uavAP: A Modular Autopilot Framework for UAVs

Mirco Theile[*], Or D. Dantsker[†], Richard Nai[‡], and Marco Caccamo [§]

Technical University of Munich, Garching, Germany

Simon Yu[¶]

University of Illinois at Urbana–Champaign, Urbana, IL, USA

Being applied to many fields of research and industry, UAVs require reliable but modular autopilot software. An autopilot task can range from simple waypoint following to complex maneuvering or adaptive mission tracking. The developed and presented autopilot, uavAP, aims to be fully modular in a decentralized manner, embracing an object-oriented design in C++. It implements a typical control stack comprising of a mission planner, global planner, local planner, and controller. To facilitate its modularity, uavAP makes use of its core, cpsCore, for module management as well as core utilities. cpsCore administers the configuration, aggregation, and synchronization of all the modules in uavAP. With the emulation environment uavEE, uavAP forms an ecosystem for rapid prototyping and testing of modules for various research directions, ranging from scheduling and memory management, through planning and control system design, to flight profile and configuration optimization. The uavAP-uavEE ecosystem has facilitated the design of an accurate UAV power model based on the aircraft's physical model, flight maneuver automation for aircraft system identification and dynamics parametrization, and an algorithm for geo-fencing of fixed-wing UAVs. This paper describes the control stack of uavAP, its core, cpsCore, as well as application examples highlighting the framework's modularity and flexibility.

I. Introduction

The popularity of UAVs in many fields of research and industry creates the need for reliable but modular autopilot software. An autopilot task can range from simple waypoint following to complex maneuvering or adaptive mission tracking. While there are existing autopilot systems with excellent community support, they are not sufficiently modular to enable rapid adaptability to varying research directions. The developed and presented autopilot, uavAP, aims to be fully modular in a decentralized manner, embracing object-oriented design in C++. Every functionality in uavAP is provided by a module, which can be adapted or replaced.

uavAP is an open-source autopilot framework.[1] The autopilot structure is designed for distributed functional executions that separate control, planning, and communication to increase safety and security at software level. It implements a typical control stack comprising of a mission planner, global planner, local planner, and controller. The high-level and abstract design of uavAP allows for seamless switching and transition between various planning and control algorithms. The customizability of the autopilot structure provides the flexibility that allows for rapid interfacing with various hardware systems such as the Al Volo FC+DAQ system,[2] which is used for high precision data collection necessary for applications such as power modeling.

To facilitate the modularity, uavAP makes use of cpsCore for module management as well as core utilities. cpsCore administers the configuration, aggregation, and synchronization of all the modules in uavAP. Through these steps,

[*]Ph.D. Student, Department of Mechanical Engineering. mirco.theile@tum.de
[†]Researcher, Department of Mechanical Engineering, or.dantsker@tum.de
[‡]M.S. Student, Department of Informatics, richard.nai@tum.de
[§]Professor, Department of Mechanical Engineering, mcaccamo@tum.de
[¶]Ph.D. Student, Department of Electrical and Computer Engineering. jundayu2@illinois.edu

each module can easily specify a set of parameters which are loaded and applied on program startup, communicate and interact with other modules, and start their task synchronously throughout threads and individual processes. Additionally, cpsCore offers core utilities for essential tasks such as scheduling, inter-process communication, and more. While initially only the core of uavAP, cpsCore has become an individual project because of its valuable support for modularization of any C++ software framework for cyber-physical systems (CPS).

The uavAP autopilot framework forms an ecosystem with uavEE, an open-source emulation environment for UAVs.[3,4] uavAP interfaces with uavEE for the communications with flight simulations while uavEE enables rapid testing and debugging of the uavAP autopilot framework and planning and control designs and implementations. More importantly, the combination of the uavAP and uavEE framework has enabled projects on variable applications in a wide range of areas. The uavAP-uavEE ecosystem has facilitated the design of an accurate UAV power model based on the physical model of the aircraft. Additionally, a flight maneuver automation framework[5] has been developed in uavAP and tested in uavEE. The framework automates flight testing maneuvers for aircraft system identification and dynamics parametrization,[6] yielding more consistent and repeatable results than human operators. Finally, an accurate kinematic model and algorithm for fixed-wing aircraft geo-fencing have been developed using uavAP and uavEE.[7]

The paper is structured as follows: In Section II, the autopilot framework uavAP is introduced with a summary of its planning and control stack and distributed architecture. This is followed by a description of its underlying core, cpsCore, which manages the modules and provides core utilities. In Section IV, a flight maneuver automation integration into uavAP is shown as an example of uavAP's modularity and flexibility. Some applications of uavAP are shown in Section V, followed by a comparison with other open-source autopilots in Section VI. Section VII concludes the paper and presents an outlook into future work.

II. Modular Autopilot Framework – uavAP

This section describes the autopilot framework by introducing the implemented control stack and its individual modules. Furthermore, its distributed architecture is depicted and described.

A. Plann

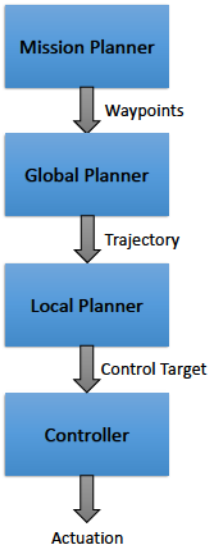


Figure 1. Planning and control stack implemented in uavAP.

The hierarchy of the control process can be represented by a stack, depicted in Figure 1. A mission planner generates waypoints according to the overall mission. The waypoints are passed to the global planner, which calculates the mission trajectory. Based on the trajectory and the current position of the aircraft, the local planner calculates the necessary angular rates and velocities to reach the trajectory, formulating a controller target. This controller target is passed to the controller, which calculates the actual actuator commands. The individual modules, as implemented in uavAP, are described in the following.

1. Mission Planner

The mission planner uses predefined missions, which can be selected by the user at run-time. The predefined mission consists of waypoints which should be passed in specified order with a specified velocity. Alternatively, the mission planner could generate waypoints to adapt its mission. However, this is out of scope for this work.

2. Global Planner

The global planner's task is to calculate a trajectory based on the set of waypoints received from the mission planner. The calculation of the trajectory can differ based on the overall goal of the mission. The simplest global planner is to connect the waypoints with lines, leading to a polygonal path, which is not the best solution since it leads to abrupt turns and consequently, high deviations from the planned trajectory. Alternatively, the waypoints can be connected with three-dimensional cubic splines, which are implemented in the *SplineGlobalPlanner* in uavAP.

A cubic spline is defined through a third degree polynomial

$$x(u) = f_x(u) = c_{0,x} + c_{1,x}u + c_{2,x}u^2 + c_{3,x}u^3. \tag{1}$$

The parametrization $u \in [0,1]$ is defined such that for $u = 0$ the spline is at the start point and at $u = 1$ at the end point. Extending this expression to three dimensions yields

$$\vec{\mathbf{x}}(u) = \mathbf{f}(u) = \mathbf{c}_0 + \mathbf{c}_1 u + \mathbf{c}_2 u^2 + \mathbf{c}_3 u^3, \quad \vec{\mathbf{x}} = \begin{pmatrix} x \\ y \\ z \end{pmatrix}, \quad \mathbf{c}_k = \begin{pmatrix} c_{k,x} \\ c_{k,y} \\ c_{k,z} \end{pmatrix} \tag{2}$$

for one specific spline. The spline between the waypoints $\mathbf{p}_i$ and $\mathbf{p}_{i+1}$ is defined by

$$\vec{\mathbf{x}}(i,u) = \mathbf{f}_i(u) = \mathbf{c}_{0_i} + \mathbf{c}_{1_i} u + \mathbf{c}_{2_i} u^2 + \mathbf{c}_{3_i} u^3. \tag{3}$$

The *SplineGlobalPlanner* uses the Catmull-Rom formulation to calculate the parameters of the splines. The complete mathematical derivation can be found in.[8] Catmull-Rom splines enforce a specified tangent at each waypoint. The tangent is based on the previous and next waypoint, making each spline dependent on only four waypoints. Defining $\mathbf{C}_i = [\mathbf{c}_{1_i}, \mathbf{c}_{2_i}, \mathbf{c}_{3_i}]^T$, the spline parameters can be calculated as follows:

$$\mathbf{C}_i = \begin{bmatrix} -\tau & 0 & \tau & 0 \\ 2\tau & \tau-3 & 3-2\tau & -\tau \\ -\tau & 2-\tau & \tau-2 & \tau \end{bmatrix} \begin{bmatrix} \mathbf{p}_{i-1} \\ \mathbf{p}_i \\ \mathbf{p}_{i+1} \\ \mathbf{p}_{i+2} \end{bmatrix} \tag{4}$$

and $\mathbf{c}_{0_i}$ is $\mathbf{p}_i$. The parameter τ indirectly defines how high the curvature is at the waypoints. A higher τ reduces the curvature at the waypoint but increases the curvature between the waypoints. Since each spline is dependent on a constant number of four waypoints the complexity of the Catmull-Rom spline generation is $\mathcal{O}(n)$, where n is the total number of waypoints in the mission. The *SplineGlobalPlanner* implements the calculation of Catmull-Rom splines

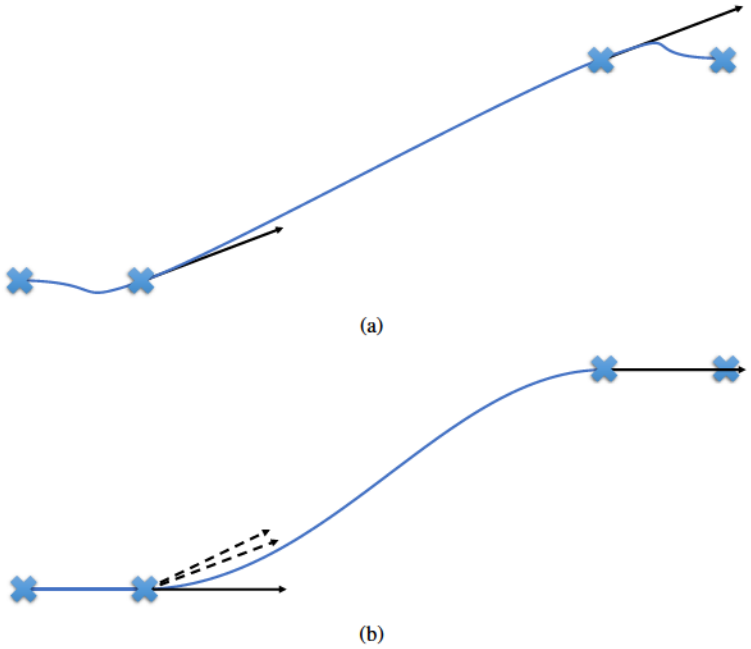


Figure 2. Catmull-Rom spline problem (a) and solution (b) for the z-coordinate spline.

because of their simplicity. The generated trajectory at its z-coordinate projection, however, shows problems when using the Catmull-Rom formulation. Figure 2(a) shows a side view of the trajectory. The black arrows show the tangents at the waypoints, which lead to unwanted altitude changes. The solution to this problem is to decouple the z-calculation from the calculation in (4). The tangent at the waypoints is set to the minimum absolute slope of three different slopes shown in Figure 2(b) on the left side. The three different slopes result from the vectors connecting the previous and the current waypoint, the current and the next waypoint, and the previous to the next waypoint. This modification is possible since the Catmull-Rom formulation allows for local control at each waypoint.

