ROS 2, and primarily supports long-term support (LTS) versions of ROS 2, which are ROS 2 Humble with Ubuntu 22.04 and ROS 2 Jazzy with Ubuntu 24.04. ROS 2 offers several improvements over the previous ROS 1 system, making it more robust and reliable [3]. The core ROSflight firmware on the embedded microcontroller does not use ROS (e.g. micro-ROS [10]), and instead communicates with the ROS 2-based I/O node (called ROSflightIO) on the companion computer over serial using MAVlink. The ROSflight simulation is built around ROS 2, and exploits this to improve modularity as will be discussed in Section V-B.

B. Mixer

In ROSflight 2.0, the mixer has been restructured to offer greater flexibility and control. The mixer is a component in the software responsible for taking in a controller command and transforming it to actuator outputs. This process is called *actuator mixing* or *force allocation*. Given a desired command, the mixer allocates actuator effort to achieve it as closely as possible. The inputs to the mixer are the output of a controller, as shown in the block diagram in Figure 2. Similar to [11], we formulate a linear mapping between the inputs and outputs using a mixing matrix, which takes the form in Equation (1).

$$\tau = M^{\dagger} u, \tag{1}$$

where $\tau \in \mathbb{R}^n$ is a vector of commands that is eventually written to the actuators (after any necessary scaling and saturation operations), $M \in \mathbb{R}^{m \times n}$ is the mixing matrix, $u \in \mathbb{R}^m$ is the vector of controller commands, and $(\cdot)^{\dagger}$ is the Moore-Penrose pseudoinverse. In ROSflight, u is composed of either RC safety pilot commands or *offboard commands* from the companion computer (offboard with respect to the FCU), as described in [5].

In ROSflight, we let $m = 6$ and $n = 10$, meaning there are ten output channels corresponding to hardware outputs (PWM or DShot) on the FCU, and six desired commands to

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

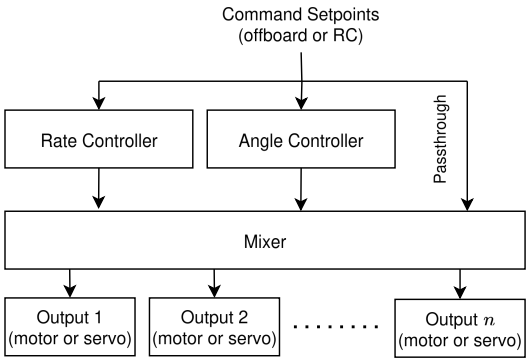

Fig. 2: Diagram of how the ROSflight mixer interfaces with other ROSflight modules. The mixer takes in either the output of a controller or setpoints that bypass the controller, called *pass-through* commands, and outputs actuator signals. The setpoints are created by the onboard computer or the RC safety pilot.

the mixer, requiring 60 values. Additionally, the ROSflight mixer defines a PWM rate and an output type (motor, servo, GPIO, or auxiliary) for each output channel, yielding an additional 20 values. Note that the actual number of PWM output pins is limited by the FCU hardware. If a hardware board has more than ten PWM output pins, the remainder are automatically defined to be auxiliary outputs, which are simply PWM outputs that do not pass through the mixer.

As in [11], u is typically a vector of the desired forces, F, and torques, τ, that are to be applied to the physical system. Thus, in ROSflight, we let $u = [F^T, \tau^T]^T \in \mathbb{R}^6$ as the default interpretation of the mixing equations. However, the actual meaning of the mixer input vector u is closely tied with the elements of the mixer matrix. Thus, in the following descriptions, we do not constrain the input vector u to be a vector of forces and torques, but instead a vector of six generic control inputs, all of which may not be used in a given setting. For example, a quadrotor mixer may take in desired forces and torques (F_z, Q_x, Q_y, Q_z), corresponding to four of the six degrees of freedom of the physical aircraft, and return individual motor commands, while a V-tail fixed-wing mixer may take in desired control surface commands $(\delta_a, \delta_e, \delta_t, \delta_r)$ and return actual servo commands. An example of Equation (1) for a V-tail fixed-wing mixer is

