LINCOLN LABORATORY
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Unitair
Unified Guidance, Navigation, and Control Implementation for Unmanned Aerial Vehicles

2021 Report

Jason Nezvadovitz   Navid Shahrestani

October 29th 2021

Contents

1 Introduction

Unmanned aerial vehicles (UAVs) have a prominent role in a wide variety of programs at MIT Lincoln Laboratory (MITLL). They continue to serve as platforms for sensor development, testbeds for artificial intelligence (AI) algorithms, core components of intelligence, surveillance, and reconnaissance (ISR) capabilities, and more.

Typically, such programs desire a reliable commercial-off-the-shelf (COTS) vehicle that can autonomously fly prescribed paths. The low-level guidance, navigation, and control (GNC) "autopilot" software enabling that basic autonomy is not supposed to be the focus of the program, but rather a "solved problem" that simply enables the main novel development, be it the payload or higher-level algorithms.

Unfortunately, in reality, a large portion of initial program effort is usually spent on autopilot development. Specifically, time is spent adapting and consequently debugging an open-source autopilot like PX4 or ArduPilot. The primary purpose of this Technology Initiative (TI) is to forgo tacking-on autopilot development to larger programs, and instead direct a dedicated effort to the development of a unified, high-performance but easy-to-use autopilot for lab-wide support.

We are targeting the following design goals for this autopilot, dubbed "Unitair."

1. **Minimal moving parts:** All processing is performed on a single COTS computer using a single middleware environment

2. **One size fits all:** Fixed-wing and multirotor UAVs of arbitrary geometries are handled by a single GNC architecture that can integrate easily with higher-level or mission-specific autonomy software

3. **State of the art:** The selected algorithms provide the foremost of modern GNC

4. **Interpretable:** The code clearly reflects the algorithm architecture with a well-documented, consistent, and lean object-oriented design

1.1 Existing Systems

The most common autopilot in use at MITLL is a modified version of PX4, a long-standing, open-source software stack built collectively by the hobby-drone community. However, its maturity is to some extent its downfall. With over 500,000 lines of code (excluding comments) from over 100 different contributors, it is a massive and difficult to parse code-base, which can quickly become a "house of cards" situation with a challenging learning curve. This sentiment is shared well beyond just us; here is an excerpt from the documentation of a more recent open-source autopilot project ROSflight:

> *"The other [autopilot] options that we tried were limited in bandwidth for streaming sensor data, and the APIs for sending control setpoints were confusing and difficult to implement. Perhaps most importantly, the code was sometimes so complex (feature-rich, but complicated) that it was difficult to figure out what the autopilot was actually doing. In talking to other researchers and industry members, we found that many people shared similar frustrations." – ROSflight Team*

We believe this quote to be tacitly referencing PX4 and its fork ArduPilot. ROSflight is aiming to solve the interpretability problem of PX4 by starting from scratch and remaining lean. However, their architecture still targets using a microprocessor separate from the main UAV computer, which we believe unnecessarily complicates UAVs. Moreover, their current software doesn't meet our "one size fits all" and "state of the art" goals.

The vast majority of other open-source autopilots (with popularity extending beyond a single university lab) are intended for the hobby remote-controlled (RC) drone community. They only provide basic attitude stabilization for ease of human-piloting, and are not intended or prepared to be the foundation of any cutting-edge autonomous system.

1.2 Technical Overview

Unitair can be succinctly described by the block diagram in Figure 1.1.

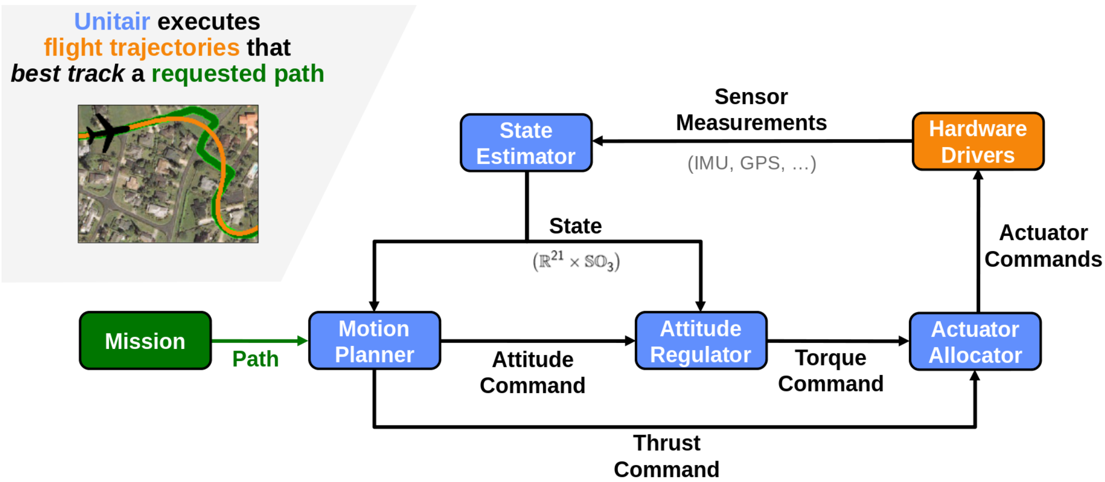

Figure 1.1: The Unitair top-level block-diagram.