3. Local Planner

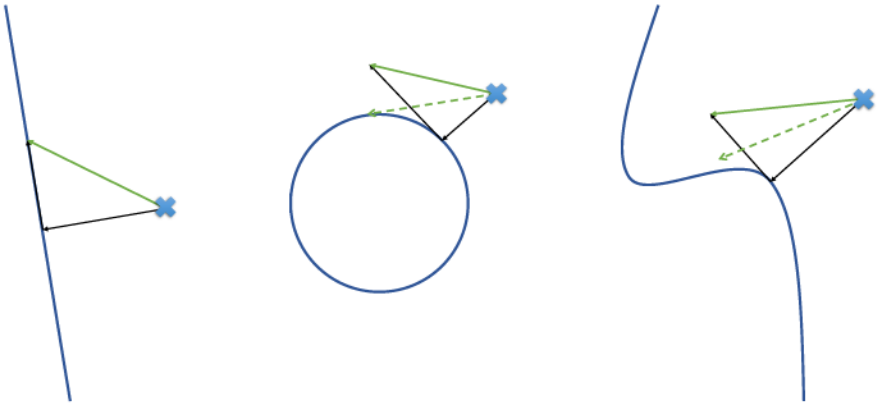


Figure 3. Three examples of the super-position in the Linear Local Planner; The X shows the position of the aircraft, and the green line is the super-position of the tangential and the orthogonal components in black. The dashed green arrow incorporates the curvature.

The trajectory that is calculated in the global planner is passed to a local planner. The local planner calculates the velocities and angular rates that are needed to converge to the trajectory. When the aircraft is moving on the trajectory,

the local planner calculates velocities and angular rates to stay on it. In the case of the *LinearLocalPlanner* in the uavAP framework, the functionality is a super-position of the movement on and towards the trajectory. A graphical representation of this super-position can be seen in Figure 3 represented by the solid green line.

Besides the super-position to calculate the target direction, the local planner can also incorporate the slope and curvature of the trajectory. For this, the planner first calculates the closest point on the trajectory to determine the local curvature and slope. For a line and orbit, the calculations of the closest point, the curvature, and the slope are straight-forward. The calculations for the *SplineGlobalPlanner* are based on the derivatives of (3). The results of adding the local curvature to the plan can be seen in Figure 3 represented by the dashed lines. A curvature target leads to an offset from the direction target of the super-position in the direction of the curvature, which allows the aircraft to converge faster.

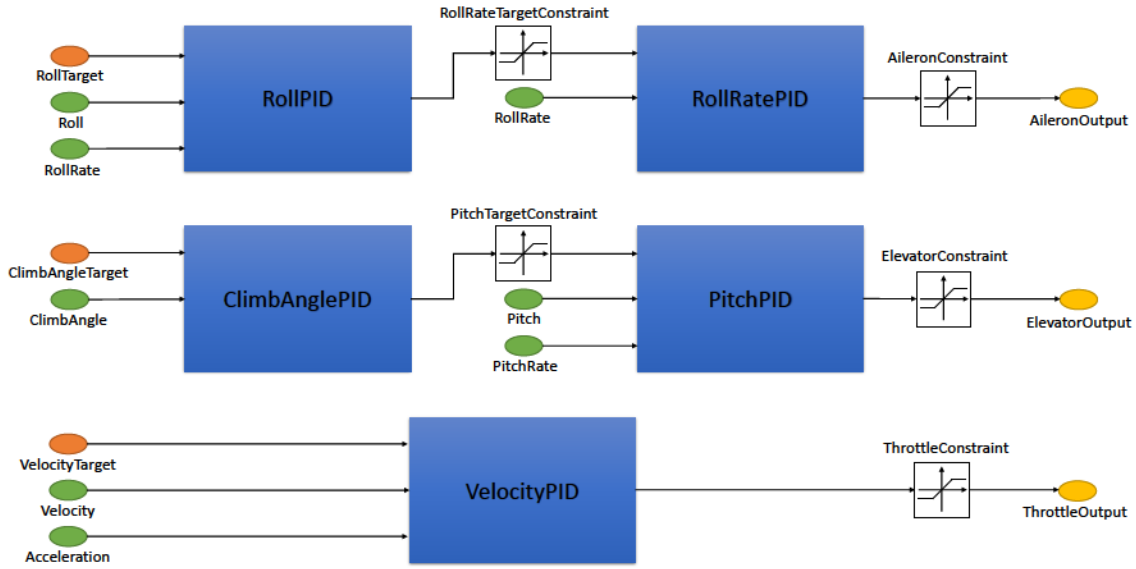


Figure 4. PID cascade: orange inputs represent the controller target from the local planner, green inputs are representing sensor data, and the yellow outputs are defining the actuation command.

Using the controller target, defining angular rate and velocity targets, the controller calculates the necessary actuation command. For its controller, uavAP uses cascaded PIDs. The schematics of the controller cascade is shown in Figure 4. The cascade consists of five PIDs that are connected in series or parallel to achieve the actuation calculation. The cascade can be separated into three different parts, yaw-rate control, climb-rate control, and velocity control. Additional PIDs can be used to actuate the rudder of the aircraft for β control.

The advantage of this PID cascade is that it is easy to set up and tune for different aircraft, using on-line tuning, as well as allowing intermediary constraints, such as the constraints on roll and pitch. Additional PIDs can be added if roll-rate or pitch-rate has to be constrained as well. On-line tuning is done using the ground station of uavAP, a part of uavEE.

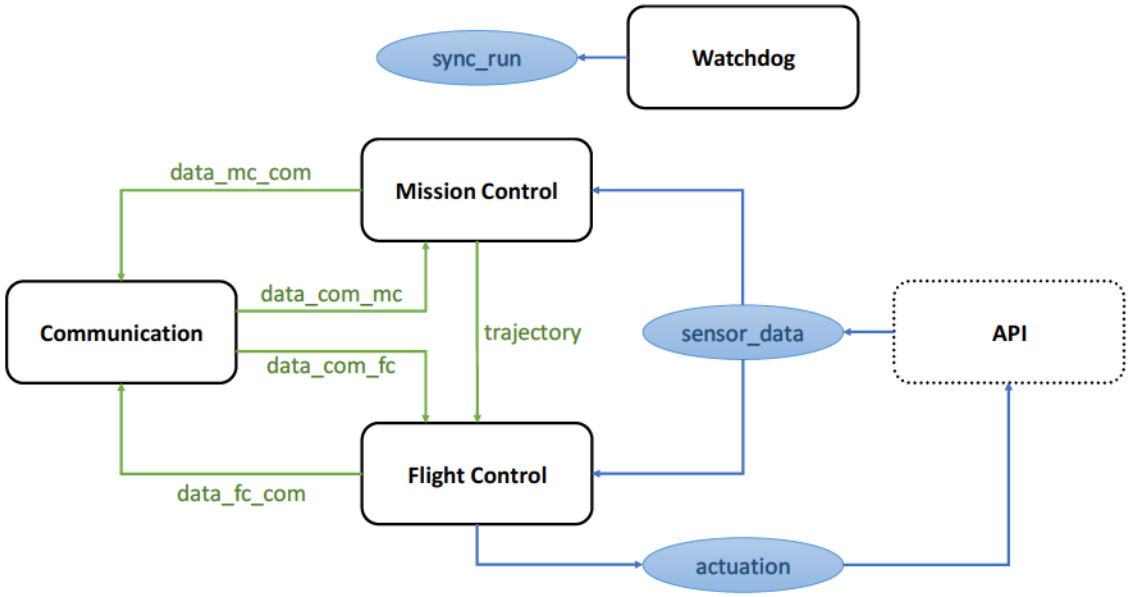


Figure 5. Control stack implementation showing processes and inter-process communication, green lines are message queues and blue lines and ellipses shared memory objects.

B. Distributed Architecture

The processes implementing the control stack and the periphery are visualized in Figure 5. Mission control is taking care of mission planning, and global planning and flight control is implementing the local planner and controller. The two are separate in order to protect the essential flight control from crashes or timing issues in mission control.

The watchdog process is the master of the synchronization in the start-up of the processes. Afterward, it monitors the processes, allowing them to restart them if they terminate or show a failure state. Additionally, the watchdog can perform strict scheduler monitoring conducting restarts of the processes if they are missing their deadlines.

The communication process is the interface of the autopilot to the outside, mainly the ground station. It is a separate process because it handles IO operations that can be slow or unpredictable. Additionally, since it is not mission-critical, if the communication process crashes, the other processes can continue unaffected. The main task of the communication process is to offer tuning, overriding, and selection interfaces to all the other processes. Additionally, it periodically sends the status information from flight control to the ground. If the user at the ground station requests the active mission and trajectory, the communication process forwards these requests to the appropriate process.

The API itself is not a process, but it is used by any background process that is responsible for the collection of sensor data and the actuation of the actuators using the actuation command. The implementation of the data collection can differ, but the API is defined in uavAP.

III. Autopilot Core – cpsCore

The core functionality of uavAP is grouped into cpsCore,[9] a C++ framework designed to simplify the design and implementation of cyber-physical systems. The cpsCore can be used as a baseline for modular object-oriented C++ frameworks. It allows for configuration, aggregation, and synchronization of individual modules and provides utility modules for a variety of standard tasks such as scheduling. This section describes cpsCore and its role within uavAP.

A. Module Management

To facilitate uavAP's high module configurability, cpsCore contains the functionality to parse configuration files for configurable classes and create and arrange them on process start-up in a modular manner. The process uses a Helper module, a module with knowledge of all possible modules that can be created to parse the configuration, which then generates and configures the specified modules. These modules are passed to an *Aggregator*, which aggregates the modules. This aggregation is then synchronized in stages and, if successfully passing all the stages, allowed to start its schedule of tasks. This process is depicted in Figure 9. An example of the *SplineGlobalPlanner*'s usage of cpsCore is provided in Figure 6. Specific details of configuration, aggregation, and synchronization are presented in the following.

```
class SplineGlobalPlanner : public IGlobalPlanner,
                            //uavAP: Its interface class
                            public AggregatableObject<ILocalPlanner, IPC, DataPresentation>,
                            //cpsCore: Dependencies to a local planner (uavAP), inter-process
                            //         communication (cpsCore), and data presentation (cpsCore).
                            //         Allows it to be aggregated.
                            public ConfigurableObject<SplineGlobalPlannerParams>,
                            //cpsCore: Struct with its parameters that should be configured
                            public IRunnableObject
                            //cpsCore: Indicating that it implements a run function for
                            //         synchronization
{
...
};
```

Figure 6. Inheritance of the *SplineGlobalPlanner* using cpsCore's functionality.