$$\begin{bmatrix} \delta_a \\ \delta_{lr} \\ \delta_{rr} \\ \delta_t \\ 0_{6\times1} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0_{1\times2} \\ 0 & -0.5 & 0.5 & 0 & 0_{1\times2} \\ 0 & 0.5 & 0.5 & 0 & 0_{1\times2} \\ 0 & 0 & 0 & 1 & 0_{1\times2} \\ 0_{6\times1} & 0_{6\times1} & 0_{6\times1} & 0_{6\times1} & 0_{6\times2} \end{bmatrix} \begin{bmatrix} \delta_a^c \\ \delta_e^c \\ \delta_r^c \\ \delta_t^c \\ 0_{2\times1} \end{bmatrix},$$

where $u = [\delta_a^c, \delta_e^c, \delta_r^c, \delta_t^c, 0_{1\times2}]^T$ is the input vector containing the desired aileron, elevator, rudder, and throttle commands, respectively, and $\tau = [\delta_a, \delta_{lr}, \delta_{rr}, \delta_t, 0_{1\times6}]^T$ is the output vector containing the commands to the servos controlling the ailerons, left ruddervator, right ruddervator, and throttle, respectively. Since in this example the remaining two control inputs in u are not used, they are zero. Note that the signs of the coefficients in the mixing matrix are dependent on the orientation of the control-surface servos on the aircraft.

Since the values of the elements of a mixing matrix are highly dependent on the physical airframe and the output of the controller, the ROSflight mixer has been re-designed to offer additional flexibility. Key features allow users to

- use predefined mixers for commonly used aircraft,
- define a custom mixer loaded at runtime,
- specify a separate mixer for the RC safety pilot and the onboard computer, and
- post-process the output vector τ with empirical motor parameters for a higher-fidelity model.

1) Predefined Mixers: Some predefined, hard-coded mixers are included for convenience in the implementation of the ROSflight firmware. These mixers can be selected at runtime via parameters, and are available for standard airframes, including quadorotor, hexarotor, standard fixed-wing, or V-tail fixed-wing airframes. A full list of predefined mixers is available on rosflight.org. Each mixer definition defines the type of each output channel (e.g., motor, servo, GPIO, or auxiliary), the PWM rate assigned to each channel, and the values of either M or $M^\dagger$. See Appendix A for the derivation and assumptions of the predefined mixers.

2) Custom Mixer: Since the mixer is intimately tied to the physical geometry of the aircraft, users may need mixers not included in the predefined mixers hard-coded in ROSflight. Thus, ROSflight includes the ability for users to load their own custom mixing matrix values via parameters. A custom mixer must include both the 60 mixing matrix parameters as well as the 20 header parameters. The names of these parameters and default values can be found on rosflight.org.

A custom mixer grants additional flexibility to ROSflight, allowing users to employ a higher fidelity model than the predefined mixers or to design mixers for nonstandard aircraft (e.g., eVTOL) without having to reflash the firmware.

3) Pass-through Mixer: ROSflight can be operated in *pass-through* mode, where offboard commands bypass the firmware controller and progress directly to the mixer, as shown in Figure (2). The high-speed serial connection enables sending these pass-through offboard commands at a high rate, thus giving direct access to individual motors at a high rate. Such a configuration enables low-level control of a vehicle from the companion computer, thus improving flexibility and lowering the barrier to entry since all control loops can be closed on the Linux-based companion computer. Use cases might include sending direct actuator commands for individual motor control during transition of an eVTOL, or for sending actuator commands from the output of a neural network controller [12], [13]. Additionally, operating ROSflight in pass-through mode enables users to filter the inputs to the mixer. This could be used to simulate hardware failures [14], cyber attacks [15], or other physical failures simply by filtering the companion computer's output before sending to the mixer.

4) Primary and Secondary Mixers: The addition of a custom mixer can raise issues for a safety pilot. For example, a quadrotor with the identity matrix as the mixing matrix

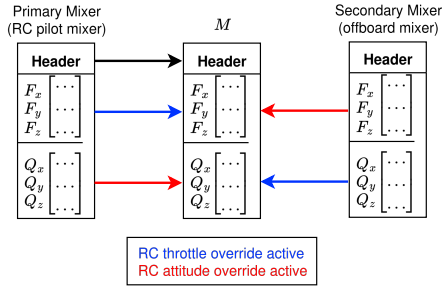

Fig. 3: Diagram of how the ROSflight mixer, M, is constructed from the primary and secondary mixers based on which RC overrides are active. Note that the header information (PWM rate and output type) for M is always constructed from the primary mixer's header information.

would allow the companion computer to send actuator commands directly to each of the four motors. This scenario, however, would make it impossible for a safety pilot to fly the quadrotor, because the rate and angle controllers used by the RC pilot output forces and torques, not direct actuator commands, as shown in Figure 2.

