Embedding Adaptive Features in the ArduPilot Control Architecture for Unmanned Aerial Vehicles

Li, Peng; Liu, Di; Xia, Xin; Baldi, S.

DOI
10.1109/CDC51059.2022.9993292

Publication date
2022

Document Version
Final published version

Published in
Proceedings of the IEEE 61st Conference on Decision and Control (CDC 2022)

2022 IEEE 61st Conference on Decision and Control (CDC)
December 6-9, 2022. Cancún, Mexico

Embedding Adaptive Features in the ArduPilot Control Architecture for Unmanned Aerial Vehicles

Peng Li, Di Liu, Xin Xia, and Simone Baldi

Abstract— **The operation of Unmanned Aerial Vehicles (UAVs) is often subject to state-dependent alterations and unstructured uncertainty factors, such as unmodelled dynamics, environmental weather disturbances, aerodynamics gradients, or changes in inertia and mass due to payloads. While a large number of autopilot solutions have been proposed to operate UAVs, none of these solutions is able to counteract the effects of state-dependent and unstructured uncertainties online by parameter estimation and adaptive control techniques. This work presents a systematic integration of adaptive control into ArduPilot, a popular open-source autopilot suite maintained by a large community of UAV developers. Adaptation features are embedded in the ArduPilot control structure without altering the original architecture, to allow users to use the autopilot suite as usual. Tests show that the proposed adaptive ArduPilot provides consistent improved performance in several uncertain flight conditions. The source code of the proposed adaptive ArduPilot is released at `https://github.com/Friend-Peng/Adaptive-ArduPilot-Autopilot`.**

I. Introduction

The possible applications of Unmanned Aerial Vehicles (UAVs) have been increasing steadily, so that several companies and research centers are experimenting the use of UAVs in daily-life tasks, e.g. for last-mile delivery of goods, monitoring of traffic systems, inspection of infrastructures, and so on. At the same time, realizing these tasks is not trivial, as the existence of gaps in guidance, navigation, and control of UAVs is recognized even in simpler tasks [1]. Nowadays, a large number of off-the-shelf autopilots propose guidance, navigation, and control solutions for UAVs: examples include PX4, ArduPilot, AscTec, Navio2, among others [2]–[4]. Despite these autopilots being maintained by large communities, their capabilities of dealing with uncertainties in the UAV or in its environment are still limited [5]. These limits become more substantial as the tasks become more complex and unstructured.

This project has received funding from European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 899987. It was also supported by Natural Science Foundation of China grant 62073074, by Special Funding for Overseas grant 6207011901, and by Double Innovation Plan grant 4207012004. *(corresponding authors: S. Baldi and D. Liu).*

P. Li is with the School of Cyber Science and Engineering, Southeast University, Nanjing, China (email: `lpeng_2013@163.com`)
D. Liu is with School of Computation, Information and Technology, Technical University of Munich, Germany, and also with School of Cyber Science and Engineering, Southeast University, China (email: `di.liu@tum.de`)
X. Xia is with the School of Mathematics, Southeast University, Nanjing, China (email: `xiaxin0209@gmail.com`)
S. Baldi is with the School of Cyber Science and Engineering, Southeast University, Nanjing, China, and also with the Delft Center for Systems and Control, TU Delft, the Netherlands (email: `s.baldi@tudelft.nl`)

A typical example of uncertainty is the presence of payloads suspended or attached to the UAV. The operational parameters of these payloads (payload mass and geometry, cable length, etc.) are typically unknown in advance. As these parameters alter the flight dynamics considerably, the autopilot controllers need to minimize the alteration effects. However, users repeatedly experience that the controllers of off-the-shelf autopilots are unable to deal with such alterations and require continuous tuning to perform satisfactorily for different payloads. In fact, off-the-shelf autopilots presently rely on linear control techniques that can only cope with specific nominal flight conditions: changing such conditions requires to change the gains of the controllers to avoid loss of performance or even instability [6], [7]. The recent survey on guidance, navigation, and control for UAVs [1] confirms that currently there is no autopilot explicitly designed towards adaptive operation, e.g. capable to elaborate the information of the current UAV dynamics and to autonomously reconfigure the control action by adaptive control techniques [8], [9]. A systematic use of these techniques would make the autopilot able to cope with uncertainty and unmodelled dynamics in the UAV or in its environment. This work presents a systematic integration of adaptive control into ArduPilot, a popular off-the-shelf autopilot. The main reason for considering ArduPilot is its open-source nature: arguably, ArduPilot and PX4 are the autopilot systems maintained by the largest communities of UAV developers. ArduPilot and PX4 autopilots can be interfaced with several open-source tools, such as the Robot Operating System (ROS) or the Micro Air Vehicle Link (MAVLink) protocol [10], [11].

Guaranteeing stability and robustness in unstructured flight scenarios are challenges in adaptive control of UAVs [12], [13]: even though some proofs exist to show stability against the adverse effects from parametric uncertainties and external perturbations, crucial *structural* assumptions are made on such uncertainties and perturbations. Most often, uncertainties and perturbations are assumed to be bounded *a priori*. However, the authors have shown in previous work that unmodelled dynamics of UAVs are state-dependent (i.e. cannot be bounded a priori) and require appropriate adaptive control designs [14], [15]. The main contribution of the present work is to show that provably stable adaptive control tools for state-dependent unstructured uncertainties can be seamlessly integrated in ArduPilot without modifying its basic architecture. The integration is presented for fixed-wing UAVs (i.e. the ArduPlane module of ArduPilot), but we expect that a similar integration is possible for any aerial or non-aerial vehicles for which ArduPilot or PX4 have been thought, such as spherical robots and hybrid rolling/flying systems [16], [17]. In fact, all these modules rely on a similar cascaded linear control architecture, which is amenable to the proposed adaptive solution. Hence, we argue that the solution proposed in this work can impact several types of unmanned vehicles and autonomous systems.

The work is organized as follows: Section II presents related works on adaptive autopilots; Section III gives information about the original open-source ArduPilot architecture; Section IV shows how adaptation is embedded without modifying the original architecture. Section V shows with comparative experiments that the proposed adaptive autopilot outperforms the original one. Conclusions are in Section VI.