1. *Configuration*

Configuration is used to make the module assembly and the modules themselves configurable. Typically a JSON file is used to define the configuration. However, support for other file types could be added easily. The configuration file indicates which objects are to be created and how their parameters are to be set. A parameter struct is used to specify the parameters of each object. An example is that of the *SplineGlobalPlanner*, as shown in Figure 7. In this struct, the default value, the corresponding string in the configuration file, and the mandatoriness, indicating if the parameter has to be specified in the configuration file, are defined. The templated `configure(Config& c)` function provides the C++ struct with reflection, a concept that allows, among others, the struct to be iterated over. A corresponding JSON type configuration file is shown in 8.

Additionally, the parameter structure is used to generate configuration files, showing all the possible parameters that can be modified. This is particularly helpful while adding new modules to help maintain configuration files. The parameter structure could further be used through other means of configuration, such as a graphical user interface, as the basic structure of configuration is templated and allows for the necessary modifications.

2. *Aggregation*

The concept of *Aggregation* is a decentralized solution for setting pointers to dependencies within a process. Instead of having one entity knowing all the dependencies of each module and setting all of them, an *Aggregation* of the modules is formed. Each module that is an *AggregatableObject* can browse through the *Aggregation* for its dependencies. If found, a weak pointer to the dependency is created and stored, which can be retrieved and upgraded to a shared pointer

```
struct SplineGlobalPlannerParams
{
//  Parameter type      name        default        id       mandatory
    Parameter<float> orbitRadius = {50.0,  "orbit_radius", false};
    Parameter<float> tau         = {0.5,   "tau",          false};
    Parameter<bool>  smoothenZ   = {true,  "smoothen_z",   false};

    template <typename Config>
    inline void
    configure(Config& c)
    {
        c & orbitRadius;
        c & tau;
        c & smoothenZ;
    }
};
```

Figure 7. Parameter structure of the *SplineGlobalPlanner*.

```
{
    "orbit_radius": 50.0,
    "tau": 0.5,
    "smoothen_z": true
}
```

Figure 8. Generated .json configuration file of structure in Figure 7.

using a templated `get<Dependency>()` function. The weak pointer is used to avoid circular ownership, which can lead to complications during tear down.

The *Aggregator* is the owner of the objects in a process and is therefore responsible for their destruction when the process is terminated. To do so in a predictable manner, the *Aggregator* first stops active subscriptions, to avoid triggers from other processes. Second, the scheduler is stopped, descheduling all of the periodic events. Finally, the *Aggregation* container of the *Aggregator* can be emptied, destroying the aggregated objects sequentially.

3. Synchronization

Synchronization in a distributed system is a crucial factor for clean and predictable behavior. Especially if there are dependencies between processes that need to be established first, synchronizing the start-up phase is crucial. In uavAP synchronization is conducted among the modules inside one process, and among the processes in a distributed multi-process setup.

As described before, the schematic, shown in Process 2 of Figure 9, illustrates the start-up steps of one single process. For synchronization, a Runner utility sequentially triggers the current run stage for each module before moving on to the next stage. It first triggers the INIT run stage. In this stage, each module should check if all its dependencies are met. If not, the Runner aborts and prints corresponding error messages. In run stage NORMAL, the objects schedule their tasks or communicate with other objects to set up the process. Run stage FINAL is reserved for tasks that need three steps to set up. After run stage FINAL, the scheduler is triggered to start its schedule.

For multi-process synchronization, as necessary in Figure 5, the single process case is extended. An entity, such as a Watchdog, starts all the desired processes, waiting until they all reach the beginning of run stage SYNC, which is an idle run stage that is used to wait for the other processes. When all the processes reached that point, the Watchdog

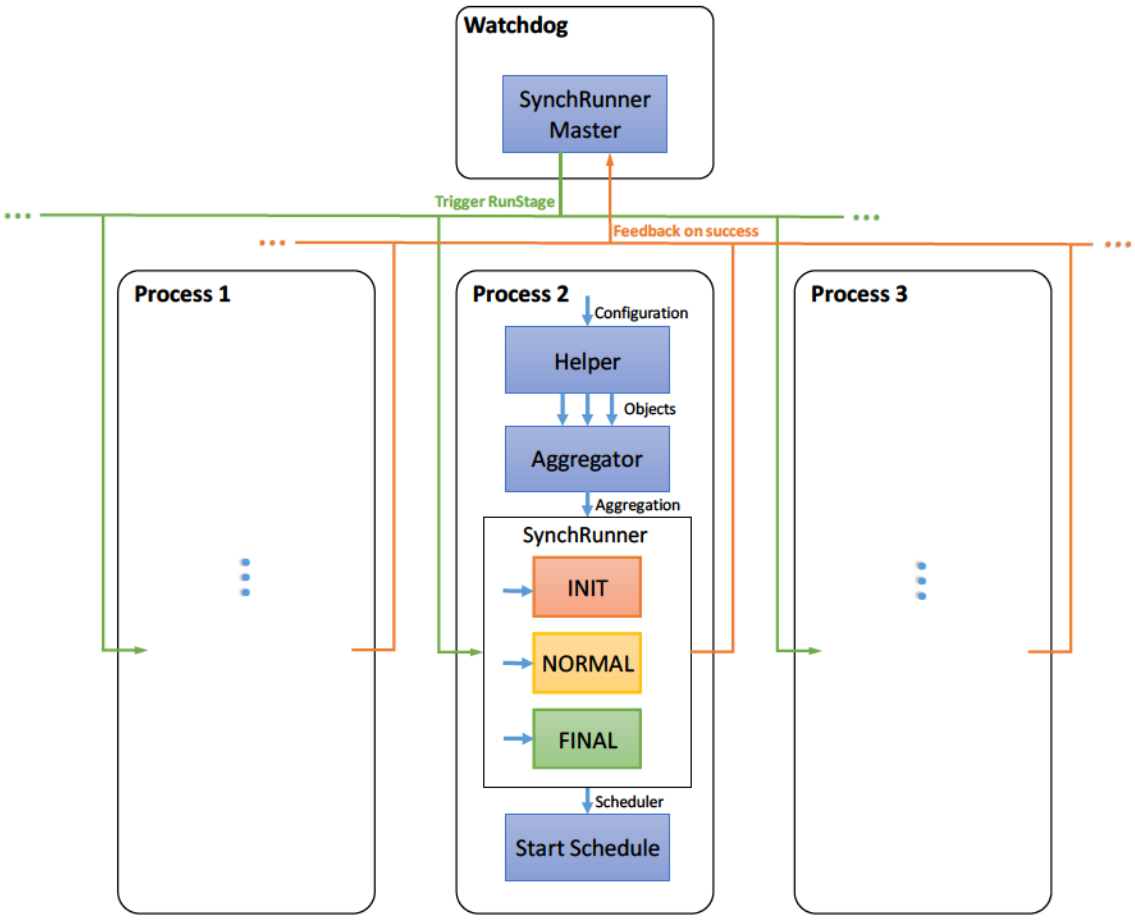


Figure 9. Multi process synchronization in uavAP

triggers the run stage INIT. All the processes now run their INIT run stage. If they succeed and do not discover a problem, they notify the Watchdog that they succeeded. This is handled by using thread barriers with a count of the number of processes. If they do not succeed, they do not notify the Watchdog leading to a time-out that lets the user know that one of the processes failed. After run stage INIT, the same synchronization procedure is executed for run stage NORMAL, followed by FINAL. After every process runs into the thread barrier of stage FINAL, they start their schedule simultaneously. In the multi-process case, the synchronization information is shared with a segment of shared memory, maintained by the synchronization master, e.g., the Watchdog.

B. Core Utilities

The core utilities of cpsCore comprise of functionality that is frequently used in CPS applications such as uavAP or also uavEE. They are implemented to be as generic as possible while allowing low-level optimization. These core utilities are used for scheduling, timing, inter-process communication, inter-device communication, and data presentation.

1. Scheduling and Timing

A scheduler handles every scheduling of periodic and non-periodic tasks in the autopilot. Any scheduler that is implemented should provide the ability to schedule periodic and non-periodic events as well as the possibility to start

and stop the schedule. The current main scheduler of uavAP is the *MultiThreadingScheduler*, which, as indicated by its name, uses multiple threads to execute tasks in parallel. When a task is scheduled, the scheduler creates an event object. Each event contains the function handle, scheduling information, a condition variable, and a thread for execution. The scheduler triggers the individual condition variables at the time of the task release. After completion of their tasks, the threads either end execution, if they were non-periodic or canceled, or wait until called again. The time provider handles the timing of the scheduler.

Any time provider in cpsCore has to provide the current time and the ability to sleep for a set amount of time or until a specific time point. The time provider used in uavAP is the *SystemTimeProvider*, which uses the standard chrono time library to provide time information as well as timing functionality, such as *wait for* or *wait until*. For manual time and scheduling control, essential for unit testing, the *MicroSimulator* can be used. It provides objects with manual time and scheduling information allowing full control during testing.

2. Inter-Process Communication

For the communication among processes, such as *Mission Control* and *Flight Control*, cpsCore offers inter-process communication (IPC) utilities. The IPC module allows message passing to one or multiple destinations. If communication to only one destination is requested, the IPC module creates a message queue to which the destination process can subscribe. For multiple destinations, the module creates a shared memory segment, allocating space for the data as well as synchronization fields. The destination processes can find the message queues or shared memory segments via string IDs. The general implementation is similar to message brokering, allowing publication and subscription.

3. Inter-Device Communication

To communicate with other devices, such as the ground station in uavAP, cpsCore provides utilities for inter-device communication (IDC). The IDC is split up into two layers, the transport layer, and the network layer. The transport layer is a current place holder if packet segmentation and acknowledgments are to be added. The network layer takes care of the dissemination of the packets to their destination. For that, it can use serial communication using the boost asio backbone, or ethernet communication using Redis,[10] specifically its message broker service. For packet verification, crucial for radio communication, a cyclic redundancy checksum (CRC) is implemented. The checksum is appended to each packet and can be verified on the receiving end.

The network and transport layer functionality are hidden from the other modules by providing an IDC module, which routes the packets according to configured string IDs. This way, the communication method can be changed through configuration and does not require recompilation or rewriting of code.

4. Data Presentation

For IDC and IPC, the data structures in uavAP have to be represented as binary. For passive data structures or plain old data (POD), this binary representation is as simple as a memory copy, provided the sender and receiver device use the same endianness. For more complicated structures with optional fields or nested members, cpsCore provides a data presentation utility. Similar to the configuration, the data presentation adds functions for code reflection, which specify how to serialize and deserialize structures. Using these functions, data presentation creates string-based packets from the complex structures which can be sent via IPC or appended with headers and send via IDC.

IV. Flight Maneuver Automation Framework – uavAP Extension

The introduced uavAP software framework allows for a simple and configurable extension of a flight maneuver automation framework on top of the existing software stack. The flight maneuver automation framework utilizes the introduced cpsCore framework to extend the current mission planner module, enabling automatic maneuver executions and transitions. A new flight analysis process is also added for providing the extended mission planner module with various aircraft states analysis needed for automating the aircraft flight maneuvers.

