2017 International Conference on
Unmanned Aircraft Systems (ICUAS)
June 13-16, 2017, Miami, FL, USA

ROSplane:
Fixed-wing Autopilot for Education and Research

Gary Ellingson[1] and Tim McLain[2]

Abstract— **This paper presents a fixed-wing autopilot code base called ROSplane. ROSplane utilizes the ROSflight board, firmware, and driver, which was developed to make autopilot development faster, easier and cheaper. By leveraging a textbook and university course content, the autopilot facilitates education and accelerates research and development. The textbook provides high-level documentation for the code. The code is structured to facilitate learning by providing a framework for student assignments. The addition of ROSplane software and documentation make ROSflight closer to a plug-and-play solution while maintaining simplicity and usability for researchers and students. ROSplane has been used in a graduate level flight dynamics class, demonstrated through test flights, and modified for research purposes.**

I. INTRODUCTION

Small, unmanned, autonomous aircraft are increasingly being used in a variety of applications, including package delivery, surveillance, and remote sensing. There has also been an increase in research and development in making these aircraft more autonomous. To achieve high levels of autonomy the aircraft system require sensors, actuators, and computational device or autopilot. The autopilot must be programed with estimation and control algorithms for processing sensor inputs and providing actuator outputs.

Education about autopilots is necessary to facilitate research and development of autonomous aircraft. Students must learn the autopilot basics before attempting to modify or improve the existing autopilot technologies. Currently available autopilot hardware and education practices make it difficult to translate autopilot education into practice. This is largely because embedded autopilots are complex to modify and common simulation software does not seamlessly translate to autopilot code.

Most autopilots used in education and research are considered *plug-and-play*, which means they can be purchased off the shelf and are fully featured without the need for any development or modification. While research applications can benefit from having plug-and-play autopilot solutions, often it is more important that the autopilot inner workings be completely open to study and modification with no hidden or obfuscated sections. In other words, for education and research the autopilot should have no black boxes. In most instances, simple and well structured software is better than software that is feature-rich. Many autopilots often hide their inner workings by being closed source, overly complex, and/or under documented. Some research groups have developed their own autopilot hardware and code base to remove all black boxes (e.g. MIT ACL UberPilot [1]). Full development of an autopilot, however, can require a significant amount of development resources.

We have created our own fixed-wing autopilot code base called ROSplane to overcome these common challenges. ROSplane is based on the textbook *Small Unmanned Aircraft: Theory and Practice* [2], which is used in a graduate-level flight dynamics and control course at Brigham Young University (BYU). The autopilot structure directly matches the architecture presented in the book and class. The code base is also set up to allow students to easily modify autopilot code for assignments or projects. This means ROSplane users (researchers and students) can directly leverage documentation in the textbook and learning gained in the course. Thus ROSplane can both facilitate learning and rapidly advance research and development of autonomous fixed-wing aircraft.

II. RELATED WORK

Several fixed-wing autopilots already exist, including proprietary solutions such as those by MikroKopter [3], Ascending Technologies [4], Lockheed Martin Procerus [5], Cloud Cap Technologies [6] and others. There are also autopilots that are open source such as the Paparazzi [7], PX4 [8], and PixHawk [9]. Additionally, most of these autopilots are fully featured and are currently used in a large variety of research applications. Most of these autopilots originated from university-based projects.

Many of these autopilots offer full plug-and-play solutions for a wide variety of autopilot use cases, including high-level waypoint following and control. They often work with ground-control stations (QGroundControl, APM Planner, Virtual Cockpit, etc.) for a clean interface to autopilot functions. For research and/or educational purposes, however, that include low-level development of estimation or control, then they become hard to use because of the abstractions created by their plug-and-play characteristics.

The Robot Operating System (ROS) [10] is a software framework and middleware commonly used in many robotic applications and includes many standardized robotic development tools. Included are tools for processing a variety of sensor information, simulating robotic hardware, and developing robotic software. ROS utilizes a publisher/subscriber framework where every task is represented by and implemented as a node in a directed graph structure with inputs and outputs to other nodes. ROS has also been applied to research of autonomous aircraft [11]–[15]. Often ROS is used

[1]Gary Ellingson is a PhD candidate in the Department of Mechanical Engineering, Brigham Young University `gary.ellingson@byu.edu`
[2]Tim McLain is a professor in the Department of Mechanical Engineering, Brigham Young University `mclain@byu.edu`

to interface with the autopilot hardware through a telemetry link to extend the autopilot capabilities, while low-level estimation and control remain on the embedded autopilot.