II. Related Work

The importance of adaptation in UAVs is well known in the literature. Adaptation to unknown payloads was studied in [18] and implemented in PX4 via a model reference adaptive control technique. The possibility of using non-recursive and recursive parameter estimation on-board of UAVs was examined in [19]. Adaptation of UAVs to different wind conditions has also been the subject of several studies, usually relying on compensating for the offset position error induced by the wind disturbance [20]–[22]. Adaptation in formations of UAVs was studied in [23], [24]. These and other works show the importance of overcoming the linear control implementation of off-the-shelf autopilots.

Software-in-the-loop or hardware-in-the-loop UAV platforms have been proposed based on open-source firmware, such as ANT-X [25] or RflySim [26], based on PX4 and MATLAB. These platforms are customizable by the user for education and research, e.g. the users can directly use MATLAB to design low-level controllers (attitude, position control) and high-level applications (decision-making, autonomous flight), with no need to access the C/C++ underlying autopilot code. However, these customizable platforms do not provide adaptive control features, which require indeed to access the C/C++ underlying autopilot code.

At the same time, researchers proposed their own autopilot architectures, either based on model predictive control [27], underactuated Euler-Lagrange dynamics [28] or deep reinforcement learning [29], among others. These autopilots require completely different architectures, which cannot operate along with the original off-the-shelf architecture. Adaptation methods based on these architectures, such as adaptive predictive control [30], disturbance observer control based on Euler-Lagrange dynamics [31], or deep learning based on neural networks [32], require to substantially modify or completely replace the original open-source autopilot.

The approach we adopt departs from these philosophies. We aim to keep the same open-source ArduPilot control architecture, while making it adaptive. The approaches we are aware of that are most in line with our philosophy are [33], [34]. The authors of [33] proposed a two-step procedure consisting of UAV system identification and Proportional-Integral-Derivative (PID) gain optimization. The objective is to find the optimal PID gains for the same original control structure of the PX4 flight stack. The authors of [34] implement a retrospective cost adaptive controller on PX4, paying attention that the designed controller integrates well with the existing PX4 firmware. Despite the progress presented in these works, the uncertainty is a standard parametric uncertainty and the crucial point of state-dependent unstructured uncertainty is overlooked there. Indeed, the stability of adaptive autopilot systems in the presence of complex state-dependent uncertainties is missing in the current literature.

TABLE I
Main Variables in ArduPilot Control Architecture

Name	Description
`_SPE_est`	Estimate of specific potential energy
`_SKE_est`	Estimate of specific kinetic energy
`_STE`	Specific total energy
`_SEB`	Specific energy balance
`_SPE_dem`	Demanded specific potential energy
`_SKE_dem`	Demanded specific kinetic energy
`_STE_dem`	Demanded specific total energy
`_SEB_dem`	Demanded specific energy balance
`desired_rate_r`	Desired roll angle rate
`achieved_rate_r`	Measured roll angle rate
`desired_rate_p`	Desired pitch angle rate
`achieved_rate_p`	Measured pitch angle rate
`accel_y`	Measured lateral acceleration
`omega_z`	Measured yaw angle rate

III. Background

Fixed-wing UAVs are complex underactuated systems, with six degrees-of-freedom (three positions and three orientation angles) controlled via four inputs (aileron, elevator, rudder, throttle). Complexity arises from the fact that longitudinal and lateral dynamics are coupled, and that many parameters and aerodynamics coefficients related to the flight dynamics are difficult to obtain. To deal with such complexity, virtually all off-the-shelf autopilots use PID control because of its simplicity and low computational features. The fixed-wing UAV control architectures in ArduPilot and PX4 (as well in other autopilots) are organized according to two layers: the Total Energy Control System (TECS) to control the throttle and pitch demand, and the attitude layer to control roll, pitch and yaw (cf. Fig. 1). Both layers utilize control loops closed with PID gains. In the following, we describe such control architecture. When possible, we use the variable names as they are found in the ArduPilot source code [2]: the most important variables are reported in Table I.

A. Total Energy Control System (TECS)

The Total Energy Control System comprises 2 PID loops:

- One PID loop aims to control the throttle demand `_throttle_dem` by regulating the total energy of the UAV (potential energy plus kinetic energy);
- Another PID loop aims to control the pitch demand `_pitch_dem` by regulating the energy balance of the UAV (potential energy minus kinetic energy).

The rationale of the TECS architecture is that the throttle increases the rate of the total energy of the UAV, whereas pitching allows the UAV to exchange potential energy into kinetic energy (and vice versa). By convention, in ArduPilot the energies are defined as

$$\begin{aligned}&\texttt{\_SPE\_est} = \texttt{\_height} * \texttt{GRAVITY\_MSS},\\ &\texttt{\_SKE\_est} = 0.5 * \texttt{\_TAS\_state} * \texttt{\_TAS\_state},\end{aligned} \tag{1}$$

where `_SPE_est` and `_SKE_est` stand for estimates of specific potential energy and specific kinetic energy, respectively, being `_height` the altitude of the UAV, `GRAVITY_MSS` the gravity coefficient, and `_TAS_state` the UAV true air speed. The UAV mass is not included in the energy, hence the term "specific energy". The specific total energy and specific energy balance are defined as

$$\begin{aligned}&\texttt{\_STE} = \texttt{\_SPE\_est} + \texttt{\_SKE\_est},\\ &\texttt{\_SEB} = \texttt{\_SPE\_est} - \texttt{\_SKE\_est},\end{aligned} \tag{2}$$

Accordingly, one can define a specific total energy demand and specific energy balance demand based on the demanded altitude `_hgt_dem_adj` and demanded true air speed `_TAS_dem_adj`, which are called like this because ArduPilot "adjusts" online the demanded altitude and air speed to respect some safety bounds. We have

$$\begin{aligned}&\texttt{\_SPE\_dem} = \texttt{\_hgt\_dem\_adj} * \texttt{GRAVITY\_MSS},\\ &\texttt{\_SKE\_dem} = 0.5 * \texttt{\_TAS\_dem\_adj} * \texttt{\_TAS\_dem\_adj},\\ &\texttt{\_STE\_dem} = \texttt{\_SPE\_dem} + \texttt{\_SKE\_dem},\\ &\texttt{\_SEB\_dem} = \texttt{\_SPE\_dem} - \texttt{\_SKE\_dem}.\end{aligned} \tag{3}$$

Algorithms 1 and 2 illustrate the ArduPilot TECS pseudocode. The complete TECS code can be found in `AP_TECS.cpp` in the ArduPilot code. The main functions to obtain the throttle and pitch demand are `_update_throttle_with_airspeed()` and `_update_pitch()`. Let us mention that the TECS loops are not "pure" PID loops: ad-hoc modifications have been implemented by the ArduPilot developers for improved performance, such as integrator anti wind-up and output constraints. For better readability, such modifications are mentioned in Algorithms 1 and 2 without the full details.