The current mainstream approach for collecting aircraft aerodynamic parameters is to manually pilot UAVs through a series of flight testing maneuvers.[11–15] Automating such flight testing maneuvers,[5,6] on the other hand, allows for the process of aircraft parametrization and modeling to be performed systematically and repeatably with minimal trial-and-error, and, more importantly, reduces the required amount of flight time and power consumption. For instance, by automating maneuvers during the flight, aircraft states such as attitude angles and velocity vectors can be set and maintained by controllers with better accuracy, consistency, and repeatability than manual piloting.

As the industry of small UAVs becomes increasingly popular, the safety and regulation for these small aircraft are also becoming essential for their applications and deployment. One of the useful and practical methods of executing the safety regulations on those small aircraft is to require them to have mandatory and built-in geo-fencing systems that provide constraints to their behaviors and missions. The flight maneuver automation framework, together with a robust and precise kinematic model detailed in,[7,16] forms an advanced geo-fencing system for UAVs to perform trajectory modeling, boundary checking, and automated evasive maneuvers.

A. Maneuver Planner

As discussed in Section II, the mission planner module in the uavAP software framework provides high-level mission planning and global plan generation. The existing global planner in the mission planner module takes mission waypoints as input parameters and generates a position-based trajectory as its output. The generated trajectory is then passed to the flight control module for local planning and controller target generation. In the above pipeline, flight maneuvers are generated by the local planner as controller targets for keeping the aircraft on the planned trajectories.

The existing global planner is useful for simple and fixed-path missions such as a race track flight path illustrated in Section V for surveying and power modeling. However, such missions limit the UAVs to fixed, position-based trajectories which prevent the aircraft from performing customized maneuvers aside from path-keeping. Therefore, in order to achieve more advanced autonomous UAV applications such as flight testing maneuver automation and geo-fencing, a more versatile mission planner is needed for planning and sequencing ad-hoc and customized flight maneuvers.

The new maneuver planner extends and augments the uavAP planning and control stack to achieve customized and flexible planning. Specifically, the maneuver planner generates user-defined flight maneuvers into override objects through the cpsCore configuration framework detailed in Section III. Such maneuvers, containing local plans, controller targets, controller outputs, and more, are published through cpsCore's IPC, as illustrated in Figure 10, to the respective modules, in which the regular mission is overridden until after executing the received flight maneuvers.

In this design, the maneuver planner provides versatile, trajectory-independent capabilities in terms of aircraft states. For instance, if a 45 degree right-rolling maneuver were needed for a particular application, the maneuver planner would simply generate a flight maneuver containing a roll angle controller target of 45 degrees and publishes such maneuver to the controller module for execution. When the maneuver was executed, the aircraft would override the regular trajectory and roll right at 45 degrees from its current state in a trajectory-free manner.

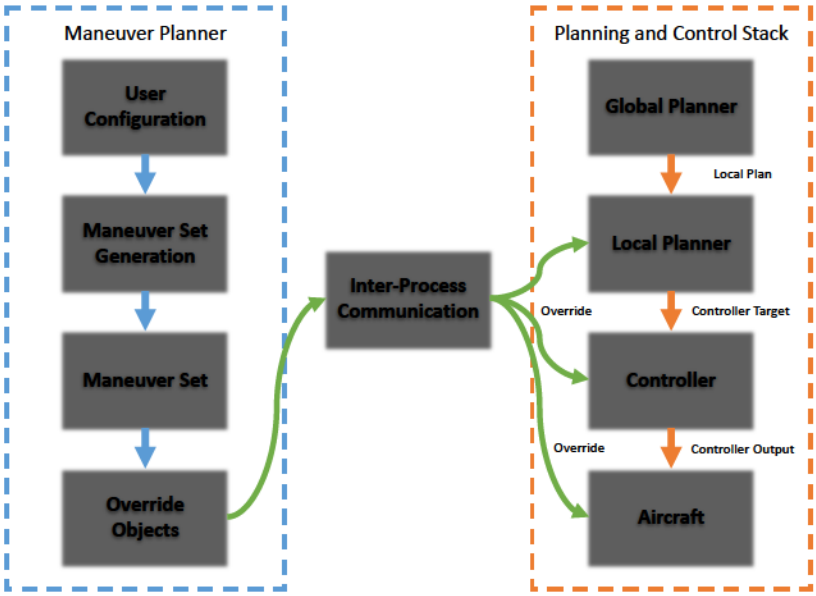


Figure 10. The maneuver planner pipeline and the uavAP planning and control stack (mission planner omitted). The override objects, representing the maneuvers, are generated by the maneuver planner, published through the cpsCore IPC framework discussed in Section III, and are subscribed by the uavAP stack for maneuver execution.

Furthermore, the maneuver planner is also capable of concatenating a series of individual flight maneuvers into a maneuver set, as shown in Figure 11, and executing through the set sequentially under some predefined transition conditions. Similar to a finite state machine (FSM), the maneuver planner stays at the current maneuver in a maneuver set and only continues to the next maneuver when the transition condition is met. When all the maneuvers in the maneuver set are exhausted, the maneuver planner halts, and the aircraft returns to its regular mission.

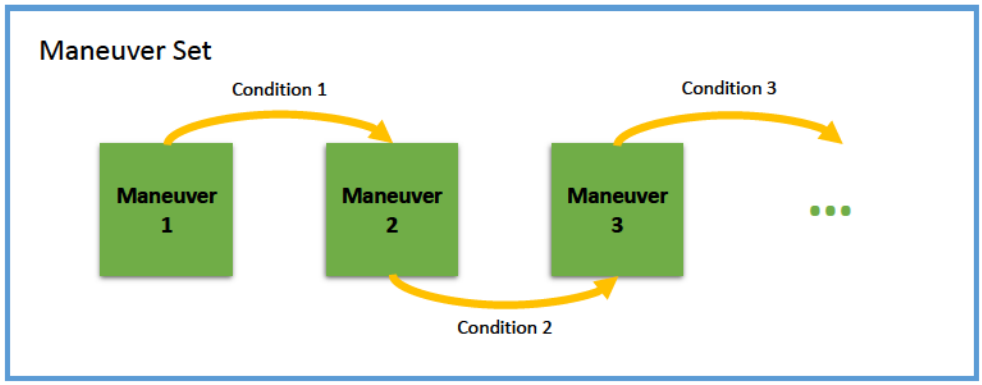


Figure 11. An example of a maneuver set, containing a series of individual flight maneuvers connected by their transition conditions.

B. Flight Analysis

For automating customized flight maneuvers with various traits, a new flight analysis process is needed in addition to the maneuver planner for analyzing various aspects of the aircraft during the flight. First of all, the analysis data provided by the flight analysis module can be used by either the uavAP modules during the flight or by post-processing programs through data collections after the flight. For example, during the execution of a particular maneuver set with

the maneuver planner, states of the aircraft are useful for post-processing, graphical visualization, scientific research and validation, and so on.

More importantly, automating the customized flight maneuvers often requires information about various aircraft states during the flight, such as whether the controllers have reached their steady states, aircraft control surface trims, how much time has elapsed since the start of the current maneuver, etc.[5] Maneuvers in many applications should transition to the next maneuver only when the roll angle controller of the aircraft has reached its steady state, i.e., the controller has stabilized around its given target. The flight analysis module would provide this steady-state information and enable the enforcement of such transitions.

V. Applications

The uavAP autopilot has thus far been applied to 3 platforms: the prototyping ecosystem emulation environment, uavEE; a robust, fixed-wing, testbed unmanned aircraft, the UIUC Avistar UAV; and a long-endurance, computationally-intensive, solar-powered unmanned aircraft, the UIUC-TUM Solar Flyer. Between these 3 platforms, the uavAP autopilot has enabled: the rapid prototyping of flight modeling and control algorithms in emulation and real flight, the design of an accurate UAV power model based on the physical model of the aircraft, flight maneuver automation for aircraft system identifications and dynamics parametrization, an algorithm for geo-fencing of fixed-wing UAVs, and power-efficient flight through a turbulent and windy environment.

A. uavEE Emulation Environment

In order to decrease development time, emulation and modeling has become an important component of the UAV development process. Instead of prototyping, testing, and analyzing through the many stages of aircraft development in hardware, which is resource and time intensive, a virtual aircraft and its sub-systems were modeled and then implemented into the uavEE emulation environment.[4] Specifically, the environment starts by creating a real-time connection between a high-fidelity flight simulator (e.g. X-Plane 11) and an autopilot software, i.e., uavAP on a desktop machine or embedded hardware, and then modeling layers are introduced (e.g. power, communication, fault, etc.), allowing for additional emulation complexity. Therefore, the physical aircraft design, the software, and the flight computation and possibly payloads can be tested in the lab. Within the scope of applying the uavAP autopilot to research tasks, uavEE was used to emulate each of the research efforts presented in the following subsections before they were tested on an actual aircraft. uavEE also provides the backbone for a ground control interface, shown in Figure 12, which is used to command and monitor the aircraft and autopilot in both emulation as well as in real flight.

B. Avistar UAV Testbed

The Avistar UAV is a highly-robust, fixed-wind unmanned aircraft, which has been used as the testbed plaform for a variety of flight software and hardware development efforts.[17–21] The aircraft was developed from the Great Planes Avistar Elite fixed-wing trainer-type radio control model and has wingspan of 1.59 m and a mass of 3.70 kg. The completed flight-ready aircraft is shown in Figure 13 and its physical and component specifications can be found in previous work.[22] The uavAP autopilot was integrated into the Al Volo flight control and data acquisition system installed in the Avistar UAV in order to enable several avenues of research: a high-fidelity, low-order propulsion power model for fixed-wing electric unmanned aircraft, a flight testing automation tool for aircraft system identifications and dynamics parametrization, and an algorithm for geo-fencing of fixed-wing UAVs.

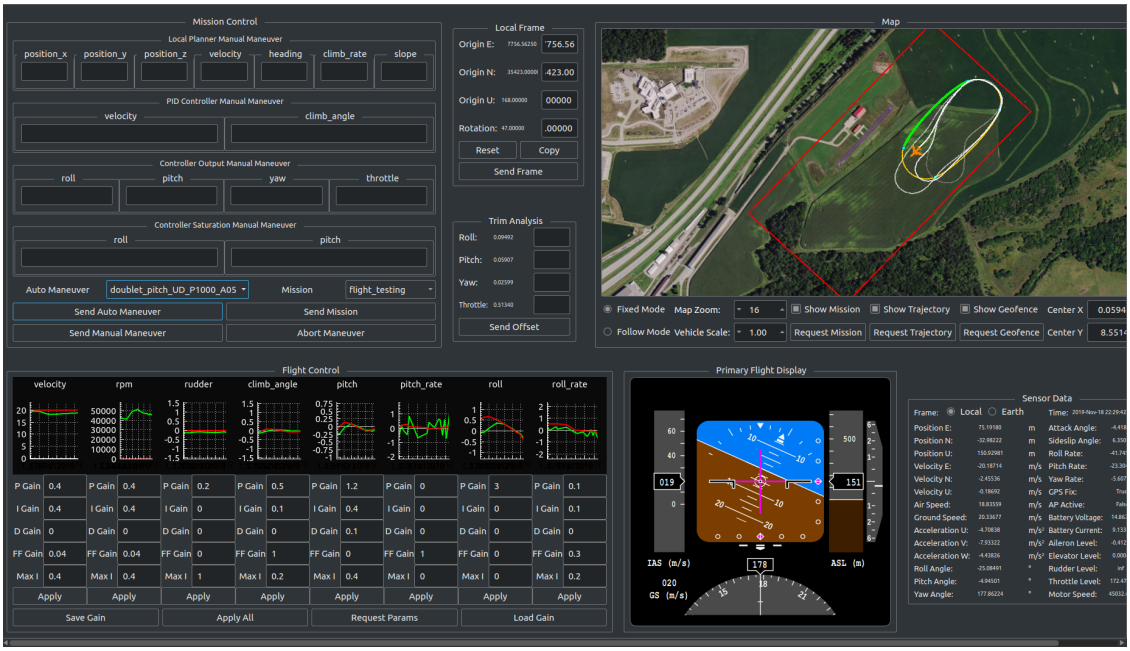


Figure 12. The uavEE ground station interface, which provides functionality in emulation and real flight; it is shown being used to automate a pitch doublet maneuver as part of the flight testing automation process.