The typical use-case is that the "mission" (either a commanding human or higher-level autonomy code) requests Unitair to have the UAV track an arbitrarily long time-series of positions called a path. For Unitair, a path is an expressive request because tracking is handled as a vehicle-aware constrained optimization. E.g., for a multirotor, a path with just one position will result in a hover at that position, while for a conventional fixed-wing, it will result in a minimum-radius orbit (often called a "loiter") about that position. These emergent behaviors are described more in Section 3.1.

Each blue block represents a single asynchronous executable running a certain algorithm. These are listed below along with some key details.

- **Planner:** Model-predictive controller (MPC) to express path-tracking as optimization
  - Aerodynamic forces are parameterized in a way that is easy to manually tune
  - A constraint on airspeed allows the model to treat angular velocity as a control without worrying about how attitude authority is velocity dependent
  - Differential dynamic-programming (DDP) naturally handles the state manifold
- **Regulator:** Adaptive feedback controller to track the planned angular velocities
  - Typical PI plus a Lyapunov-theoretic adaptive term to ease tuning
  - Designed to account for airspeed without extensive vehicle profiling
- **Allocator:** Sequential quadratic-programming (SQP) to map a wrench onto actuators
  - Incorporates the PX4-style fixed-wing "mixer" into the more principled framework typically employed by multirotors and spacecraft
- **Estimator:** Extended Kalman filter (EKF) to fuse sensors into a belief state
  - Same Lie-theoretic formulation used in military-grade navigation systems
  - Platform agnostic by treating the inertial measurement unit (IMU) as process
  - Highly efficient leverage of Jacobian sparsity

Finally, the orange block represents a collection of asynchronous drivers reading from sensors and signaling actuators (detailed in Chapter 2). All software is written in C++ with the exception of some UI tools written in Python. Unlike the other autopilots we have researched, Unitair has *every* block in the diagram running on a single-board computer (SBC), thus cutting out the need for handling a microprocessor.

A microprocessor separate from the main "mission computer" is especially cumbersome when the user wishes to access or interface with any of the lower-level algorithms directly. For example, integrating vision-based navigation with PX4's estimator firmware has required some surprisingly challenging plumbing on a number of MITLL programs. Forgoing the microprocessor allows for simpler runtime introspection, runtime reconfiguration, physical packaging, and software management. Part of this TI's novelty is in demonstrating that a single modern computer's bandwidth is sufficient for most UAVs.

2 Hardware

The Unitair hardware architecture (shown in Figure 2.1) strives to simplify the canonical mixed microcontroller/SBC paradigm by eliminating the microcontroller entirely and migrating that functionality to the SBC. As with any engineering decision, there are trade-offs that must be weighed, and in this case we chose to trade-off the deterministic real-time performance guarantees provided by a microcontroller in an effort to create a system that is easy for developers of all backgrounds to interface, modify, and extend.

As detailed in Section 2.2, hardware benchmarking of our system shows that a Linux-based OS running ROS (patched with the PREMPT-RT patch) is capable of meeting the timing required for reliable UAV flight. Future migration to ROS 2.0 could potentially allow the same easy-to-use system to also provide some level of real-time software guarantees [1] [2]. Thus, forgoing the microcontroller is a well justified trade-off.