Algorithm 1: ArduPilot architecture for throttle demand in TECS

```
Input: Target altitude _hgt_dem_adj and
       target airspeed _TAS_dem_adj
Output: Throttle demand _throttle_dem
1 Update energies: Given _STE as in (2),
  _STE_demand as in (3) (and their corresponding
  rates), calculate the total energy errors:
      $\texttt{\_STE\_error} = \texttt{\_STE\_demand} - \texttt{\_STE}$,
      $\texttt{STEdot\_error} = \texttt{STEdot\_demand} - \texttt{\_STEdot}$.
2 Feedforward throttle: Calculate
      $\texttt{ff\_throttle} = \texttt{nomThr} + \texttt{STEdot\_demand} * \texttt{Scale}$,
  where nomThr is a nominal throttle at cruising
  speed and Scale is a parameter to compensate the
  drag increase when increasing STEdot_demand.
3 Proportional-Derivative action: Calculate
      $\texttt{\_throttle\_dem} = \texttt{K\_p} * \texttt{\_STE\_error} + \texttt{throttle\_damp} * \texttt{STEdot\_error}$,
  where K_p and throttle_damp are the
  proportional and derivative gains.
4 Constrain throttle demand: Calculate
      $\texttt{\_throttle\_dem} = \texttt{\_throttle\_dem} + \texttt{ff\_throttle}$,
  then constrain _throttle_dem into the interval
  $[\texttt{THRmin}, \texttt{THRmax}]$.
5 Integral action: Calculate the integrator state
      $\texttt{\_integTHR\_state} = \texttt{\_integTHR\_state} + \texttt{\_STE\_error} * \texttt{K\_i} * \texttt{\_DT}$,
  where K_i is the integral gain and _DT is the
  sampling time. Constrain the integrator state
  between appropriate upper and lower limits.
6 Output throttle demand: Calculate
      $\texttt{\_throttle\_dem} = \texttt{\_throttle\_dem} + \texttt{\_integTHR\_state}$,
  and constrain it again between appropriate upper
  and lower limits.
```

B. Attitude Control

The attitude control aims to control roll, pitch and yaw dynamics by acting on the corresponding deflection surfaces: aileron (for roll), elevator (for pitch) and rudder (for yaw). The three loops are composed of a cascaded PID control of angular velocity and angle control. All three loops are affected by the same scaling factor `scaler`, dependent on the air speed and representing the fact that the deflection effect is different at different air speeds.

- The roll controller is a cascaded PID where the inner loop uses a scaled version of the roll rate error

  $$\begin{aligned}\texttt{rate\_error\_r} = (&\texttt{desired\_rate\_r}\\ &- \texttt{achieved\_rate\_r}) * \texttt{scaler},\end{aligned} \tag{4}$$

  to control the aileron deflection angle `delta_a`. Here, `desired_rate_r` is defined by the outer loop, via

  $$\texttt{desired\_rate\_r} = \mathbf{Fr}(\texttt{angle\_err\_r}), \tag{5}$$

  where the function **`Fr`** comprises a scaling and a saturation action of the roll angle error `angle_err_r`.
- The pitch controller is a cascaded PID where the inner loop uses a scaled version of the pitch rate error

  $$\begin{aligned}\texttt{rate\_error\_p} = (&\texttt{desired\_rate\_p}\\ &- \texttt{achieved\_rate\_p}) * \texttt{scaler},\end{aligned} \tag{6}$$

Algorithm 2: ArduPilot architecture for pitch demand in TECS

```
Input: Target altitude _hgt_dem_adj and
       target airspeed _TAS_dem_adj
Output: Pitch demand _pitch_dem
1 Update energies: Given _SEB as in (2),
    _SEB_demand as in (3) (and their corresponding
    rates), calculate the energy balance errors:
      $\texttt{\_SEB\_error} = \texttt{\_SEB\_demand} - \texttt{\_SEB},$
      $\texttt{SEBdot\_error} = \texttt{SEBdot\_demand} - \texttt{\_SEBdot}.$
2 Proportional-Derivative action: Calculate
      $\texttt{temp} = \texttt{\_SEB\_error} + 0.5 * \texttt{SEBdot\_demand} + \texttt{SEBdot\_error} * \texttt{pitch\_damp},$
    where pitch_damp is the derivative gain.
3 Integral action: Calculate the integrator state
      $\texttt{\_integSEB\_state} = \texttt{\_integSEB\_state} + \texttt{\_SEB\_error} * \texttt{K\_I} * \texttt{\_DT},$
    where K_I is the integral gain and _DT the
    sampling time. Then, constrain
    _integSEB_state so that the total PID action is
    within the saturation range of the pitch demand.
4 Output pitch demand: The final pitch demand is
      $\texttt{\_pitch\_dem} = (\texttt{temp} + \texttt{\_integSEB\_state})/\texttt{gainInv},$
    where gainInv is a flare gain. Then constrain
    _pitch_dem in the operating range of the pitch.
```

to control the elevator angle delta_e. Here, desired_rate_p is defined by the outer loop, via

$$\texttt{desired\_rate\_p} = \mathbf{Fp}(\texttt{angle\_err\_p}) + \texttt{rate\_offset\_p}, \quad (7)$$

where the function **Fp** comprises a scaling and a saturation action of the pitch angle error angle_err_p, while rate_offset_p is a roll compensation term.