Figure 13. Flight-ready Avistar UAV.

1. Propulsion Power Model for Fixed-Wing Electric Unmanned Aircraft

A high-fidelity, low-order power model for electric, fixed-wing unmanned aircraft[19] was developed and integrated into uavEE. Previous works have separately looked at aircraft power modelling[23–25] and propulsion system modelling[26–28] with varying degrees of assumptions and verification. Compared to existing works, the propulsion power model developed provides a more holistic approach to UAV propulsion power modeling and has been tested under realistic flight conditions. The power model uses propulsion system modeling of the propeller and motor as well as aircraft power modeling using flight mechanics derivations. In order to enable online computation with limited resources, the resulting expression has been limited to using only measurable aircraft state variables, propulsion system parameters and curves, and (scalar) constants. The final expression for the developed power model is:

$$P_{propulsion} = K_p \frac{v^3}{\eta_p \eta_m} + K_i \frac{\cos^2 \gamma}{\eta_p \eta_m v \cos^2 \phi} + mg \frac{v \sin \gamma}{\eta_p \eta_m} + m \frac{\vec{a} \cdot \vec{v}}{\eta_p \eta_m} \tag{5}$$

where K_p and K_i are scalar constants that can be determined from aircraft specifications or can be learned through linear regression with non-linear kernel using a training data set. Note that complete derivation and validation can be found in related work.[4,19]

The resulting power model was evaluated by means of flight testing using uavAP. By flying a reference flight path, containing turns, climbs, descents, and straight line segments, the flight testing showed very close agreement between the power and energy estimates determined using the power model from aircraft state data and actual experimental power and energy measurements. Additionally, using the emulation environment, the reference flight path was also flown using the same autopilot and a simulated radio control model aircraft trainer, which was very similar to the one used in experimental flight testing. These flight paths are displayed in Figure 14. The flight path was nearly identical with the exception of 2 corners, where in experimental flight testing, light wind gusts deviated the aircraft slightly. The power and energy data generated was in close agreement with the experimental data as can be seen in Figure 15. The significance of this result is that the developed propulsion power model is able to accurately estimate the power consumption of an electric UAV based on flight path state, without needing precise aerodynamic measurements or estimation, e.g. angle-of-attack. Therefore, power estimation can be done with minimal computation.

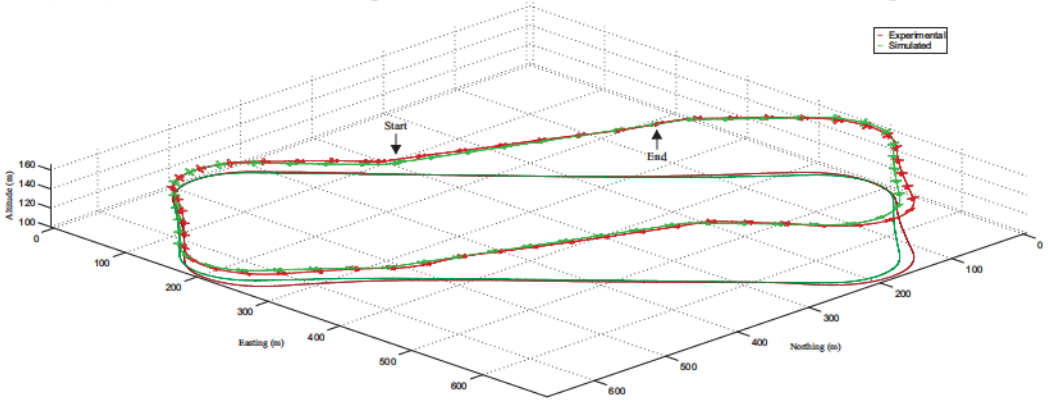
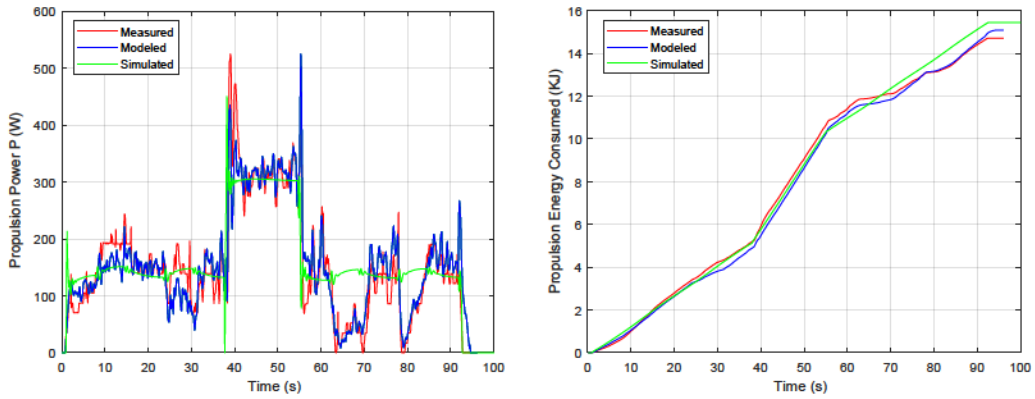


Figure 14. Comparison of aircraft path for experimental (red) and simulated flight (green) results; the airplane is plotted at 6x scale and every 2 seconds.



(a) (b)

Figure 15. Comparison of (a) propulsion power and (b) energy consumed from experimental measured (red), experimental modeled (blue), and simulated (green) results using the propulsion power model.

2. *Flight Testing Automation*

The current state-of-the-art in performing flight testing maneuvers for aircraft parameterization has been manual piloting with instruction relayed through flight test cards. Performing manual flight maneuvers off of manually read test cards has shown to require thousands of hours of costly flight testing.[29] There have been ongoing efforts to parameterize aircraft dynamics on manned and unmanned aircraft using multi-sine and stick-shaker inputs. However, these can be error-prone, very computationally intensive, and require large datasets.[30–32] Instead, the flight maneuver automation framework, described in Section IV, extends the existing uavAP stack to streamline the flight testing and flight dynamics

parameterization processes of an unmanned aircraft.[6] The flight maneuver automation framework is able to command the aircraft through a user-defined, conditional set of motions and states to induce certain maneuver sets, which allow for dynamics to be more easily parameterized; these sets of maneuvers, motions, and states follow manned flight testing techniques.[33] Maneuvers of interest that have been implemented into the automation framework included: idle descent, stall, phugoid, doublets, and singlets, which provide the basis for determining the aircraft aerodynamics, longitudinal stability, and control effectiveness, respectively. Additionally, automating the data collection process using the new flight analysis module allows for reliable data selecting, eliminating work hours of parsing and matching data ranges to maneuvers.

The flight maneuver automation framework was initially demonstrated using software-in-the-loop simulation in the uavEE. A comparison between automated and manually-piloted flight was performed for testing stall using the full-scale Cessna 172 under ideal (still atmosphere) conditions in the X-Plane 11 Flight Simulator[a]; this can be seen in Figure 16–19. In those time histories, the difficulty exhibited by the trained human pilot in simultaneously controlling the aircraft altitude and roll and heading angles can be seen. In comparison, the time history of autonomously controlled stall speed maneuver show smooth and accurate results. The flight maneuver automation framework was then demonstrated on the Avistar UAV testbed aircraft and subsequently used to collect an aircraft dataset. Due to limited calm weather day opportunities, only a subset of the maneuvers developed were flown, which include stall speed, stall polar, idle descent, singlets, e viewed in related work[21] and can b

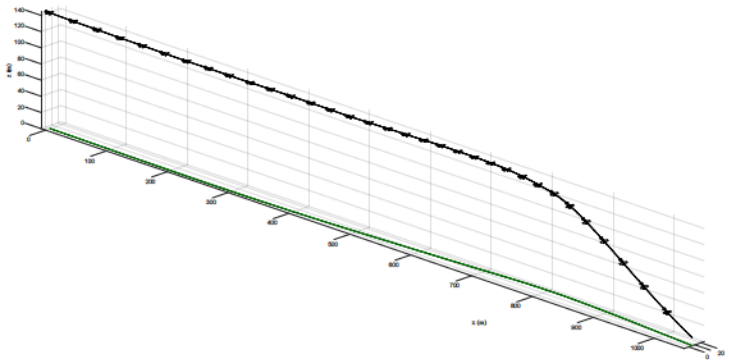


Figure 16. T ce every 1.0 s).

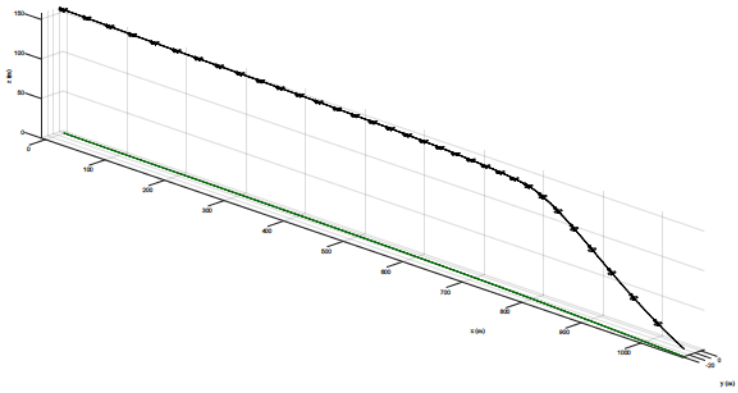


Figure 17. Trajectory plot of a stall speed maneuver performed by the flight maneuver automation framework (the aircraft is drawn once every 1.0 s).

[a]The manually-pilot aircraft was flown by a trained human pilot using a professional-grade simulator yoke system, throttle quadrant, and rudder pedals. Both maneuvers were set up the same, with the aircraft flying at 40 m/s and oriented at a yaw angle of 0 deg (East).