ROSflight [16] is a input/output (I/O) board, firmware, and driver for ROS based autopilots. It was created to facilitate research-autopilot development. It utilizes inexpensive, open-source hardware and software to create a cheep, readily-available fight-control-unit I/O board. The I/O board abstracts the real-time critical processes such as sensor and receiver reading and actuator output. It allows autopilot estimation and control to happen directly in ROS, thus avoiding complex embedded development where possible. The ROSflight system also includes aircraft simulation tools that interfaces ROS with a Gazebo simulator.

The *Small Unmanned Aircraft: Theory and Practice* [2] textbook teaches all of the physical concepts and controls technologies necessary for autopilot development. For brevity, we will refer to the textbook as the UAV book. The course material includes MATLAB and Simulink templates that are used by students to create a full autopilot simulation. The files for each UAV book chapter are provided to the students with the main, flight-critical functions partially deleted. Throughout the course, students complete these functions and then test their autopilot simulation as part of their homework and lab assignments. These files allow the students to implement the inner workings of the autopilot without having to build from scratch all of the simulation structure and data flow.

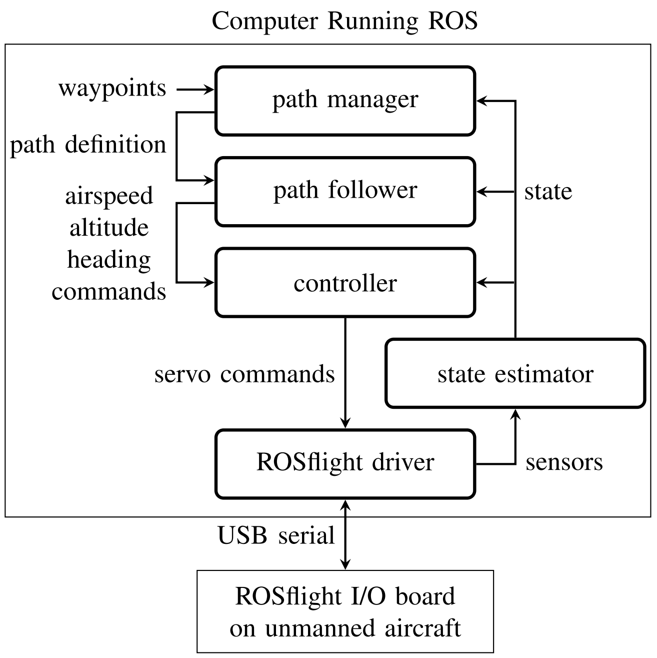

Fig. 1. *ROSplane system using the ROSflight I/O board [16] and implementing the UAV book architecture within ROS. Blocks that are bold with rounded corners are ROS nodes and blocks with square corners represent physical computing devices. Notice similarities to architecture found in the UAV book.*

III. System Architecture

The structure of the ROSplane implementation provides several usability benefits for education and research. First, the ROSplane system is implemented in ROS so the UAV book provides useful documentation. It matches the MATLAB simulation code in both in its form and educational function. The ROSplane code is also set up to be easily distributed for instruction purposes by removing some critical pieces while abstracting the complex structure and data flow, similar to the UAV book MATLAB and Simulink files.

A. Autopilot Structure

ROSplane implements the autopilot functions from the UAV book and maintains the same architecture. Fig. 1 shows the autopilot implemented in ROS, including the ROSflight driver and I/O board.

The autopilot computer receives sensor information from the ROSflight I/O board and provides the measurements to the state estimator ROS node. The sensors measurements incorporated are the three-axis accelerations and angular-rates from the inertial measurement unit (IMU), differential pressure from the airspeed sensor, barometric pressure altitude, and global positioning system (GPS) position and velocity. The sensor measurements are provided to the estimator, which propagates the IMU inertial state at 100 Hz and process GPS position updates at 5 Hz. An extended data rate and loop rate discussion for ROSflight is included in [16]. The estimator incorporates an extended Kalman filter to provide a state estimate to the rest of the autopilot nodes at 100 Hz and is described in [2].