- The yaw controller is a cascaded PID where the inner loop uses the lateral acceleration accel_y to control the rudder angle delta_r, and the outer loop defines a desired lateral acceleration by compensating the yaw angle rate omega_z with a turn coordination calculation, and by applying a high-pass filter.

Algorithm 3 illustrates the ArduPilot pseudocode for pitch control (roll control is similar and not repeated). Algorithm 4 illustrates the ArduPilot pseudocode for yaw control. The complete attitude control code can be found in AP_RollController.cpp, (roll), AP_PitchController.cpp (pitch) and AP_YawController.cpp (yaw), in the ArduPilot code. Again, ad-hoc modifications such as integrator anti wind-up and output constraints are mentioned in Algorithms

Algorithm 3: ArduPilot architecture for pitch control

```
Input: Pitch demand _pitch_dem from TECS
Output: Elevator deflection angle delta_e
1 Update pitch data error: Calculate the desired pitch
    rate desired_rate_p from the pitch angle error
    angle_err_p as in (7). Then calculate the pitch
    rate error rate_error_p as in (6).
2 Integral action: Calculate:
      $\texttt{I\_p} = \texttt{I\_p} + \texttt{rate\_error\_p} * \texttt{ki\_p} * \texttt{\_DT},$
    where ki_p is the integral gain and _DT is the
    sampling time. Then, apply integrator anti wind-up
    when the elevator deflection angle delta_e
    reaches its upper or lower limits.
3 Proportional-Derivative action: Calculate:
      $\texttt{P\_p} = \texttt{desired\_rate\_p} * \texttt{kp\_p},$
      $\texttt{FF\_p} = \texttt{desired\_rate\_p} * \texttt{kff\_p},$
      $\texttt{D\_p} = \texttt{rate\_error\_p} * \texttt{kd\_p},$
    where kp_p, kff_p and kd_p are proportional,
    feedforward and derivative gains, respectively.
4 Elevator deflection: Calculate:
      $\texttt{delta\_e} = \texttt{P\_p} + \texttt{I\_p} + \texttt{D\_p} + \texttt{FF\_p},$
    then constrain delta_e within the operating range
    of the elevator deflection angle.
```

3 and 4 without the full details. To account for couplings between roll, pitch and yaw during turning maneuvers, some compensation terms are designed in terms of feedforward action in the desired pitch angle (called roll compensation) in the yaw rate (called turn coordination).

IV. Proposed Approach

We are now in the position to show the integration of the original ArduPilot architecture with adaptation features. The integration relies on four steps (cf. Algorithm 5):

- Select the essential PID terms of each control loop and collect such terms in a vector xi;
- Define, for each loop, a sliding variable s based on the aforementioned PID terms;
- Design the control law by augmenting the original PID action with an adaptive-robust action;
- Design the adaptive law for the adaptive-robust term.

Let e represent an error variable of interest in the UAV (e.g. total energy error, energy balance error, roll, pitch or yaw rate error). The first step is to collect the PID terms in

$$\texttt{xi} = [\texttt{K\_p} * \texttt{e},\ \texttt{K\_d} * \texttt{e\_dot},\ \texttt{K\_i} * \texttt{e\_int}]^T,$$

where e_int denotes the integral error. The sliding variable s can be defined as

$$\texttt{s}(t) = \texttt{K\_d} * \texttt{e\_dot}(t) + \texttt{K\_p} * \texttt{e}(t) + \texttt{K\_i} * \texttt{e\_int}(t). \quad (8)$$

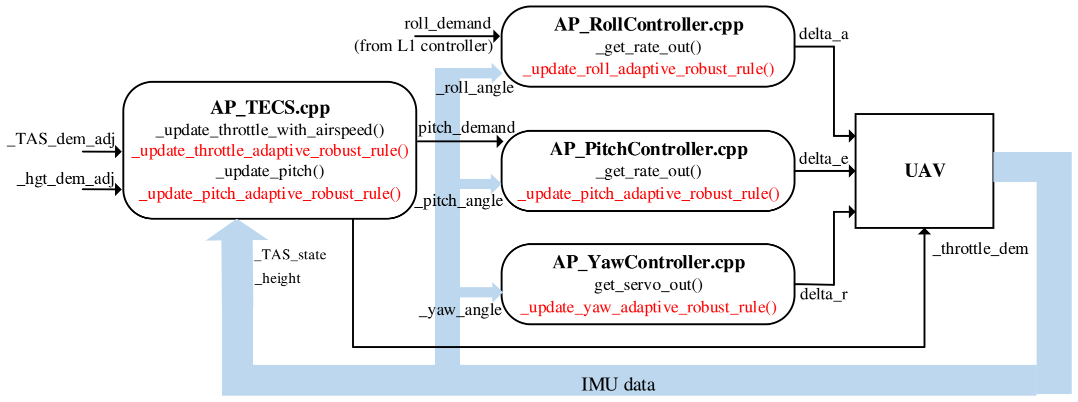


Fig. 1. Control architecture in ArduPilot, with proposed modules reported in red color: TECS and attitude (roll, pitch and yaw) control receive feedback from the UAV via the Inertial Measurement Unit (IMU). TECS and roll controller receive commands (demanded air speed, demanded height and demanded roll) from the higher navigation layers. TECS provides the demanded pitch to the pitch controller and the demanded throttle to the throttle servo motor. The attitude controllers provide the demanded deflection angles to the corresponding servo motors.

```
Algorithm 4: ArduPilot architecture for yaw control
  Input: Acceleration accel_y and bank angle
         bank_angle from inertial sensors
  Output: Rudder deflection angle delta_r
1 Compensation and filtering: Calculate the turn
    coordination term rate_offset_y from
    bank_angle, then calculate the high-pass filter
    input
        $\mathtt{rate\_hp\_in} = \mathbf{ToDeg}(\mathtt{omega\_z} - \mathtt{rate\_offset\_y}),$
    where ToDeg(·) converts radians to degrees. Then,
    pass rate_hp_in through the high-pass filter
    function to obtain rate_hp_out.
2 Integral action: Calculate the integrator input
        $\mathtt{integ\_in} = -\mathtt{\_K\_I} * (\mathtt{\_K\_A} * \mathtt{accel\_y} + \mathtt{rate\_hp\_out}),$
    where _K_I is the integral gain and _K_A the
    acceleration gain. Then, calculate the integrator state
        $\mathtt{\_integrator} = \mathtt{\_integrator} + \mathtt{integ\_in} * \mathtt{delta\_time},$
    where delta_time is the sampling time. Then,
    apply integrator anti wind-up when the rudder
    deflection angle delta_r reaches its upper or
    lower limits.
3 Rudder deflection: Calculate
        $\mathtt{I\_y} = \mathtt{\_K\_D} * \mathtt{\_integrator} * \mathtt{scaler}^2,$
        $\mathtt{D\_y} = \mathtt{\_K\_D} * (\mathtt{-rate\_hp\_out}) * \mathtt{scaler}^2,$
        $\mathtt{\_last\_out} = \mathtt{I\_y} + \mathtt{D\_y},$
        $\mathtt{delta\_r} = \mathtt{\_last\_out},$
    where _K_D is the derivative gain. Then, constrain
    delta_r in the operating range of the rudder.
```