To address this safety concern, ROSflight supports loading two separate mixers: the primary mixer and the secondary mixer. The primary mixer is used by the RC safety pilot and thus should always be defined such that the RC pilot can control the vehicle, while the secondary mixer is used by the companion computer. When the companion computer has control over the vehicle, ROSflight uses the secondary mixer; otherwise, it uses the primary mixer.

When conducting flight tests with a companion computer, it can be helpful to isolate certain channels of the offboard command to incrementally test companion computer control. To accommodate this, the RC pilot in ROSflight has two types of control over the vehicle: attitude override and throttle override. When RC attitude override is enabled, the RC pilot has control over the Q_i portions of the command vector u. When RC throttle override is enabled, the RC pilot has control over the F_i portions of the command vector. When either override is enabled, the mixer that transforms the control input vector $u = [F_x, F_y, F_z, Q_x, Q_y, Q_z]^T$ to motor outputs τ is a mixture of the primary and secondary mixers, as shown in Figure (3). This allows the RC pilot to control either the throttle or the attitude while the companion computer controls the other. Note that this requires that the mixing matrices and the control inputs are designed so that this split may occur. In other words, if the mixer output does not have meaning when the torque commands Q_i are separated from the force commands F_i, then attempting to use attitude and throttle override independently of each other may cause undesirable behavior. If that is the case, setting the attitude and throttle override RC channels to the same value (thus activating both or neither) would lead to the correct interaction between mixers.

Both the primary mixer and the secondary mixer should be selected by setting the appropriate parameter through the ROSflight parameter interface. If the secondary mixer is left unspecified, it will default to the same value as the primary mixer. This is useful for users who do not need a distinction between the primary and secondary mixers.

5) Empirical Motor Parameters: As described in Appendix , using the motor and propeller parameters can increase the fidelity of the mixing matrix. In ROSflight, the `USE_MOTOR_PARAM` parameter tells the firmware whether or not the mixer directly outputs motor setpoints, $\delta_{t,i}$, or desired angular speeds, Ω_i. If the mixer outputs desired angular speeds, then ROSflight converts these values to desired motor setpoints before they are sent to the motors and ESCs.

V. Integration Updates

A. Hardware Support

ROSflight supports FCU hardware configurations having at a minimum a six-axis inertial measurement unit (IMU with three-axis accelerometer + three-axis gyroscope) for local state estimation and control, and a radio control (RC) receiver for some control inputs. Here, we employ FCUs that additionally include three-axis magnetometer, barometer, differential pressure, and GPS sensors; and a telemetry data link. Measurement of this additional sensor data via the FCU ensures consistent and accurate relative timestamps for use by the companion computer. Figure 4 illustrates the interconnection of components, including the FCU, companion computer, telemetry, and ground control elements.

Two hardware configurations are currently supported and demonstrated (Table I). Configuration 1 is comprised of commercial off the shelf (COTS) components from 3DR and Nvidia, and is therefore more accessible. Configuration 2, provided by AeroVironment, Inc. (AV) benefits from a more tightly integrated hardware package, as well as higher performance IMU and differential pressure sensors. Both configurations share compatible STM32H7 microcontrollers allowing significant software commonality. RC and telemetry data links used in hardware experiments are listed in Table II. Note that other RC and telemetry data links are also supported.

To support different hardware (e.g., GPS, magnetometer, other sensor boards, or a different FCU) and to improve adaptability to different hardware configurations, ROSflight is designed so that the core firmware functionality is abstracted from the hardware implementation, called the hardware abstraction layer (HAL). The HAL for a given hardware configuration is created by inheriting from a base class that defines all the functions specifically related to hardware required by the ROSflight firmware. For example, to support a different GPS receiver, a user would create a driver to interface with the new receiver and then interface with ROSflight using the associated class member functions defined by the HAL. More generally, the entire board can be replaced as long as it employs the board HAL.