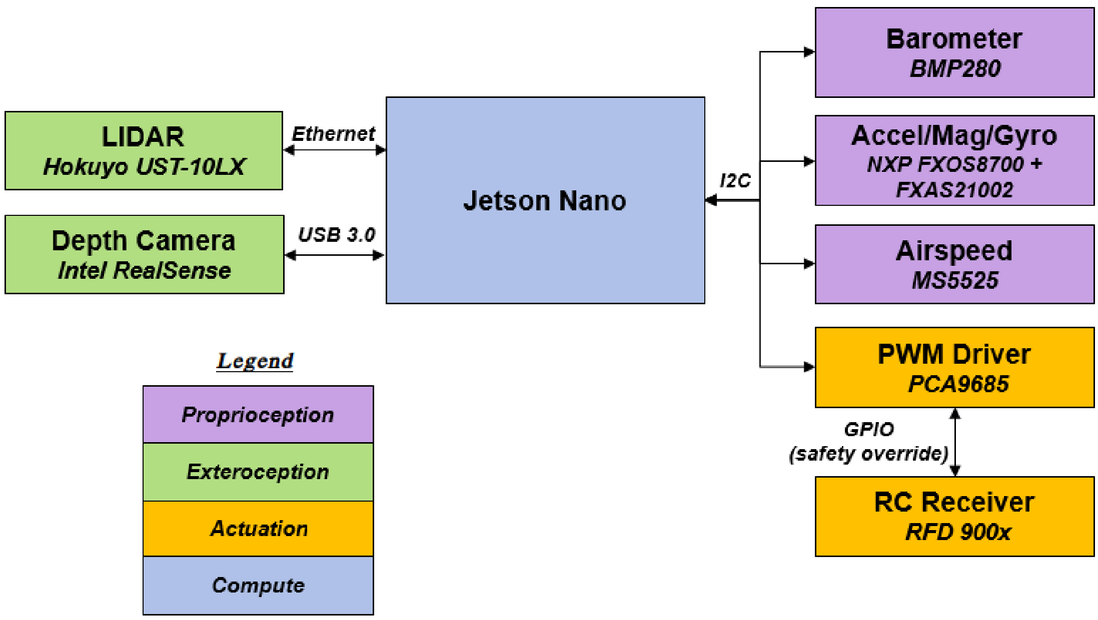

Figure 2.1: The Unitair hardware block-diagram. Part numbers are provided as examples for the general device suite that a Unitair UAV would employ.

2.1 Device Selection

For this project we chose to leverage the immense growth of commercial UAVs in the last decade, which has created a market for cheap and readily available sensors and processors [3]. Below is a brief description of each component:

- **Exteroception sensors**: Acquire information about the environment
  - Depth Camera (Intel Realsense)
    - Active IR stereo camera for simultaneous localization and mapping
  - LIDAR (Hokuyo UST-10LX)
    - Small-form factor 2D LIDAR for simultaneous localization and mapping
- **Proprioceptive sensors**: Measure values internal to the system
  - 9DOF IMU (NXP FXOS8200 + FXAS21002)
    - Measures linear acceleration, rotational velocity, and magnetic field to determine the orientation of a vehicle
    - Accuracy of this device heavily impacts overall system performance, therefore a survey was performed (shown in Table 2.1) to evaluate the "zero-rate-level" (the deviation of output signal from the ideal while stationary) across common commercial gyroscopes
  - Barometer (BMP280)
    - Measures atmospheric pressure to determine altitude.
  - Airspeed (MS5525)
    - Pressure transducer that measures differential pressure within a pitot tube (needed for outdoor operation where wind can cause significant deviations between airspeed and ground speed).
- **Actuation**: Drives thruster motors and control-surface servos
  - PWM driver (PCA9685)
    - Generates up to 16 control signals, each of which can control an electronic speed controller (ESC) or servo
  - RC receiver (RFD 900x)
    - Provides safety mechanism for user to remotely disable all actuators
- **Compute:** Responsible for running all algorithms
  - SBC (Jetson Nano or NX)
    - Executes the entire GNC stack on a fraction of the CPU, leaving both CPU and GPU open for mission-specific processing

	+/- 250 dps	+/- 500 dps	+/- 2000 dps
L3GD20	10 dps	15 dps	75 dps
FXAS21002C	0.3906 dps	0.78125 dps	3.125 dps
LSM9DS0	10 dps	15 dps	25 dps
LSM9DS1	<= 30 dps	<= 30 dps	<= 30 dps
MPU-9250	5 dps	not provided	not provided
BMI055	1 dps	not provided	not provided

Table 2.1: Gyroscope zero-rate-level comparison in degrees per second.

2.2 Real-Time Performance

Real-time systems are a large focus in the aviation community, and have generated a wealth of documentation to support certification authorities such as the FAA (e.g. DO-178C). The term "real-time" is often conflated with low-latency, or how quickly a system is able to process information. However, real-time computing refers to how deterministic a system is in meeting a specified timing constraint (i.e. deadline) no matter what the actual timing constraint may be.

If a system can guarantee that its outputs will be logically correct and that it will meet the deadlines specified for that computation, then it can be classified as "real-time". Depending on the criticality of meeting that deadline, a system can be further categorized as "hard real-time" if missing a deadline is considered a total system failure (e.g. a rocket engine control unit), "firm real-time" if missing a deadline renders that computation invalid but does not cause a system failure (e.g. a robot assembly line), or "soft real-time" if a missed deadline degrades performance but does not render the computation invalid (e.g. a video delivery service).