Accordingly, the augmented control action can be written as

$$\mathtt{u\_aug}(t) = \underbrace{\overbrace{\mathtt{s}(t)+\mathtt{ff}(t)}^{\text{PID action}} + \overbrace{\rho(t)}^{\text{Adaptive gain}} * \overbrace{\mathtt{sat}(\mathtt{s}(t)/\mathtt{eps})}^{\text{Robust action}}}_{\text{Proposed augmentation}}, \quad (9)$$

where we have highlighted the different terms of the control law. Here, ff represents the feedforward term not included in s; eps > 0 is a design constant for the saturation function sat$(\cdot)$, and ρ is an adaptive gain designed as

$$\rho(t) = \hat{\kappa}_0(t) + \hat{\kappa}_1(t) * ||\mathtt{xi}(t)|| + \hat{\kappa}_2(t) * ||\mathtt{xi}(t)||^2, \quad (10)$$

$$\dot{\hat{\kappa}}_i(t) = |\mathtt{s}(t)| * ||\mathtt{xi}(t)||^i - \mathtt{alpha} * \hat{\kappa}_i(t), \quad (11)$$

with $\hat{\kappa}_i(0) > 0, \mathtt{alpha} > 0, i = 0, 1, 2$. The stability is based on adaptive sliding mode theory for the dynamics

$$\dot{\mathtt{s}}(t) + k\mathtt{s}(t) = -\mathtt{u\_aug}(t) + \delta(t), \quad (12)$$

$$||\delta(t)|| \leq \kappa_0^* + \kappa_1^* ||\mathtt{xi}(t)|| + \kappa_2^* ||\mathtt{xi}(t)||^2, \quad (13)$$

where δ is an aggregate uncertainty with state dependent upper bound. Then, $\hat{\kappa}_i$ is interpreted as an estimate of κ_i^*. Due to space limits, we refer to the previous work of the authors [15] for stability analysis. The adaptive-robust action in (9) does not modify the original architecture. The users can use the autopilot as usual and tune few additional gains alpha and eps. The approach is easily encapsulated as few extra lines to the original code (refer to the overall control architecture sketched in Fig. 1, or to our released code https://github.com/Friend-Peng/Adaptive-ArduPilot-Autopilot).

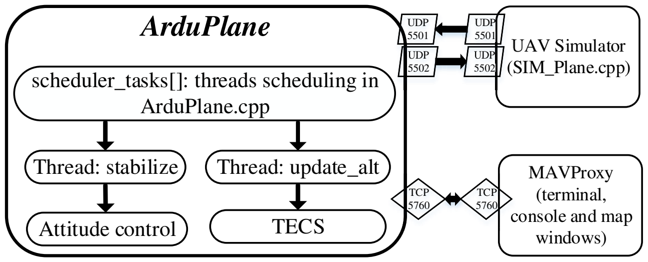
Remark 1: A few modifications in the proposed adaptive ArduPilot are needed to align it with the ad-hoc modifications of the original ArduPilot. Most notably, to align the integral action of the adaptive law (11) with the anti wind-up action of ArduPilot, we freeze the integration of κ_i whenever ArduPilot does so for the integrator state.

Remark 2: The proposed adaptive approach applies also to PX4. In fact, the PX4 function control_bodyrate() in ecl_roll_controller.cpp (roll), ecl_pitch_controller.cpp (pitch) and

Algorithm 5: Integration of proposed adaptation in ArduPilot

```
Input: Errors e, e_dot, e_int (for each loop)
Output: Augmented control u_aug (for each loop)
1 Select PID terms: For TECS throttle demand:
      xi = [K_p * _STE_error,
            throttle_damp * STEdot_error,
            _integTHR_state].
  For TECS pitch demand:
      xi = [_SEB_error/gainInv,
            SEBdot_error * pitch_damp/gainInv,
            _integSEB_state/gainInv].
  For roll control:
      xi = [desired_rate_r * kp_r,
            rate_error_r * kd_r,
            I_r].
  For pitch control:
      xi = [desired_rate_p * kp_p,
            rate_error_p * kd_p,
            I_p].
  For yaw control:
      xi = [_K_D * _integrator * scaler^2,
            _K_D * (-rate_hp_out) * scaler^2].
2 Sliding surface: For each loop, calculate s as in (8)
  based on the aforementioned PID terms.
3 Adaptive-robust action: For each loop, update the
  adaptive gains in discrete time
      kappa_i = kappa_i + _DT * kappa_i_dot,
  where kappa_i_dot is $\dot{\hat{\kappa}}_i(t)$, $i = 0, 1, 2$ in (11).
  Then, calculate the adaptive gain $\rho(t)$ as in (10).
4 Augmented input: For each loop, the feedfoward
  terms are as follows. For TECS throttle demand:
      ff = ff_throttle.
  For TECS pitch demand:
      ff = 0.5 * SEBdot_demand/gainInv.
  For roll control:  ff = FF_r.
  For pitch control: ff = FF_p.
  For yaw control:   ff = 0.
  Then, calculate u_aug, the sum of PID and
  adaptive-robust action as in (9).
```

ecl_yaw_controller.cpp (yaw) is analogous to the attitude control in ArduPilot. Also, the PX4 functions _update_throttle_setpoint() and _update_pitch_setpoint() in TECS.cpp are analogous to the total energy control system in Ardupilot.