B. Software-in-the-Loop Simulation

A simulator is an essential component in robotics research that allows for rapid development before deploying on hardware. In many cases, the transition from simulation to

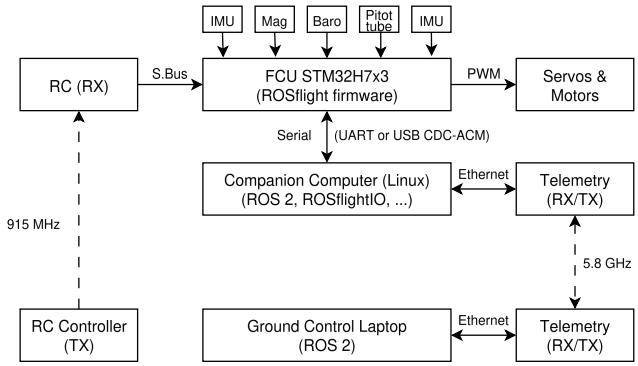

Fig. 4: Hardware elements

(a) FCU configuration 1

Component	Function
3DR Pixracer Pro	FCU with IMU and barometer
3DR M10034C	GPS and magnetometer
3DR M10121A	Power board
3DR MS4525DO	Differential pressure
Nvidia Jetson Orin Nano 8GB developer kit OR Raspberry Pi 5 8GB	Companion computer

(b) FCU configuration 2

Component	Function
AV Varmint	FCU with integrated IMU, barometer, magnetometer, differential pressure, GPS, power, and Nvidia Jetson Orin NX 16GB companion computer
1575A-L	GPS L1 active antenna

TABLE I: Controller configurations

hardware experiments can require extensive modifications or adjustments to the software used for the experiments due to hardware or other real-world constraints. ROSflight seeks to minimize the effort to transition from simulation to hardware environments by mirroring the hardware environment as closely as possible in simulation. This enables the same software that runs in simulation (i.e., the ROSflight firmware and an associated autonomy stack) to also control the vehicle in hardware, with no changes.

Many simulation environments have been developed to aid UAV research, including general-purpose simulators and application-specific simulators [16]. Each simulator has different attributes that make it more or less suited to a given research problem. For example, research that uses computer vision as an essential component of the autonomy stack likely needs a photorealistic simulator, while research focusing on multi-agent coordination and localization may only need a headless environment to do Monte-Carlo experiments.

The ROSflight simulation package is flexible and modular so that users can customize it to their needs by supporting their own simulator or by including custom functionality. ROSflight natively supports three simulators out of the box: HoloOcean [17], a photorealistic visualizer built on Unreal

Component	Function
TBS Crossfire 8Ch Diversity Rx	RC receiver
TBS Crossfire TX	RC transmitter
Doodle Labs 5GHz Embedded Mesh Rider, RM-5800-2J-XM	Air telemetry
Doodle Labs 5GHz Smart Radio, RM-5800-2J-XE	Ground telemetry

TABLE II: Remote Control and Telemetry

Engine 5, Gazebo [18], a popular open-source simulator for robotics, and a custom lightweight simulator that uses ROS 2 RViz for visualization.

The ROSflight simulation is split into a series of modules which are implemented as separate ROS 2 nodes and are described in Table III. This modularity enables users to define which modules are loaded at runtime. Since different visualizers come with different capabilities, this allows visualizers to be exchanged with minimal effort. For example, Gazebo performs dynamic propagation as part of the visualization engine, while the lightweight RViz visualizer only displays the 3D motion of the aircraft. Thus, when using Gazebo with ROSflight, the dynamics module is not launched while the rest of the ROSflight simulation modules remain unchanged.

Module Name	**Description**
Time Manager	Manages simulation time
SIL Board	Instantiation of firmware and simulated FCU board
RC	Simulates RC safety pilot connection
Sensors	Creates simulated sensor measurements
Forces and Moments	Computes aerodynamic forces and moments
Dynamics	Handles dynamic integration, and manages truth and environment state
Visualizer	Plots the vehicle in the simulated environment

TABLE III: Simulation Module Descriptions

The separation of responsibilities into modules also enables easy customization of each module. For example, eVTOL AAM aircraft are often quad-plane or tilt-rotor type vehicles and, as such, have nonstandard aerodynamic and propulsion models. The responsibility of the forces and moments module in ROSflight is to compute the aerodynamic and propulsive forces and moments from the input actuator commands. Users wishing to use a different aerodynamic model need only to replace the ROSflight forces and moments module, while the rest of the simulation environment remains unchanged. Sensors can also be added and customized with minimal effort. The sensors node is responsible for creating simulated sensor measurements based off of the true state of the vehicle, and includes an IMU, GNSS, barometer, magnetometer, sonar, and differential pressure sensor by default. To add a new sensor, one simply needs to create a ROS 2 publisher that generates the desired sensor data at a given rate.