A system supporting real-time operation requires both an operating system kernel that allows full preemption and user-space software (libraries and applications) designed to eliminate or mitigate the use of non-deterministic operations. The former was addressed by the Linux operating system's PREEMPT_RT kernel patch, the latter requires careful software engineering effort to employ the following guidelines:

- Memory Management
  - Pre-allocate stack with fixed size objects
  - Dynamic memory allocation done before any real-time code is executed, then locking virtual address space to a fixed size to prevent returning de-allocated memory to the kernel

  - Use of "real-time safe" memory allocators such as Two-Level Segregated Fit
  - Limit operations with poor cache locality (such as with executing inherited functions; which requires the program to access data used in the function, the vtable for the associated class, and the function instructions, all of which may or may not be stored in the cache together)
  - Avoid exception handling to limit the amount of memory that gets pushed to the stack

- Device I/O
  - Keep all disk read/writes and any interaction with physical devices out of "real-time" code sections

- Multi-threaded programming
  - Create all threads at the start of a program to confine thread allocation nondeterminism
  - Avoid priority inversion by utilizing priority inheritance, lock-free data structures, or the complete avoidance of blocking synchronization primitives
  - Avoid the use of forks or spinlocks

Profiling was performed on our flight hardware to measure process latency and jitter across various operating regimes. The sample results shown in Figure 2.2 indicate that the system is capable of meeting timing constraints that are much smaller in magnitude than the control bandwidth required for typical subsonic UAVs. The histogram below was generated by profiling the process loop over the course of 10 minutes and artificially loading the CPU using the Linux "stress" tool.

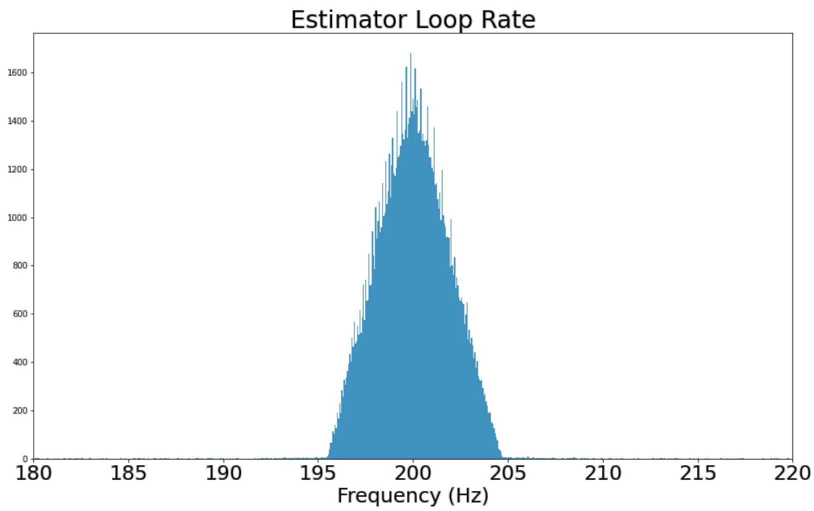

Figure 2.2: Loop-rate histogram for the state estimator (5 ms integration timestep).

3 Software

There is an extensive, ever-evolving wealth of research on GNC algorithms for UAVs. Individually, the algorithms employed by Unitair are not novel, rather tried-and-true. These are: differential dynamic-programming (DDP) [4] [5], Lyapunov-gradient-based adaptive control [7] [8], sequential quadratic-programming (SQP) control allocation [9], and extended Kalman filtering (EKF) [11] [12] [14].

However, there is a degree of novelty in how Unitair is applying these algorithms. Typical UAV papers detail application for either a multirotor *or* a fixed-wing. This is because of a very important distinction between their dynamics: a multirotor's attitude authority is effectively independent of its airspeed. This allows multirotors to cascade a translational control system on top of an independent attitude control system, with the benefit that each subsystem becomes significantly simpler.

For fixed-wings to utilize the same cascade, the current UAV literature requires (sometimes implicitly) modeling or restricting aerodynamic regimes because:

- An attitude controller requires different tuning depending on the airspeed
- A translational motion planner needs to consider how airspeed affects the fulfillment of attitude commands

Modeling aerodynamic regimes requires wind-tunnel tests or CFD, both of which are usually out-of-scope for UAV research programs where the focus is autonomy rather than vehicle design. Therefore, most researchers resign to restricting their fixed-wing to a narrow regime (e.g. "coordinated-turn") and move on to the higher-level tasks.