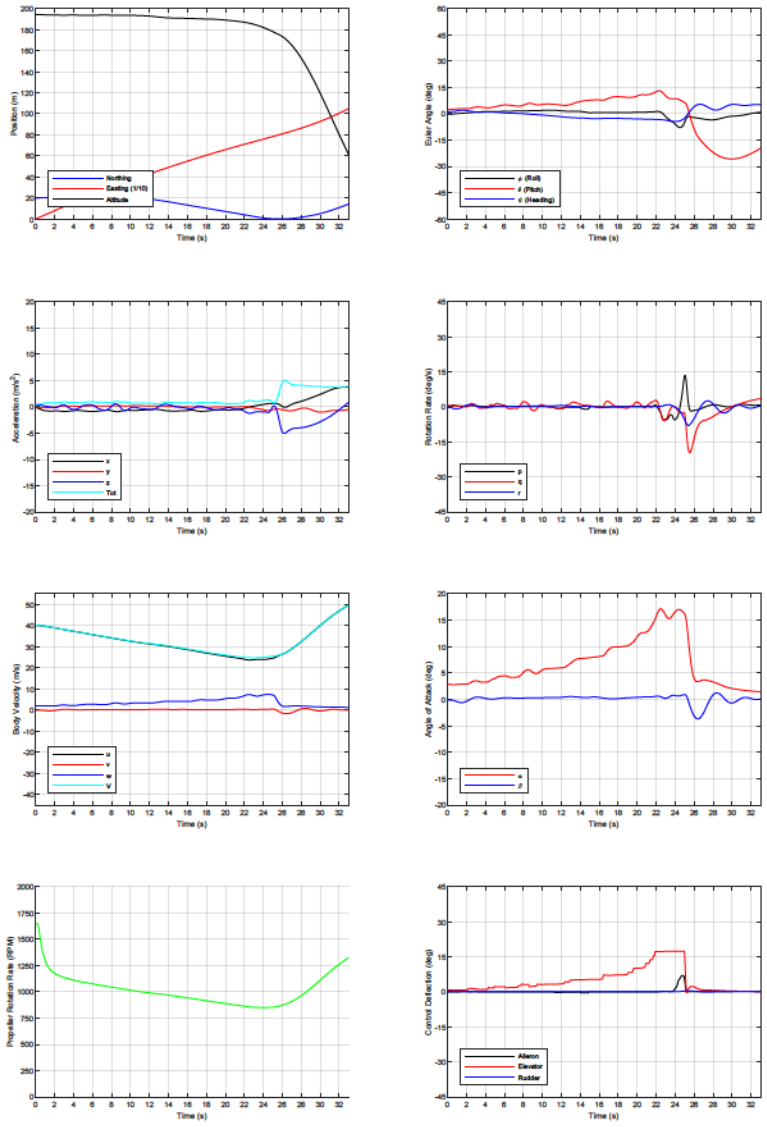


Figure 18. A time history of a stall speed maneuver performed by manual piloting.

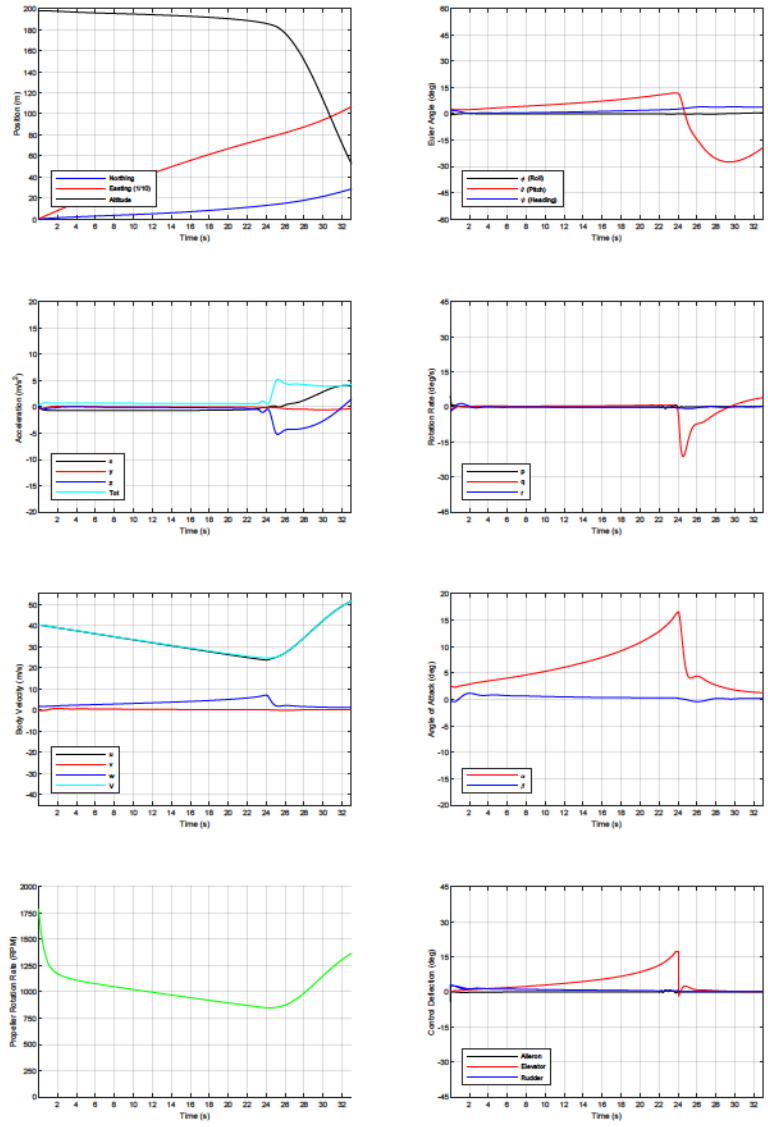


Figure 19. A time history of a stall speed maneuver performed by the maneuver automator.

3. *Geo-Fencing Algorithm*

To enable safe interactions among the surrounding humans, environments, and other aircraft, UAVs have to be constrained to designated areas or spaces. For rotary aircraft, such as quadcopters, the task of enforcing the geo-fence is relatively simple since those type of aircraft are capable of stopping in mid-air and turning around with zero translational velocity. However, for fixed-wing aircraft, such execution of maneuvers is impossible as they need to maintain a minimum velocity in order to stay airborne. Consequently, a precise kinematic model for fixed-wing aircraft is required to determine the feasibility of a trajectory as well as the exact time for the initiation of an evasion maneuver. Most analytical kinematic models only constrain the maximum curvature of a trajectory, namely a Dubin's Curve.[35,36] These fixed-wing aircraft geo-fencing algorithms[37,38] argue that the trajectories for a fixed-wing aircraft form a symmetric fan pattern around the velocity vector. This fan pattern, however, is based on the instantaneous change in roll, which was shown to not be applicable for fixed-wing aircraft.

Therefore, a precise kinematic model, the Beta-Trajectory, was developed for trajectory prediction and evasive maneuvering.[7] The Beta-Trajectory implements a kinematic model for fixed-wing aircraft where roll is governed by constrained roll rates, yielding Beta-Curves. The algorithm then uses the results of the model to avoid boundaries and stay in a designated area. The full derivation of the Beta-Trajectory can be found in this technical report.[16] In order to evaluate the proposed geo-fencing system, the Beta-Trajectory was implemented into uavAP using the flight maneuver automation framework, tested in uavEE, and then subsequently tested in real flights using the Avistar UAV. Figure 20 shows the performance of the Beta-Trajectory geo-fencing algorithm in real flights.

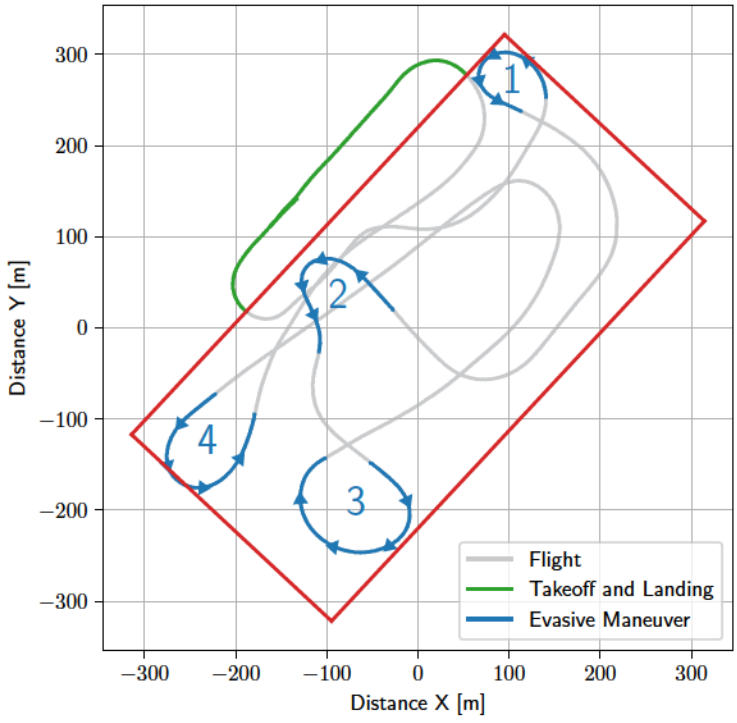


Figure 20. The real-life flight path of the Avistar UAV deployed with the uavAP autopilot with geo-fencing; red shows the geo-fence, blue shows the evasive maneuvers labeled from 1-4 (note that the boundary slack value is 5 meters and velocity is 20 m/s).

C. UIUC-TUM Solar Flyer

The UIUC-TUM Solar Flyer, which is shown in Figure 21, is a long-endurance, solar-powered unmanned aircraft currently in development to enable computationally-intensive flight that could support real-time data processing for a wide range of applications.[39] The aircraft is a highly-optimized, fixed-wing design that was assembled from only commercial-off-the-shelf (COTS) components and operates a narrow range of airspeeds in order to achieve highly-efficient flight. The aircraft has been instrumented with a custom Al Volo flight control and data acquisition system that integrates the uavAP autopilot, which has been adapted for the demanding requirements dictated by the aircraft and its flight profile.

Due to the low operating airspeed of the sailplane design, the aircraft is very susceptible to atmospheric turbulence and wind. Therefore it was crucial to integrate a responsive wind estimator and airspeed controller into uavAP. Among other flight testing recently performed, the UIUC-TUM Solar Flyer was autonomously flown using uavAP to measure the aircraft's power consumption,[40] which is crucial for modeling the aircraft. Additionally, in order to verify the aircraft's ability to maintain a precise flight path under varying flight conditions, the aircraft was flight tested using uavAP in various amounts of wind, up to the aircraft's typical cruise speed (note that the aircraft maximum speed is greater than cruise). Figure 22 shows the resulting trajectory, of the UIUC-TUM Solar Flyer attempting to maintain a repeated level race track maneuver under varying wind conditions, which would be sufficient to accomplish a typical mission profile, e.g. equivalent zig-zagged racetrack coverage profile over a field.

Figure 21. The UIUC-TUM Solar Flyer aircraft shown with solar arrays.