Each module in the ROSflight simulation package is defined by an interface class. This interface class defines

Config.	Ave (ms)	Max (ms)	Min (ms)
1	1.712	84.103	0.146
2	0.416	2.334	0.174

TABLE IV: Bench test of RTT and publishing rate for both hardware configurations. The companion computer published ROS 2 offboard commands at 400 Hz.

Config.	Ave (ms)	Max (ms)	Min (ms)	Average received command rate (Hz)
1	1.024	56.921	0.154	1140
2	0.385	3.095	0.166	1360

TABLE V: Bench test of RTT and publishing rate for both hardware configurations. The companion computer published commands over the ROS 2 network as fast as possible.

the ROS 2 interfaces (e.g., publishers, subscribers, etc.) that a module has, and defines virtual functions that must be implemented by derived classes. As long as a node inherits from the interface class and implements the required functionality, that node will interact with the rest of the ROSflight simulation environment. Thus, other simulators can easily be supported by writing a ROS 2 wrapper around the simulator's API—as long as the new wrapper has the same interfaces as the interface class, the new simulator will interact properly with the rest of the ROSflight simulation.

VI. Hardware Demonstrations

Hardware experiments were performed to validate the improvements made to ROSflight 2.0. In the first, we measure the approximate serial delays from sending offboard commands to ROSflight from the companion computer. In the second, we demonstrate ROSflight in pass-through mode controlling a multirotor at 400 Hz.

A. Serial Delay

To measure the approximate serial delay, we modified ROSflight firmware to send all received offboard commands back to the companion computer. The companion computer first packed an offboard command message, marked the timestamp, and sent it to ROSflight firmware. The firmware parsed the message as part of its normal routine and immediately sent it back. Thus, each offboard command was sent twice across the serial connection. The companion computer received the message and computed the round-trip time (RTT). Note that the ROSflight firmware continued to run all of its normal operations during this experiment, including collecting, packing, and sending sensor data, responding to heartbeat requests, filling parameter requests, etc. This means that some variability and delay is included in the RTT measurement, as well as the time required for the normal operation of ROSflight. Data was collected for 45 seconds, and the average, maximum, and minimum RTT values were recorded. Table IV shows RTT statistics for when the companion computer sends offboard commands at 400 Hz. A Rasberry Pi 5 was used as the companion computer for all configuration 1 tests.

Table V shows RTT statistics for when the companion computer sent offboard commands as fast as possible. To do this, ROS 2 commands were published at a configurable rate, which was increased until the companion computer was unable publish ROS 2 messages fast enough. The maximum rate for both configurations was above 1100 Hz. The size of the offboard command message is 24 bytes, meaning that the serial connection was able to send > 52800 bytes per second of offboard command information under normal operation of ROSflight.

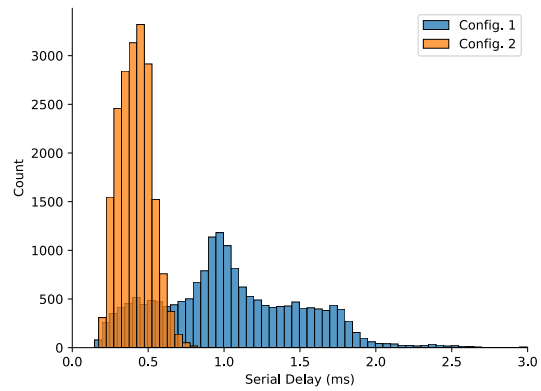

Fig. 5: Recorded RTT for offboard commands published at 400 Hz for both hardware configurations.

Figure 5 shows a histogram of the recorded RTT values for the data presented in Table IV, with commands being sent at 400 Hz. This shows that the maximum value of 56.921 ms reported in Table IV for configuration 1 is an outlier. Configuration 2 demonstrated significantly superior RTT values over configuration 1. Note that configuration 2 has an integrated design, but configuration 1 relies on a physical USB wire for the serial connection.

B. ROSflight in Pass-through

A flight test of ROSflight flying a multirotor in pass-through mode was conducted. For this experiment, configuration 2 was used on a Holybro x650 quadrotor frame. A custom mixer was computed using the proposed methods. A ROS 2 node containing a PID-based angle controller was implemented on the companion computer, which output forces and torques. These force and torque offboard commands were published over the ROS 2 network to the ROSflightIO node at 400 Hz, which forwarded them to ROSflight over the serial connection. Since ROSflight was operating in pass-through mode, these offboard commands did not pass through either of ROSflight's controllers, but instead progressed directly to the mixer. This operation is equivalent to sending direct motor commands from the companion computer to ROSflight over the serial connection.