The autopilot further consists of several ROS nodes that make up the rest of autopilot architecture including autopilot control loops, path follower, and path manager. Various chapters of the UAV book describe these pieces in detail. While the UAV book further develops autonomous path planner and vision based navigation sections, currently ROSplane does not implement these highest level functions. The current implementation receives preplanned waypoints from the operator and then execute the maneuvers to fly the path connecting the waypoints using orbital and straight-line segments. ROSplane default behavior is to update each control output at 100 Hz, and in step with the state estimation. All nodes can be throttled to operate at lower rates as processing limitations require.

Autopilot simulation can also be performed on a ROS computer by exchanging the ROSflight driver node with a Gazebo simulator (see Fig. 2). Gazebo simulates the dynamics of the aircraft and the aircraft sensors. Running the Gazebo simulation requires desktop-level computing resources. Since most of the student learning and coding happen at desktop workstations then this type of computing environment is both appropriate and advantageous. The Gazebo simulation provides simulated sensor measurements to the ROSplane estimator and receives actuator commands from the ROSplane controller. The simulation architecture is described in [16] and is similar to the simulator described

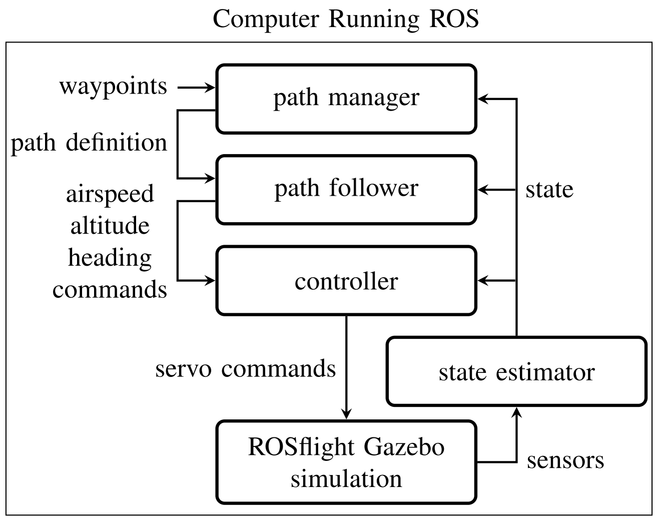

Fig. 2. *ROSplane system simulation using the ROSflight Gazebo simulation. Blocks that are bold with rounded corners are ROS nodes and the block with square corners represents a physical computing device. Notice the driver block is switched for simulation block.*

in [17]. Fixed-wing aircraft dynamics were added to feed the sensor models [2]. Different aircraft can be simulated by providing aerodynamic coefficients and control derivatives specific to that aircraft.

Using ROS for development of the autopilot code also has many advantages [16]. The main advantage is that it removes the necessity for embedded development and makes the desktop development environment the same as the operating and running environment. This removes the necessity for specialized software tool chains and hardware drivers. This facilitates debugging, simplifies simulation and testing, and improves autopilot accessibility for students. The use of ROS also standardizes the implementation, enabling it to work on multiple platforms and computer form factors.

B. Code Structure

The ROSplane code is set up so that it can be easily used by instructors and students in a teaching environment by abstracting the ROS information flow and carefully defining the structure of what the students do for their assignments. This limits the scope of the students assignments and allows students with limited ROS experience to develop autopilot code, within the ROS framework. ROSplane is implemented in C++ and uses class inheritance, polymorphism, and C++ data structures to abstract ROS communication.