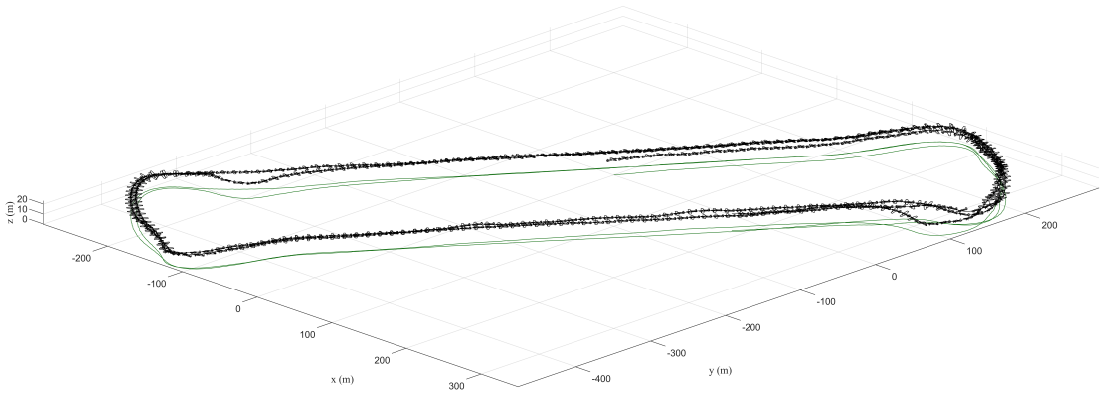


Figure 22. The trajectory of the UIUC-TUM Solar Flyer during a repeated level race track maneuver (note that the aircraft is plotted 6x scale every 0.5 sec).

VI. Related Autopilots

In this section, a comparison between uavAP and other openly available autopilot implementations is performed. The control stacks and software framework of Ardupilot, Paparazzi and PX4 are examined.

A. Ardupilot

Ardupilot is an open source software suite first establised in 2009. It is not tied to a specific set of hardware but rather, it is firmware capable of running on various embedded platforms.[41] Nevertheless, fully packaged autopilots with hardware and software, such as the The ArduPilot Mega (APM) based off the Arduino Mega, exist for convenience. APM has inspired many derivatives, such as FlyMaple.[42,43] Ardupilot's runtime stack is organized hierarchically as follows. The lowest layer is the hardware level, consisting of external sensors, open hardware standards such as Pixhawk,[44] controller chips such as the Mateksys F405[45] or Navio2, and complete drones such as the Bebop2.[46] The next level is the OS layer, which consists of OSs like ChibiOS, BusyBox Linux and Linux.[41] Ardupilot itself then runs above the OS layer. The flight code is further segmented into 3 layers, the hardware abstraction layer, shared libraries, and vehicle specific flight code. The hardware abstraction layer allows ArduPilot to be portable and platform agnostic. Shared libraries exist for the supported four vehicle types: Copter, Plane, Rover and AntennaTracker. The communication layer resides above the flight layer and communication is done via devices implementing the MAVLink protocol. In addition to these core autopilot modules, Ardupilot's codebase also has miscellaneous support tools. The highest layer of the runtime stack is the UI/API layer, which consists of the Ground Station and any DroneKit applications and their corresponding hosts.[41] In terms of software design, Ardupilot is focused on reliably going from one waypoint to another. Ardupilot provides a reliable trajectory planner to travel between waypoints and assumes the average user is not interested in modifying the trajectory planner or other critical features. uavAP allows users to modify the trajectory planner, scheduler, inter device communication, and various other low level features if they so desire. This modularity allows users to implement more complex solutions ranging from low level to high level. Thus, uavAP is more geared towards being a testbed for various state-of-the-art research implementations.

uavAP's runtime stack is comparable to Ardupilot's. At the hardware layer, uavAP uses Al Volo libraries to communicate with sensors and data acquisition systems but can be extended to other platforms. One noteworthy difference is that below the hardware level, uavAP can also be configured to take actuation and flight data from real flights or emulated/replayed flight conditions in uavEE. uavEE is then capable of performing both SITL (Software in the Loop) and HITL (Hardware in the Loop) Simulation, with flight simulators such as X-Plane 11. Conversely, Ardupilot has its own SITL (Software In The Loop) simulation framework and simulator. It is a build of the autopilot using any C++ compiler, and thus allows the autopilot to be tested without hardware. Ardupilot's SITL Simulator can also be used with a wide variety of 3rd party simulators, such as Gazebo, XPlane-10, RealFlight, Morse, Replay, JSBSIM, AirSim, Silent Wings Soaring, Last Letter, CRRCSim, or SCRIMMAGE. Hardware In The Loop Simulation is currently only supported for planes in X-Plane and FlightGear.[41] While Ardupilot's SITL necessitates manual connections from the autopilot executable to the ground control station, physics and flight simulators, and proxy telemetry if multiple ground control stations are desired, uavAP uses the uavEE system to handle all inter-agent communication associated with HITL/SITL simulation. Ardupilot's fixed wing operating modes are comparable to uavAP. Ardupilot has various different flight modes, ranging from full manual control of aircraft control surfaces, to roll and pitch override, to circling a point, to following a mission. In AUTO mode, where the autopilot flies a mission, Ardupilot's framework allows the ground station to update the mission.[41] Conversely in uavAP, manual flight is not supported by default. In the interest of autonomy, the mission is preconfigured ahead of the flight.

B. Paparazzi

Paparazzi UAV is another open-source project encompassing both the hardware and software aspects of UAV systems. As per their website, they support more target platforms than Ardupilot.[47] One example of a pre-built board running the autopilot software is the STM based Chimera board.[42,43,47] Paparazzi's usage flow is as follows. The autopilot is configurable by an XML, where the flight mode state machine is defined, along with aircraft modules, additional header files, ground control settings, and exceptions. The specified firmware is then built and cross-compiled for that target aircraft hardware, and uploaded to the embedded board. Paparazzi has a wide array of modules for performing tasks such as reading external sensors or controlling cameras. The default features for fixed wing aircraft are the following: manual control via an RC transmitter, RC receiver, servo and motor control, control with augmented stability (AUTO1), autonomous navigation (AUTO2), and communication to and from the ground station. Autonomous navigation includes features like waypoint navigation, segment and circle navigation, takeoff and landing, and advanced fail-safe planning (e.g. geo-fencing).[47] Its configurable state machine nature also allows high flexibility and complexity in algorithm design for control, communication and other custom features. In comparison, uavAP provides a concrete control stack with defined roles, with the goal of minimizing complexity associated with changing autopilot functionality.

The system communication flow is as follows. When configured for real flight, the aircraft communicates over a wireless link to a ground network, which then sends the data to a server that logs, distributes, and pre-processes the messages for the ground control station and other ground agents. When configured for simulated flight, the hardware communication link is replaced with a SITL simulator that simulates actuation and radio communication. A Gaia agent then allows the user to set environmental variables such as windspeed and direction, sensor failure, and flight simulation speed. Paparazzi has two built in simulators, sim and nps (New Paparazzi Simulator). It also supports the Gazebo simulator.[47] In comparison to uavAP, uavAP sends sensor data and receives ground commands in real flight over a radio link to a uavEE environment with radio communication and ground station nodes. In simulated flight, sensor data from the flight simulator is sent over a simulated serial link to uavAP and processed with the same API. Environmental factors such as wind can be configured in the flight simulator (currently X-Plane 11). All simulated sensor data can be corrupted to simulate sensor fault, and all communication between simulation peripherals (i.e. flight simulator, power modeller, sensor fault modeller) is handled by the ROS environment uavEE is built on.

C. PX4

PX4 is another open source autopilot for drones and other unmanned aerial vehicles focused on support for a broad category of aerial vehicles, sensors and control hardware, and safe flight modes. It comes with a ground station called QGroundControl, supports PixHawk hardware, and uses the MAVSDK library for communication with companion devices using the MAVLink protocol.[48] Various embedded boards are designed to use the PX4 autopilot, such as PIXHAWK, Pixfalcon, and PixRacer.[42,43] PX4 is split into a flight stack layer and middleware layer. The flight stack layer is responsible for all flight control tasks, such as guidance, navigation, control algorithms, and reading sensors. The workflow inside the flight stack is as follows. An estimator feeds a state estimate to a controller, which produces a command, and a mixer translates them to motor commands. This layer is vehicle specific and depends on factors such as the aircraft's motor arrangements and rotational inertia. PX4 uses a state machine in the flight controller to select a flight controller based on the level of flight autonomy desired.[48] The middleware layer handles communication with the external world and it includes device drivers for embedded sensors and communication with companion computers, ground control stations, etc. Similar to Ardupilot, PX4 provides broad community support and is focused on simple and reliable flight control. Thus, PX4 focuses on providing robust semi-autonomous flight (e.g. pitch and roll controlled by the autopilot but yaw manually controlled by RC stick) and autonomous waypoint following and assumes the average user is not interested in modifying the trajectory planner or other critical underlying features. In contrast, uavAP's

modular framework is designed to allow modifications to any underlying feature, making it more suitable to be a testbed for various state-of-the-art research implementations.

In addition, PX4 can be interfaced to run on a computer modeled vehicle in a simulated flight world. Currently for fixed wing aircraft, PX4 supports the Gazebo simulator for SITL and HITL and X-Plane for HITL only. When doing SITL simulation, PX4 communicates with offboard APIs, the ground control station and the simulator over the MAVLink protocol on UDP. Faster than real-time and lockstep simulation is also supported, as well as joystick integration, sensor failure, and camera simulation.[48] When uavAP communicates with uavEE and vice versa in SITL, HITL and real flight, point to point serial communication with CRC is used. Faster than real-time simulation playback is also supported in the uavEE environment via ROS-bags (saved flight data) and accelerated X-Plane simulation speed.

VII. Conclusion and Future Work

This work presented uavAP, a modular autopilot for UAVs, providing some details of its control stack implementation as well as applications. uavAP has been used in past research for the design of an accurate UAV power model, a flight maneuver automation framework, and an accurate kinematic model and algorithm for fixed-wing aircraft geo-fencing. Its core, cpsCore, is the C++ object-oriented backbone, used for module management such as configuration, aggregation, and synchronization. In essence, uavAP is a collection of modules merged together using cpsCore to form a flexible and distributed autopilot framework for UAVs.

In future work, uavAP will be applied to a broad range of research directions. The critical computation path of flight control provides a challenge for real-time system management, especially when parallelizing it with data-intensive vision computation. Providing real-time guarantees requires complex software isolation techniques, which can be implemented and tested in uavAP. Further work can be conducted with planning and control algorithms, branching out into trajectory optimization or power-optimal flight using techniques of artificial intelligence. While expanding the system to intelligent, unpredictable algorithms, a pairing of those algorithms with reliable, less complex algorithms might be essential. This architectural challenge can easily be addressed with uavAP, in which the intelligent algorithm can even be isolated as its own process. Another branch of research with the need for modularity and flexibility is the area of multi-agent systems, specifically multi-agent UAVs. In this field, uavAP can be used to facilitate the testing and development of various communication schemes, consensus algorithms, or even multi-agent reinforcement learning. As it is an open-source project, uavAP aims to expand into more research communities, with the goal to be a testbed of state-of-the-art research.