ArduPlane
scheduler_tasks[]: threads scheduling in ArduPlane.cpp
Thread: stabilize
Thread: update_alt
Attitude control
TECS
UDP 5501
UDP 5502
UDP 5501
UDP 5502
UAV Simulator (SIM_Plane.cpp)
TCP 5760
TCP 5760
MAVProxy (terminal, console and map windows)

Fig. 2. ArduPilot SITL set up: the SITL environment includes ArduPlane, the flight simulator and MAVProxy. ArduPlane communicates with simulator by UDP protocol (port 5501 and 5502), and by TCP protocol (port 5760) with MAVProxy. ArduPlane has a scheduler table scheduler_tasks[] listing all tasks. For compactness, we only show attitude control and TECS threads: other threads, e.g. for navigation, are not shown.

V. Software-In-The-Loop Validation

The performances of the original (non-adaptive) and the proposed (adaptive) ArduPilot are compared for a full flight scenario consisting of a take-off phase, waypoint following and landing. The flight scenario is implemented in the software-in-the-loop (SITL) environment of ArduPilot [35]. The ArduPilot SITL set up is illustrated in Fig. 2. The SITL environment allows to interface ArduPlane (responsible for autopilot and simulator of UAV dynamics) with MAVProxy (responsible for ground control station, GUI and command console). With this interfacing step, ArduPlane can receive the flight mode commands from MAVProxy and send flight data to MAVProxy. ArduPlane version 4.0.6 was used. MAVProxy is a popular open-source Ground Control Station (GCS) suite, where the user can set the flight mode via command console and set waypoints. During the flight tests, the UAV takes off and follows an orbit at constant altitude. We use the commands mode takeoff, arm throttle and mode rtl (or mode circle) to make the fixed-wing UAV take off and fly around the orbit. The original ArduPilot code is used for comparison, with the original PID gains as set in ArduPilot code [2]. The parameters for the proposed augmented ArduPilot are listed in Table II.

To evaluate the robustness of the autopilot to uncertainty, we simulate a change in the mass of the UAV in the simulator: accordingly, the ArduPlane simulator will change the inertia corresponding to this mass change. The autopilot is not aware of the mass change in advance, so that it should compensate for the resulting alteration effects (e.g. alteration in altitude and airspeed due to the mass change). The initial mass of the UAV is 2kg, and the mass can drop to 1kg (this can represent the UAV dropping or losing the load),

TABLE II
Parameter selection for the proposed adaptation

Loop	$\hat{\kappa}_i(0)$	alpha	eps
pitch	10^{-5}	10	420
roll	10^{-5}	150	1000
yaw	10^{-5}	0.001	0.001
TECS throttle	10^{-5}	0.001	10
TECS pitch	10^{-5}	4	45

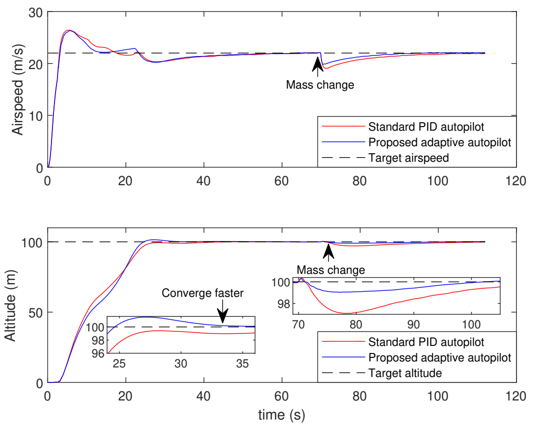


Fig. 3. Mass change 2.0kg → 1.0kg while orbiting: comparison of airspeed and altitude for original and proposed ArduPilot. The proposed solution not only enables the UAV to reach faster the target altitude and target airspeed, but it also exhibits much smaller altitude drop with faster recovery after mass change.

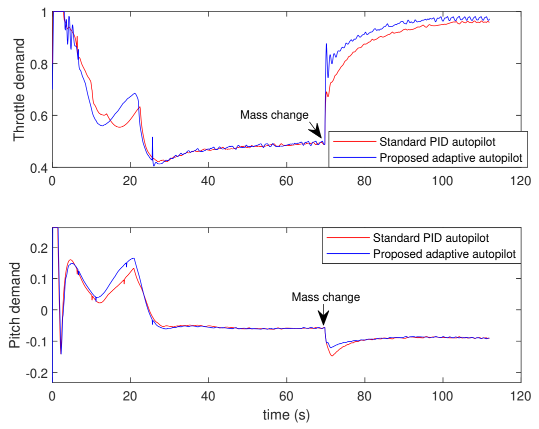


Fig. 5. Mass change 2.0kg → 1.0kg while orbiting: comparison of throttle demand and pitch demand for original and proposed ArduPilot. It can be noticed that the proposed solution responds faster to the mass change in terms of throttle demand, which in turns requires less pitch variation. Fig. 3 shows that this results in a much smaller altitude drop, with faster recovery.

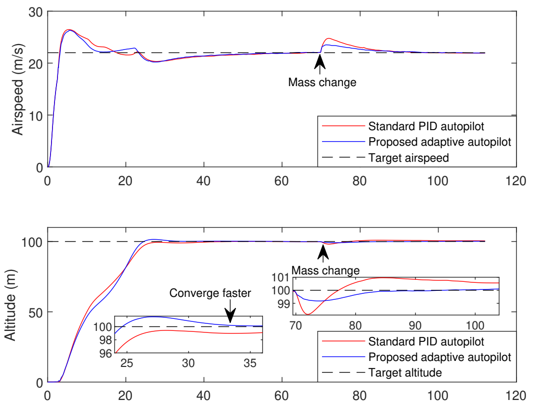


Fig. 4. Mass change 2.0kg → 4.0kg while orbiting: comparison of airspeed and altitude for original and proposed ArduPilot. The proposed solution not only enables the UAV to reach faster the target altitude and target airspeed, but it also exhibits much smaller altitude drop with faster recovery after mass change.