Each ROS node has a key task to perform. The performance of the task is both what is studied and implemented by the students. We will refer to the performance of the task by each node as the *work* that the node performs. The rest of the code is to satisfy the requirements of the ROS middleware communication and to provide the structure for the student implemented *work* function. In general, each ROS node includes all of the following code pieces:

- main function
  - base class pointer to a child object
  - ROS spin function
- base class
  - protected input, output, and param structs
  - pure virtual *work* function declaration
    - ∗ receives static input and param structs
    - ∗ receives reference to non-static output struct
  - ROS subscriptions and publications
  - ROS timer that calls *work* at a specified rate
- child class (inherits from base class)
  - parent's *work* function implementation
    - ∗ gets information from input and params structs
    - ∗ performs the node's primary task, e.g. estimation
    - ∗ puts result in output struct
  - other helper functions as necessary

The above code structure allows the instructor to remove the child class and let the students reimplement the autopilot function without having to rewrite the ROS communication. In the above code, the *work* function is *pure virtual*, meaning it is defined by the base class but implemented only by the child class. Thus, the child class already has the correct information flowing into and out of the work function through the input, output, and param structs that are defined in the base class. The code structure also allows the autopilot to utilize different implementations of the same block by simply changing which child class is constructed (e.g. polymorphism).

IV. Results

To demonstrate the usefulness of ROSplane for education and research, this section provides a summary of the results of having used ROSplane in both the classroom and the laboratory. While our evaluation of usefulness is subjective, this paper offers a description of our efforts that may be helpful for other educators and researchers.

The ROSplane code and Gazebo simulation was first used in the flight dynamics class at BYU during winter semester 2016. Approximately 20 students were provided with the ROSplane source code and Gazebo simulator. The students had already gained experience with the MATLAB and Simulink implementations and were required to reimplement or extend several pieces of the autopilot in a variety of ways. They then ware able to test their work in the simulation environment. Valuable feedback was incorporated from these students to improve ROSplane. The students expressed that the code was fairly easy to use after being familiar with the MATLAB and Simulink files.

ROSplane was then demonstrated with flight tests on a HobbyKing Bixler 3 (see Fig. 3). The aircraft carried an onboard raspberry pi 2 computer running ROS and a Naze32 Rev5 I/O board running the ROSflight firmware.

Fig. 3. *Bixler 3 aircraft flown in ROSplane demonstration.*

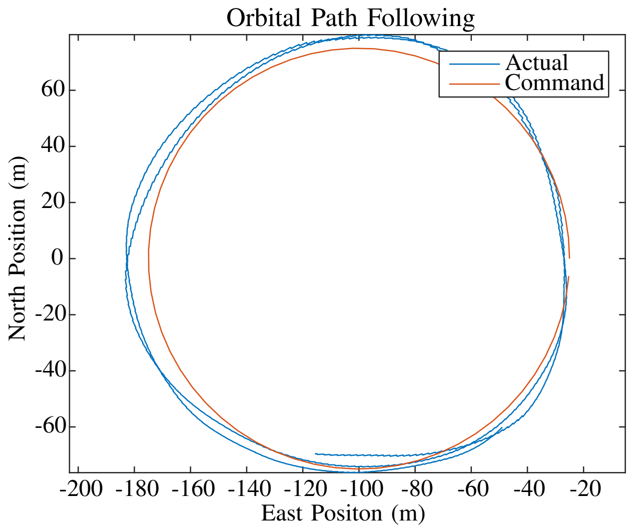

Fig. 4. *ROSplane demonstrating orbit following. Flown in approximately 4 m/s cross wind.*

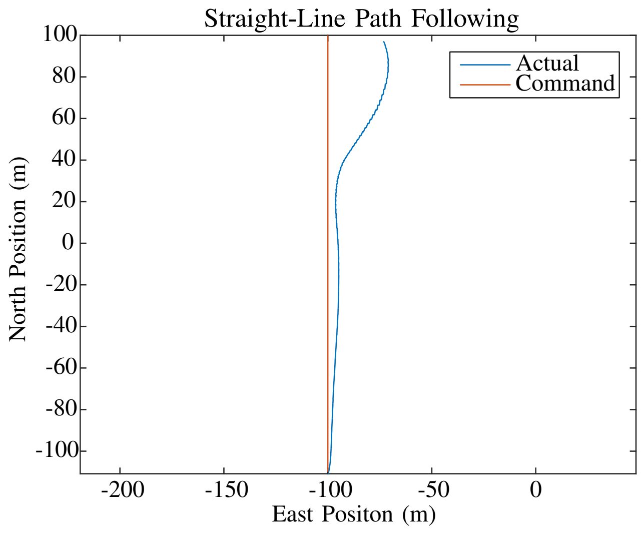

Fig. 5. *ROSplane demonstrating path following. Flying south after being disturbed from path.*

The flight took place using the same code that was provided to and reimplemented by the students. Fig. 4 and Fig. 5 show the aircraft telemetry while it followed orbital and straight-line paths respectively. The results shown represent neither the state-of-the-art in aircraft estimation and control, nor a complex use case. They do, however, demonstrate that students in a university course can produce the inner workings of a fixed-wing autopilot for a practical and useful flight using ROSflight and the documentation provided in the UAV book.