The system's response to a triangle-wave roll command is shown in Figure 6, demonstrating that the companion computer was able to control the multirotor over a serial connection at 400 Hz with all control loops on the companion

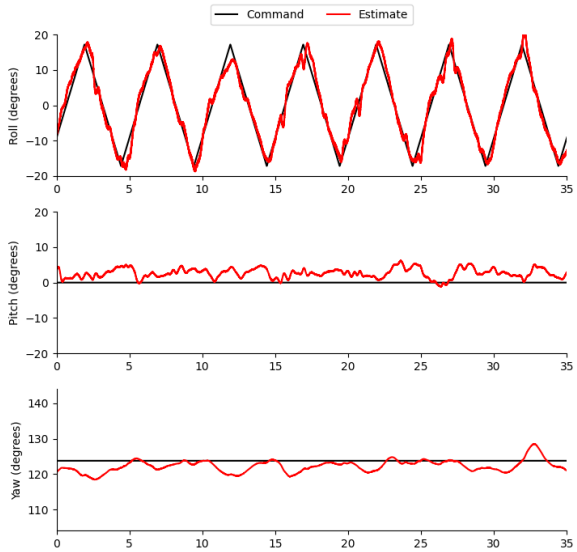

Fig. 6: Multirotor response in hardware to roll triangle-wave setpoints with the onboard computer sending forces and torques to the ROSflight mixer in pass-through mode, under mild wind conditions. Configuration 2 was used for this experiment.

computer. We emphasize that the offboard commands were sent directly to the ROSflight mixer, bypassing all control loops on the FCU.

VII. Summary

In this work we present significant advancements to ROSflight that improve the usability and modularity of the system. These advancements include the transition from ROS 1 to ROS 2, customizable actuator mixing, new supported hardware, and a more modular simulation environment. These improvements enable ROSflight to accelerate research in areas like advanced air mobility and UAVs. We demonstrated these new improvements in hardware, using ROSflight to control a multirotor at 400 Hz in pass-through mode, meaning all control loops were closed on the Linux-based companion computer. Approximate round-trip time delays in the serial connection were presented for both of the supported hardware configurations. Because ROSflight moves most of the higher-level autonomy functions to the companion computer, the ability to operate ROSflight in pass-through mode enables faster development and access to greater compute resources. ROSflight is documented in greater detail on the project website, rosflight.org, where users can contribute or ask questions.

Appendix

This appendix contains the derivation of the general-form and predefined mixer equations for standard multirotor vehicles used in ROSflight. Also note that this derivation assumes that all outputs are motor outputs, not servo outputs. The fixed-wing mixer derivation is not included here, but the mixer equations for a general aircraft can be derived by following a similar procedure.

A. General Form

As defined in [11], the thrust and torque generated by a motor and propeller in vector form are

$$\vec{F}_i = C_T \frac{\rho D^4}{4\pi^2} \Omega_i^2 \hat{e}_i \tag{2}$$

$$\begin{aligned} \vec{Q}_i &= \vec{r} \times \vec{F}_i + C_Q \frac{\rho D^5}{4\pi^2} \Omega_i^2 d_i \hat{e}_i \\ &= C_T \frac{\rho D^4}{4\pi^2} \Omega_i^2 (\vec{r} \times \hat{e}_i) + C_Q \frac{\rho D^5}{4\pi^2} \Omega_i^2 d_i \hat{e}_i, \end{aligned} \tag{3}$$

where $\hat{e}_i$ is the unit vector pointed in the direction of the axis of rotation of the motor, C_T and C_Q are the thrust and torque coefficients associated with the propeller (assumed to be constant), ρ is the air density, D is the propeller diameter, $\vec{r}$ is the vector pointing from the aircraft center of mass to the motor's axis of rotation, $d_i \in \{-1, 1\}$ encodes the direction of rotation of a propeller, and Ω_i is the angular velocity of the propeller and motor.