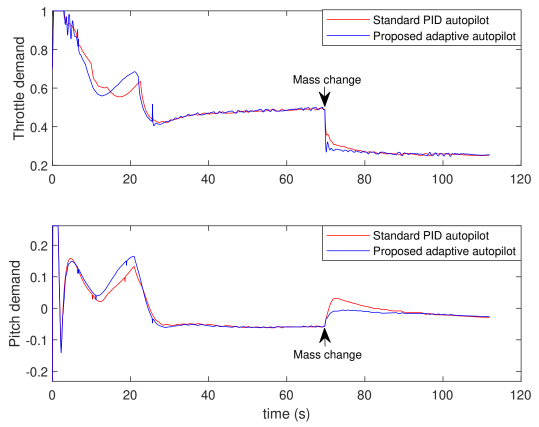


Fig. 6. Mass change 2.0kg → 4.0kg while orbiting: comparison of throttle demand and pitch demand for original and proposed ArduPilot. It can be noticed that the proposed solution responds faster to the mass change in terms of throttle demand, which in turns requires less pitch variation. Fig. 4 shows that this results in a much smaller altitude drop, with faster recovery.

or increase to 4kg (this can represent the UAV catching an additional load via some net or clamping system).

To evaluate the performance, the tracking error cost is calculated for the 5 ArduPilot loops. The cost of the original ArduPilot (without mass change) is used as a normalizing factor, i.e. its cost is 5.0 (due to 5 loops). The tracking error costs are shown in Table III: the table shows that the cost of the original ArduPilot increases to 17 and 21 (more than 3 and 4 times the baseline cost) due to mass change, showing that the mass change scenario is critical for performance degradation. The tracking error costs of the proposed adaptive ArduPilot are consistently better than the original ArduPilot: the proposed adaptive method improves performance up to 20.8% without mass change and more than 35.5% in the presence of mass change. Most importantly, the inevitable performance degradation due to mass change is less than 3 and 4 times the baseline cost, i.e. the performance degradation is smaller in the proposed ArduPilot as compared to the original one. This shows improved robustness of the proposed method.

The experiments with half mass change and double mass

TABLE III

TRACKING ERROR COSTS FOR ORIGINAL AND PROPOSED AUTOPILOT

Mass	Original (non-adaptive) autopilot						
	Roll	Pitch	Yaw	TECS throttle	TECS pitch	Total	
$2 \rightarrow 1$kg	1.38	0.88	5.68	10.33	3.09	21.36	
2kg	1	1	1	1	1	5.0	
$2 \rightarrow 4$kg	0.72	1.3	4.0	5.43	5.9	17.35	
Mass	Proposed (adaptive) ArduPilot						
	Roll	Pitch	Yaw	TECS throttle	TECS pitch	Total	Improv.%
$2 \rightarrow 1$kg	1.47	0.94	5.44	4.94	2.97	15.76	(**35.5%**)
2kg	1	1	0.49	0.73	0.92	4.14	(**20.8%**)
$2 \rightarrow 4$kg	0.78	1.17	3.6	2.55	3.9	12.0	(**44.6%**)

change are visualized in Figs. 3~6 for the original and the proposed adaptive ArduPilot. As shown in Figs. 3 and 4, the proposed adaptive ArduPilot has negligible altitude drop after mass change as compared to the original one. Meanwhile, Figs. 5 and 6 report the TECS throttle and pitch demand, i.e. the set points passed to the low-level control loops: the throttle demand for the proposed adaptive ArduPilot is more reactive during the mass change, which in turns require a smaller pitch variation.

VI. Conclusions

This work has shown a new adaptive ArduPilot that embeds the original open-source ArduPilot architecture in a seamless way with adaptive mechanisms. Comparative experiments with the original ArduPilot have shown consistent improved performance in the presence of changing flight dynamics. Interesting future work is to cover all modules of ArduPilot (ArduPlane, ArduCopter, ArduRover) and other open-source autopilots such as PX4.

References

[1] J. A. Marshall, W. Sun, and A. L'Afflitto, "A survey of guidance, navigation, and control systems for autonomous multi-rotor small unmanned aerial systems," *Annual Reviews in Control*, vol. 52, pp. 390–427, 2021.

[2] "Open source for ardupilot open source autopilot," Online, https://github.com/ArduPilot/ardupilot.

[3] "Open source for drones-px4 open source autopilot," Online, https://px4.io/.

[4] "OpenPilot," Online, http://www.ehirobo.com/openpilot.

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

[6] D. Invernizzi, M. Lovera, and L. Zaccarian, "Dynamic attitude planning for trajectory tracking in thrust-vectoring UAVs," *IEEE Transactions on Automatic Control*, vol. 65, no. 1, pp. 453–460, 2020.

[7] A. Bosso, C. Conficoni, D. Raggini, and A. Tilli, "A computational-effective field-oriented control strategy for accurate and efficient electric propulsion of unmanned aerial vehicles," *IEEE/ASME Transactions on Mechatronics*, pp. 1–1, 2020.

[8] N. Hovakimyan and C. Cao, *L1 Adaptive Control Theory: Guaranteed Robustness with Fast Adaptation*. SIAM, Advances in Design and Control, 2010.

[9] E. Lavretsky and K. Wise, *Robust and Adaptive Control: With Aerospace Applications*. Springer, Advanced Textbooks in Control and Signal Processing, 2013.

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

[11] A. Koubâa, A. Allouch, M. Alajlan, Y. Javed, A. Belghith, and M. Khalgui, "Micro air vehicle link (MAVlink) in a nutshell: A survey," *IEEE Access*, vol. 7, pp. 87 658–87 680, 2019.

[12] J. Yang, X. Wang, S. Baldi, S. Singh, and S. Farì, "A software-in-the-loop implementation of adaptive formation control for fixed-wing UAVs," *IEEE/CAA Journal of Automatica Sinica*, vol. 6, no. 5, pp. 1230–1239, 2019.