Several pieces of ROSplane were further modified for research purposes. Current and former students of the flight dynamics class were able to replace the longitudinal loops of the controller block with the total energy control scheme presented in [18]. Using any other autopilot, replacing the inner loops of a controller would likely have been difficult due to the complex nature of autopilot code and the difficultly in anticipating all of the implications of such changes. This was straightforward within ROSplane because the students were already familiar with the autopilot architecture from the UAV book. This demonstrates that ROSplane has the potential to be useful in research applications.

Because ROSplane is implemented in the ROS publisher/subscriber framework, each piece of the autopilot can be easily removed and replaced as research needs change. For example, if a new camera-based, estimation scheme is developed, then the estimator can be replaced. Further more, a large number of ROS packages for a variety of uses are freely available, including state-of-the-art image processing, estimation, and control, any of which could be adapted into the ROSflight architecture.

V. Current and Future Work

ROSplane is currently being applied to a number of other educational and research projects. These projects will help to further refine and improve the code base, which will, in turn, help to accelerate education and research.

Based on feedback from students ROSplane is being implemented in the python programing language. The python implementation will include the same code structure as described above and will maintain ROS compatibility. The goal is to make ROSplane even more accessible to students so that it can optionally replace the MATLAB and Simulink assignments in the flight dynamics course.

Several undergraduate students are also using ROSplane in a design-build-fly competition. The students will be adding to the code to enable specific functionality required for their competition, but that is not including in the UAV book. Added capabilities will include the ability to perform a landing flare, follow dynamic waypoints, and avoid collisions with other similar aircraft. The students are also created a ground station that will work with ROSplane for visualizing telemetry and sending high-level commands.

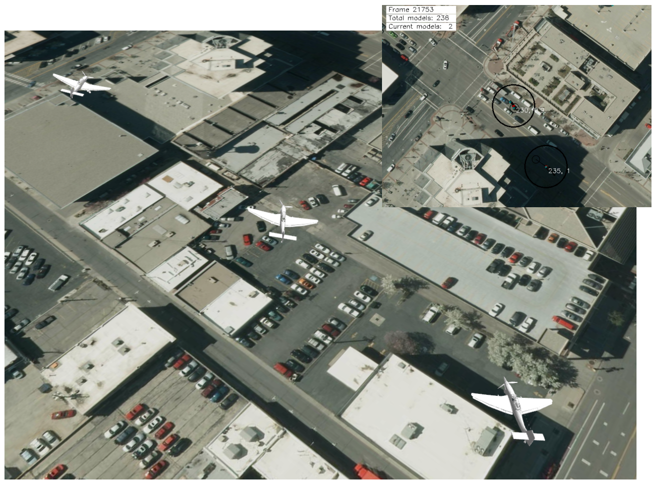

Fig. 6. A Gazebo screenshot showing multiple simulated aircraft. Simulated camera gimbal view is shown top right.

ROSplane is also being used for research in a multi-aircraft tracking project. Initial experiments have already begun using the Gazebo simulation environment. The simulation includes multiple aircraft and ground vehicles (see Fig.6). The simulation has been extended to include a simulated gimbal and camera on each aircraft, which allow Gazebo to simulate camera measurements. The project will be extended to hardware testing at a future date.

ROSplane is found in an online github repository [19]. Collaboration leading to improvements in robustness and usability is welcome. While the algorithms that are contained could be easily extended and improved, the repository will be limited to the documented algorithms in the UAV book. Our hope is that by limiting feature creep, the code base will maintain simplicity and usability necessary for students and researchers. Researchers are welcome to make their own changes as their needs require, but additional features will not be merged into ROSplane.