We wish to rewrite Equations (2) and (3) using Equation (1) to derive the mixing matrix M. Then, the desired forces and torques relate to the desired angular velocities squared by the mixing matrix M according to

$$\begin{bmatrix} \vec{F}_d \\ \vec{Q}_d \end{bmatrix} = M \begin{bmatrix} \Omega_1^2 \\ \Omega_2^2 \\ \vdots \\ \Omega_n^2 \end{bmatrix} = \sum_{i=1}^{n} M_i \Omega_i^2 = \sum_{i=1}^{n} \begin{bmatrix} \vec{F}_i \\ \vec{Q}_i \end{bmatrix}$$

where $\vec{F}_d$ and $\vec{Q}_d$ are the desired thrust and torque, M_i is the ith column of M, and $\vec{F}_i$ and $\vec{Q}_i$ are the thrust and torque produced by the ith motor.

The ith column of M is then given by

$$\begin{bmatrix} \vec{F}_i \\ \vec{Q}_i \end{bmatrix} = M_i \Omega_i^2 = \underbrace{C_T \frac{\rho D^4}{4\pi^2} \begin{bmatrix} \hat{e}_i \\ \vec{r} \times \hat{e}_i + \frac{C_Q D d_i}{C_T} \hat{e}_i \end{bmatrix}}_{M_i} \Omega_i^2.$$

This is a general form, so it is valid for any motor configuration, and the desired angular speeds can be computed by left-multiplying both sides by $M^\dagger$, where $(\cdot)^\dagger$ is the Moore-Penrose pseudoinverse.

If we know the motor parameters, we can compute the desired input voltage based on the desired angular velocities. From [11], we have

$$Q_p = \frac{\rho D^5}{4\pi^2} \Omega_i^2 C_Q \tag{4}$$

$$Q_m = K_Q \left[\frac{1}{R} (V_{\text{in}} - K_V \Omega_i - i_0 \right], \tag{5}$$

where Q_p is the torque generated by the propeller due to air resistance, Q_m is the torque generated by the motor, K_Q is the motor torque constant, R is the resistance of the motor, K_V is the back-EMF voltage constant, and i_0 is the no-load current.

Setting $Q_m = Q_p$ (as is true under steady-state conditions) yields

$$V_{\text{in}} = \frac{R C_Q}{K_Q} \frac{\rho D^5}{4\pi^2} \Omega_i^2 + i_0 R + K_V \Omega_i. \tag{6}$$

The throttle setting, $\delta_{t,i} \in [0, 1]$ can then be calculated using $V_{\text{in}} = \delta_{t,i} V_{\text{max}}$. This throttle setting is the PWM setting that the mixer writes to the motors, assuming that output voltage is scaled linearly to the PWM duty-cycle setting.

B. Simplifications

The general form of the mixing matrix described above requires knowledge of the motor parameters of the system. Some simplifications can be made to generate mixing matrices that do not depend on the motor and propeller parameters. This results in a more intuitive and simple mixing matrix, but requires different controller gains on the firmware controller to achieve the same performance.

Assumption 1: All motors and propellers have the same thrust and torque coefficients.

Assumption 2: $\hat{e}_i = -\hat{k} = \begin{bmatrix} 0 & 0 & -1 \end{bmatrix}^T$ in body-fixed north-east-down (NED) frame, meaning that the motors are all oriented in the $-z$ direction (i.e., thrusting straight up).

Then,

$$\vec{r} \times \hat{e}_i = \begin{bmatrix} r_y \hat{e}_{i,z} - r_z \hat{e}_{i,y} \\ r_z \hat{e}_{i,x} - r_x \hat{e}_{i,z} \\ r_x \hat{e}_{i,y} - r_y \hat{e}_{i,x} \end{bmatrix} = \begin{bmatrix} -r_y \\ r_x \\ 0 \end{bmatrix} = \begin{bmatrix} -||\vec{r}|| \sin\theta \\ ||\vec{r}|| \cos\theta \\ 0 \end{bmatrix}$$

and

$$\begin{bmatrix} \vec{T}_i \\ \vec{Q}_i \end{bmatrix} = C_T \frac{\rho D^4}{4\pi^2} \begin{bmatrix} 0 \\ 0 \\ -1 \\ -||\vec{r}|| \sin\theta \\ ||\vec{r}|| \cos\theta \\ \frac{C_Q D}{C_T} d_i \end{bmatrix} \Omega_i^2, \tag{7}$$

where θ is the angle from the positive body-fixed x-axis, as in [11].

Assumption 3: Each desired input value is computed independently (i.e., separate gains in a controller) than the other input values, and $||\vec{r}||$ is constant for all motors.