Acknowledgments

The material presented in this paper is based upon work supported by the National Science Foundation (NSF) under grant number CNS-1646383. Marco Caccamo was also supported by an Alexander von Humboldt Professorship endowed by the German Federal Ministry of Education and Research. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the NSF.

References

[1]Mirco Theile, "uavAP: A Modular Autopilot for Unmanned Aerial Vehicles," https://github.com/theilem/uavAP, Accessed May 2020.

[2]Al Volo LLC, "Al Volo: Flight Data Acquisition Systems," http://www.alvolo.us.

[3]Mirco Theile, "uavEE: A Modular Emulation Environment for Rapid Development and Testing of Unmanned Aerial Vehicles," https://github.com/theilem/uavEE, Accessed May 2020.

[4]Theile, M., Dantsker, O. D., Nai, R., and Caccamo, M., "uavEE: A modular, power-aware emulation environment for rapid prototyping and testing of uavs," *2018 IEEE 24th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, IEEE, 2018, pp. 217–224.

[5]Yu, S., *Flight Maneuver Automation for System Analysis of Small Fixed-Wing UAVs*, Bachelor's thesis, University of Illinois at Urbana-Champaign, Department of Electrical and Computer Engineering, Urbana, IL, 2019.

[6]Dantsker, O. D., Yu, S., Vahora, M., and Caccamo, M., "Flight Testing Automation to Parameterize Unmanned Aircraft Dynamics," AIAA Paper 2019-3230, AIAA Aviation and Aeronautics Forum and Exposition, Dallas, Texas, June 2019.

[7]Theile, M., Yu, S., Dantsker, O. D., and Caccamo, M., "Trajectory Estimation for Geo-Fencing Applications on Small-Size Fixed-Wing UAVs," *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2019, pp. 1971–1977.

[8]Twigg, C., "Catmull-rom splines," *Computer*, Vol. 41, No. 6, 2003, pp. 4–6.

[9]Mirco Theile, "Modular C++ Framework for Cyber-Physical Systems," https://github.com/theilem/cpsCore, Accessed May 2020.

[10]Redislabs, "Redis," https://redis.io/, Accessed May 2020.

[11]Dantsker, O. D., Ananda, G. K., and Selig, M. S., "GA-USTAR Phase 1: Development and Flight Testing of the Baseline Upset and Stall Research Aircraft," AIAA Paper 2017-4078, AIAA Applied Aerodynamics Conference, Denver, Colorado, June 2017.

[12]Regan, C. D. and Taylor, B. R., "mAEWing1: Design, Build, Test - Invited," AIAA Paper 2016-1747, AIAA Atmospheric Flight Mechanics Conference, San Diego, California, Jun. 2016.

[13]Bunge, R. A., Alkurdi, A. E., Alfaris, E., and Kroo, I. M., "In-Flight Measurement of Wing Surface Pressures on a Small-Scale UAV During Stall/Spin Maneuvers," AIAA Paper 2016-3652, AIAA Flight Testing Conference, Washington, D.C., Jun. 2016.

[14]Bunge, R. A., Savino, F. M., and Kroo, I. M., "Approaches to Automatic Stall/Spin Detection Based on Small-Scale UAV Flight Testing," AIAA Paper 2015-2235, AIAA Atmospheric Flight Mechanics Conference, Dallas, Texas, Jun. 2015.

[15]Ragheb, A. M., Dantsker, O. D., and Selig, M. S., "Stall/Spin Flight Testing with a Subscale Aerobatic Aircraft," AIAA Paper 2013-2806, AIAA Applied Aerodynamics Conference, San Diego, CA, June 2013.

[16]M. Theile and S. Yu, "Kinematic Model for Fixed-Wing Aircraft with Constrained Roll-Rate," Tech. rep., University of Illinois at Urbana-Champaign, Department of Computer Science, Sep. 2018.

[17]Mancuso, R., Dantsker, O. D., Caccamo, M., and Selig, M. S., "A low-power architecture for high frequency sensor acquisition in many-DOF UAVs," *2014 ACM/IEEE International Conference on Cyber-Physical Systems (ICCPS)*, IEEE, 2014, pp. 103–114.

[18]Dantsker, O. D., Loius, A. V., Mancuso, R., Caccamo, M., and Selig, M. S., "SDAC-UAS: A Sensor Data Acquisition Unmanned Aerial System for Flight Control and Aerodynamic Data Collection," *AIAA Infotech@Aerospace Conference, Kissimee, Florida, Jan 2015.*.

[19]Dantsker, O. D., Theile, M., and Caccamo, M., "A High-Fidelity, Low-Order Propulsion Power Model for Fixed-Wing Electric Unmanned Aircraft," AIAA Paper 2018-5009, AIAA/IEEE Electric Aircraft Technologies Symposium, Cincinnati, OH, July 2018.

[20]Dantsker, O. D., Imtiaz, S., and Caccamo, M., "Electric Propulsion System Optimization for a Long-Endurance and Solar-Powered Unmanned Aircraft," AIAA Paper 2019-4486, AIAA/IEEE Electric Aircraft Technology Symposium, Indianapolis, Indiana, Aug. 2019.

[21]Dantsker, O. D., Caccamo, M., Theile, M., and Mancuso, R., "Flight & Ground Testing Data Set for an Unmanned Aircraft: Great Planes Avistar Elite," AIAA Paper 2020-0780, AIAA SciTech Forum, Orlando, Florida, Jan 2020.

[22]Dantsker, O. D., Mancuso, R., Selig, M. S., and Caccamo, M., "High-Frequency Sensor Data Acquisition System (SDAC) for Flight Control and Aerodynamic Data Collection," *32nd AIAA Applied Aerodynamics Conference*, 2014, p. 2565.

[23]Lee, J. S. and Yu, K. H., "Optimal Path Planning of Solar-Powered UAV Using Gravitational Potential Energy," *IEEE Transactions on Aerospace and Electronic Systems*, Vol. 53, No. 3, June 2017, pp. 1442–1451.

[24]Grano-Romero, C., García-Juárez, M., Guerrero-Castellanos, J. F., Guerrero-Sánchez, W. F., Ambrosio-Lázaro, R. C., and Mino-Aguilar, G., "Modeling and control of a fixed-wing UAV powered by solar energy: An electric array reconfiguration approach," *2016 13th International Conference on Power Electronics (CIEP)*, June 2016, pp. 52–57.

[25]Hosseini, S., Dai, R., and Mesbahi, M., "Optimal path planning and power allocation for a long endurance solar-powered UAV," *2013 American Control Conference*, June 2013, pp. 2588–2593.

[26]Karabetsky, D., "Solar rechargeable airplane: Power system optimization," *2016 4th International Conference on Methods and Systems of Navigation and Motion Control (MSNMC)*, Oct 2016, pp. 218–220.

[27]Lindahl, P., Moog, E., and Shaw, S. R., "Simulation, Design, and Validation of an UAV SOFC Propulsion System," *IEEE Transactions on Aerospace and Electronic Systems*, Vol. 48, No. 3, JULY 2012, pp. 2582–2593.

[28]Bradt, J. B. and Selig, M. S., "Propeller Performance Data at Low Reynolds Numbers," *AIAA Paper 2011-1255, AIAA Aerospace Sciences Meeting, Orlando, Florida, Jan. 2011.*

[29]Canin, D. G., McConnell, J. K., and James, P. W., "F-35 High Angle of AttackFlight Control Development and Flight Test Results," AIAA Paper 2019-3227, AIAA Aviation and Aeronautics Forum and Exposition, Dallas, Texas, June 2019.

[30]Morelli, E., "Flight Test Validation of Optimal Input Design and Comparison to Conventional Inputs," AIAA Paper 1997-3711, AIAA Atmospheric Flight Mechanics Conference, New Orleans, Louisiana, Aug. 1997.

[31]Sobron, A., *On Subscale Flight Testing: Applications in Aircraft Conceptual Design*, Ph.D. thesis, Linkoping University, Department of Management and Engineering, Linkoping, Sweden, 2018.

[32]Grauer, J. A. and Boucher, M., "Aircraft System Identification from Multisine Inputs and Frequency Responses," AIAA Paper 2020-0287, AIAA SciTech Forum, Orlando, Florida, Jan. 2020.

[33]Kimberlin, R. D., *Flight Testing of Fixed-Wing Aircraft*, AIAA Education Series, AIAA, Reston, VA, 2003.

[34]O. D. Dantsker and R. Mancuso and M. Vahora and M. Caccamo, "Unmanned Aerial Vehicle Database," http://www.uavdb.org.

[35]Dubins, L. E., "On Curves of Minimal Length with a Constraint on Average Curvature, and with Prescribed Initial and Terminal Positions and Tangents," *American Journal of Mathematics*, Vol. 79, No. 3, 1957, pp. 497–516.

[36]Lugo-Cárdenas, I., Flores, G., Salazar, S., and Lozano, R., "Dubins path generation for a fixed wing UAV," *2014 International Conference on Unmanned Aircraft Systems (ICUAS)*, May 2014, pp. 339–346.

[37]Gurriet, T. and Ciarletta, L., "Towards a generic and modular geofencing strategy for civilian UAVs," *2016 International Conference on Unmanned Aircraft Systems (ICUAS)*, June 2016, pp. 540–549.

[38]Dill, E. T., Young, S. D., and Hayhurst, K. J., "SAFEGUARD: An assured safety net technology for UAS," *2016 IEEE/AIAA 35th Digital Avionics Systems Conference (DASC)*, Sept 2016, pp. 1–10.

[39]Dantsker, O. D., Theile, M., Caccamo, M., and Mancuso, R., "Design, Development, and Initial Testing of a Computationally-Intensive, Long-Endurance Solar-Powered Unmanned Aircraft," AIAA Paper 2018-4217, AIAA Applied Aerodynamics Conference, Atlanta, Georgia, Jun. 2018.

[40]Dantsker, O. D., Theile, M., Caccamo, M., Yu, S., Vahora, M., and Mancuso, R., "Continued Development and Flight Testing of a Long-Endurance Solar-Powered Unmanned Aircraft: UIUC-TUM Solar Flyer," AIAA Paper 2020-0781, AIAA Scitech 2020 Forum, Orlando, Florida, Jan 2020.

[41]"Ardupilot Autopilot suite," http://ardupilot.org, 2019.

[42]Ebeid, E., Skriver, M., and Jin, J., "A survey on open-source flight control platforms of unmanned aerial vehicle," *2017 Euromicro Conference on Digital System Design (DSD)*, IEEE, 2017, pp. 396–402.

[43]Ebeid, E., Skriver, M., Terkildsen, K. H., Jensen, K., and Schultz, U. P., "A survey of open-source UAV flight controllers and flight simulators," *Microprocessors and Microsystems*, Vol. 61, 2018, pp. 11–20.

[44]"pixhawk — The Hardware Standard for Open Source Autopilots," https://pixhawk.org/.

[45]"Matek Systems," http://www.mateksys.com/.

[46]"Parrot Bepop 2 FPV Drone," https://www.parrot.com/global/drones/parrot-bebop-2-fpv.

[47]http://wiki.paparazziuav.org/wiki/Overview, 2018.

[48]"PX4 Documentation," https://docs.px4.io/, 2020.