VI. Conclusion

ROSplane is a useful tool for teaching and learning as well as research. It is based on a textbook and university course that make it easily accessible to researchers and students. It uses the ROSflight I/O board and ROS middleware to make autopilot development easier by reducing the amount of embedded development required.

Use in a university course has shown how ROSplane can be used for educational purposes because the code is structured to abstract ROS communication through polymorphism. Flight demonstrations and code modifications also show that it can easily be used for autopilot research.

ROSplane is not as fully featured as some plug-and-play autopilots but it is useful because it is well-documented by the UAV book and has a well-structured code base. Every piece of the autopilot functionality is easily understood and therefore easily modified. Higher-level functionality can be added, while low-level components can be easily modified as research requires.

References

[1] M. J. Cutler, *Design and control of an autonomous variable-pitch quadrotor helicopter*. PhD thesis, MIT, 2012.

[2] R. W. Beard and T. W. McLain, *Small Unmanned Aircraft: Theory and Practice*. Princeton University Press, 2012.

[3] GmbH, HiSystems, "Mikrokopter," 2016.

[4] Ascending Technologies GmbH, "Ascending Technologies." `http://www.asctec.de/en/`, 2016.

[5] Lockheed Martin, "Kestrel Flight Systems & Autopilot." `http://www.lockheedmartin.com/us/products/procerus/kestrel-autopilot.html`, 2016.

[6] Cloud Cap Technology, "Piccolo autopilots." `http://www.cloudcaptech.com/products/auto-pilots`, 2016.

[7] B. Gati, "Open source autopilot for academic research-the Paparazzi system," *American Control Conference (ACC), 2013*, 2013.

[8] L. Meier, D. Honegger, and M. Pollefeys, "PX4 : A Node-Based Multithreaded Open Source Robotics Framework for Deeply Embedded Platforms," *International Conference on Robotics and Automation*, pp. 6235–6240, 2015.

[9] Ardupilot.com, "Ardupilot — open source autopilot." `http://ardupilot.com/`, 2016.

[10] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler, and A. Mg, "ROS: an open-source Robot Operating System," *ICRA*, vol. 3, no. Figure 1, p. 5, 2009.

[11] M. Achtelik, M. Achtelik, S. Weiss, and R. Siegwart, "Onboard IMU and monocular vision based control for MAVs in unknown in- and outdoor environments," *Proceedings - IEEE International Conference on Robotics and Automation*, pp. 3056–3063, 2011.

[12] M. Nieuwenhuisen, D. Droeschel, M. Beul, and S. Behnke, "Obstacle detection and navigation planning for autonomous micro aerial vehicles," *2014 Int. Conf. Unmanned Aircr. Syst.*, no. May, pp. 1040–1047, 2014.

[13] A. Harmat, M. Trentini, and I. Sharf, "Multi-camera tracking and mapping for unmanned aerial vehicles in unstructured environments," *Journal of Intelligent and Robotic Systems: Theory and Applications*, vol. 78, no. 2, pp. 291–317, 2015.

[14] N. Berezny, L. De Greef, B. Jensen, K. Sheely, M. Sok, D. Lingenbrink, and Z. Dodds, "Accessible aerial autonomy," *2012 IEEE Conference on Technologies for Practical Robot Applications, TePRA 2012*, pp. 53–58, 2012.

[15] M. Langerwisch, M. Ax, S. Thamke, T. Remmersmann, A. Tiderko, K. D. Kuhnert, and B. Wagner, "Realization of an autonomous team of unmanned ground and aerial vehicles," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 7506 LNAI, no. PART 1, pp. 302–312, 2012.

[16] 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)*, pp. 758–762, IEEE, 2016.

[17] F. Furrer, M. Burri, M. Achtelik, and R. Siegwart, *Robot Operating System (ROS): The Complete Reference (Volume 1)*, ch. RotorS—A Modular Gazebo MAV Simulator Framework, pp. 595–625. Springer International Publishing, 2016.

[18] M. E. Argyle and R. W. Beard, "Nonlinear total energy control for the longitudinal dynamics of an aircraft," in *2016 American Control Conference (ACC)*, pp. 6741–6746, IEEE, 2016.

[19] "ROSplane." `https://github.com/byu-magicc/ros_plane`, 2017.