[13] J. Barreiro-Gomez, I. Mas, J. Giribet, P. Moreno, C. Ocampo-Martinez, R. Sánchez-Peña, and N. Quijano, "Distributed data-driven UAV formation control via evolutionary games: Experimental results," *Journal of the Franklin Institute*, vol. 358, no. 10, pp. 5334–5352, 2021.

[14] S. Roy, S. Baldi, and L. M. Fridman, "On adaptive sliding mode control without a priori bounded uncertainty," *Automatica*, vol. 111, p. 108650, 2020.

[15] P. Li, D. Liu, and S. Baldi, "Adaptive integral sliding mode control in the presence of state-dependent uncertainty," *IEEE/ASME Transactions on Mechatronics*, pp. 1–11, 2022.

[16] R. Chiappinelli, M. Cohen, M. Doff-Sotta, M. Nahon, J. R. Forbes, and J. Apkarian, "Modeling and control of a passively-coupled tilt-rotor vertical takeoff and landing aircraft," in *2019 International Conference on Robotics and Automation (ICRA)*, 2019, pp. 4141–4147.

[17] S. Sabet, M. Singh, M. Poursina, and P. E. Nikravesh, "A highly maneuverable hybrid energy-efficient rolling/flying system," in *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2021, pp. 2485–2490.

[18] A. Erasmus and H. Jordaan, "Robust adaptive control of a multirotor with an unknown suspended payload," *IFAC-PapersOnLine*, vol. 53, no. 2, pp. 9432–9439, 2020, 21st IFAC World Congress.

[19] K. Amelin, S. Tomashevich, and B. Andrievsky, "Recursive identification of motion model parameters for ultralight UAV," *IFAC-PapersOnLine*, vol. 48, no. 11, pp. 233–237, 2015, 1st IFAC Conference on Modelling, Identification and Control of Nonlinear Systems MICNON 2015.

[20] B. Zhou, H. Satyavada, and S. Baldi, "Adaptive path following for unmanned aerial vehicles in time-varying unknown wind environments," in *2017 American Control Conference (ACC)*, 2017, pp. 1127–1132.

[21] S. I. Azid, K. Kumar, M. Cirrincione, and A. Fagiolini, "Wind gust estimation for precise quasi-hovering control of quadrotor aircraft," *Control Engineering Practice*, vol. 116, p. 104930, 2021.

[22] S. Farí, X. Wang, S. Roy, and S. Baldi, "Addressing unmodeled path-following dynamics via adaptive vector field: A UAV test case," *IEEE Transactions on Aerospace and Electronic Systems*, vol. 56, no. 2, pp. 1613–1622, 2020.

[23] A. L. Fradkov, S. Tomashevich, B. Andrievsky, K. Amelin, and I. N. Kaliteevskiy, "Adaptive coding for data exchange between quadrotors in the formation," *IFAC-PapersOnLine*, vol. 49, no. 13, pp. 275–280, 2016, 12th IFAC Workshop on Adaptation and Learning in Control and Signal Processing ALCOSP 2016.

[24] S. Baldi, D. Sun, G. Zhou, and D. Liu, "Adaptation to unknown leader velocity in vector-field UAV formation," *IEEE Transactions on Aerospace and Electronic Systems*, vol. 58, no. 1, pp. 473–484, 2022.

[25] S. Panza, D. Invernizzi, M. Giurato, and M. Lovera, "Design and characterization of the 2DoF drone: a multirotor platform for education and research," *IFAC-PapersOnLine*, vol. 54, no. 12, pp. 32–37, 2021, iFAC Workshop on Aerospace Control Education WACE 2021.

[26] S. Wang, X. Dai, C. Ke, and Q. Quan, "RflySim: A rapid multicopter development platform for education and research based on Pixhawk and MATLAB," in *2021 International Conference on Unmanned Aircraft Systems (ICUAS)*, 2021, pp. 1587–1594.

[27] D. Reinhardt and T. A. Johansen, "Control of fixed-wing UAV attitude and speed based on embedded nonlinear model predictive control," *IFAC-PapersOnLine*, vol. 54, no. 6, pp. 91–98, 2021, 7th IFAC Conference on Nonlinear Model Predictive Control NMPC 2021.

[28] S. Baldi, S. Roy, K. Yang, and D. Liu, "An underactuated control system design for adaptive autopilot of fixed-wing drones," *IEEE/ASME Transactions on Mechatronics*, pp. 1–12, 2022.

[29] L. He, N. Aouf, and B. Song, "Explainable deep reinforcement learning for UAV autonomous path planning," *Aerospace Science and Technology*, vol. 118, p. 107052, 2021.

[30] E. A. Niit and W. J. Smit, "Integration of model reference adaptive control (MRAC) with PX4 firmware for quadcopters," in *2017 24th International Conference on Mechatronics and Machine Vision in Practice (M2VIP)*, 2017, pp. 1–6.

[31] A. Moeini, M. A. Rafique, Z. Xue, A. F. Lynch, and Q. Zhao, "Disturbance observer-based integral backstepping control for UAVs," in *2020 International Conference on Unmanned Aircraft Systems (ICUAS)*, 2020, pp. 382–388.

[32] B. Pugach, B. Beallo, D. Bement, S. McGough, N. Miller, J. Morgan, L. Rodriguez, K. Winterer, T. Sherman, S. Bhandari, and Z. Aliyazicioglu, "Nonlinear controller for a UAV using echo state network," in *2017 International Conference on Unmanned Aircraft Systems (ICUAS)*, 2017, pp. 124–132.

[33] W. Saengphet, S. Tantrairatn, C. Thumtae, and J. Srisertpol, "Implementation of system identification and flight control system for UAV," in *2017 3rd International Conference on Control, Automation and Robotics (ICCAR)*, 2017, pp. 678–683.

[34] A. Goel, J. A. Paredes, H. Dadhaniya, S. A. Ul Islam, A. M. Salim, S. Ravela, and D. Bernstein, "Experimental implementation of an adaptive digital autopilot," in *2021 American Control Conference (ACC)*, 2021, pp. 3737–3742.

[35] "SITL Simulator," Online, https://ardupilot.org/dev/docs/sitl-simulator-software-in-the-loop.html.