Then, the constant terms in (7) can be factored out and subsumed into the controller gains (and thus the desired inputs), leading to

$$\begin{bmatrix} \vec{T}_i \\ \vec{Q}_i \end{bmatrix} = \begin{bmatrix} 0 & 0 & -1 & -\sin\theta & \cos\theta & d_i \end{bmatrix}^T \Omega_i^2.$$

Assumption 4: $V_{\text{in}} \approx \frac{R\rho D^5 C_Q}{4\pi^2 K_Q} \Omega_i^2$, meaning that the squared term in (6) dominates.

The constants in 6 can be subsumed as above. Furthermore, since $V_{\text{in}} = \delta_{t,i} V_{\text{max}}$ we have

$$\begin{bmatrix} \vec{T}_i \\ \vec{Q}_i \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & -\sin\theta & \cos\theta & d_i \end{bmatrix}^T \delta_{t,i}. \tag{8}$$

Equation (8) shows the ith column of a simplified M, where the rows correspond to geometric properties associated with the given airframe. The assumptions made in the derivation of this equation limit the accuracy of this mixer model, but allow for an easy and intuitive understanding of the mixing matrix. The hardcoded mixing matrices (for multirotors) in ROSflight follow this form, allowing users to easily determine the mixing matrix and its limitations without knowledge of the motor parameters of the system.

References

[1] 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.

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

[3] 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

[4] J. Jackson, G. Ellingson, and T. McLain, "ROSflight: A lightweight, inexpensive mav research and development tool," in *2016 International Conference on Unmanned Aircraft Systems (ICUAS)*, 2016, pp. 758–762.

[5] 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.

[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] Paparazzi. (2025) Paparazzi UAV. [Online]. Available: https://wiki.paparazziuav.org/wiki/Main_Page

[8] iNav. (2025) iNav. [Online]. Available: https://github.com/inavFlight/inav/wiki

[9] 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.

[10] K. Belsare, A. C. Rodriguez, P. G. Sánchez, J. Hierro, T. Kołcon, R. Lange, I. Lütkebohle, A. Malki, J. M. Losa, F. Melendez, M. M. Rodriguez, A. Nordmann, J. Staschulat, and J. von Mendel, *Micro-ROS*. Cham: Springer International Publishing, 2023, pp. 3–55. [Online]. Available: https://doi.org/10.1007/978-3-031-09062-2_2

[11] R. W. Beard and T. W. McLain, *Small Unmanned Aircraft: Theory and Practice*, 2012, see also https://github.com/byu-magicc/mavsim_public.

[12] W. Koch, R. Mancuso, R. West, and A. Bestavros, "Reinforcement learning for UAV attitude control," *ACM Trans. Cyber-Phys. Syst.*, vol. 3, no. 2, Feb. 2019. [Online]. Available: https://doi.org/10.1145/3301273

[13] N. Lambert, D. Drew, J. Yaconelli, S. Levine, R. Calandra, and K. Pister, "Low-level control of a quadrotor with deep model-based reinforcement learning," *IEEE Robotics and Automation Letters*, vol. PP, pp. 1–1, 07 2019.

[14] F. Nan, S. Sun, P. Foehn, and D. Scaramuzza, "Nonlinear mpc for quadrotor fault-tolerant control," *IEEE Robotics and Automation Letters*, vol. 7, no. 2, pp. 5047–5054, 2022.

[15] H. Mahmood, U. Ali, and A. Akhtar, "Path invariance of a quadrotor system under cyber attacks with theoretical guarantees," in *2025 American Control Conference (ACC)*, 2025, pp. 972–977.

[16] C. A. Dimmig, G. Silano, K. McGuire, C. Gabellieri, W. Hönig, J. Moore, and M. Kobilarov, "Survey of simulators for aerial robots: An overview and in-depth systematic comparisons [survey]," *IEEE Robotics Automation Magazine*, vol. 32, no. 2, pp. 153–166, 2025.

[17] E. Potokar, K. Lay, K. Norman, D. Benham, S. Ashford, R. Peirce, T. B. Neilsen, M. Kaess, and J. G. Mangelson, "HoloOcean: A full-featured marine robotics simulator for perception and autonomy," *IEEE Journal of Oceanic Engineering*, vol. 49, no. 4, pp. 1322–1336, 2024.

[18] N. Koenig and A. Howard, "Design and use paradigms for Gazebo, an open-source multi-robot simulator," in *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (IEEE Cat. No.04CH37566)*, vol. 3, 2004, pp. 2149–2154 vol.3.