Unitair's GNC architecture is designed to alleviate this burden for fixed-wings, without sacrificing applicability to multirotors. To accomplish this, we formulate the translational planner and the attitude regulator in somewhat novel ways detailed in Sections 3.1 and 3.2 respectively.

3.1 Motion Planner

The planner is responsible for generating angular velocity and thrust commands that, if fulfilled, will cause the UAV to track the path (a position time-series) specified by the user (a human or higher-level software). This is accomplished via model-predictive control (MPC): a trajectory optimization problem is solved from the current state estimate, and then the first decision of the solution is executed before re-solving the optimization again from the new state estimate.

The dynamics for the trajectory optimization are the usual rigid-body equations [14]:

$$\dot{p} = Rv \tag{3.1}$$

$$\dot{v} = \frac{1}{m}f + R^{\intercal}g - \omega \times v \tag{3.2}$$

$$\dot{R} = R[\omega]_{\times} \tag{3.3}$$

where $p(t) \in \mathbb{R}^3$ is the position of the center-of-mass (COM) expressed in world (inertial) coordinates, $v(t) \in \mathbb{R}^3$ is the velocity of the COM relative to the world expressed in body (co-rotating, COM-centered) coordinates, $R(t) \in \mathbb{SO}3$ is the rotation matrix taking the body coordinate basis to the world coordinate basis, $\omega(t) \in \mathbb{R}^3$ is the angular velocity of the body relative to the world expressed in body coordinates, $g \in \mathbb{R}^3$ is the gravitational field expressed in world coordinates, $m \in \mathbb{R}_+$ is the body's total mass, and $f(t) \in \mathbb{R}^3$ is the net force at the COM expressed in body coordinates.

The net force is generated by the collective thrust $u(t) \in \mathbb{R}$ and aerodynamic forces. A simplified aerodynamic model (visualized in Figure 3.1) is used to keep the whole model tractable and easy to fit while capturing the overall effects needed for aerial maneuvering.

$$f = \begin{bmatrix} u - c_d v_x \\ 0 \\ -c_l v_z \end{bmatrix} \tag{3.4}$$

The "drag coefficient" $c_d \in \mathbb{R}_+$ tunes the UAV's top flight speed, while the "lift coefficient" $c_l \in \mathbb{R}_+$ tunes the UAV's nominal angle-of-attack (AoA). Both of these coefficients should be set to zero for multirotors, which reduces this guidance model to the one used consistently in multirotor MPC literature [6].

Even if c_d and c_l are misspecified, the UAV can still fly owing to the feedback inherent to MPC. E.g., if the UAV requires a larger AoA to maintain altitude than the model predicts, it will descend, and the optimization will re-solve for a new trajectory with a larger AoA. Then, in flight (physical or simulation), c_d and c_l can be tuned.

It is imperative for the planner to understand that, for a fixed-wing, as airspeed $v_a(t) \in \mathbb{R}$ goes to zero, so does its authority over ω. However, we don't want to break the simplifying abstraction that ω is a "control," so instead of telling the planner what happens as airspeed drops, we instead impose a state constraint that airspeed remain above a specified minimum $\underline{v}_a \in \mathbb{R}_+$.

$$v_a = v_x + c_s u \tag{3.5}$$

$$v_a > \underline{v}_a \tag{3.6}$$

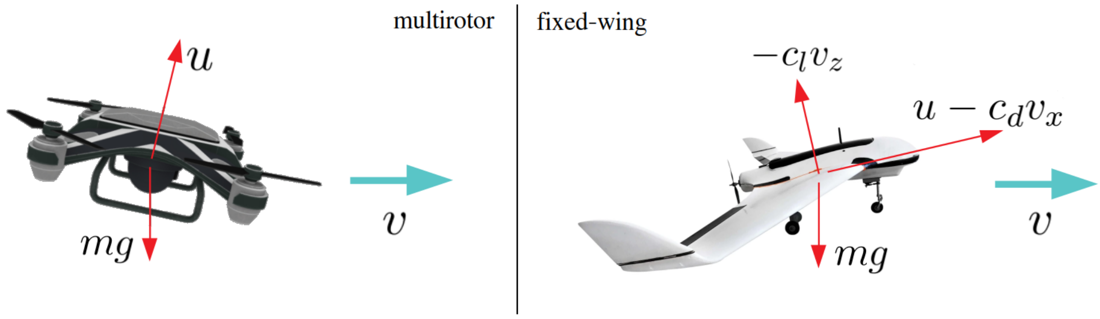

Figure 3.1: Visualization of the planner's aerodynamic model. Both the multirotor and fixed-wing are assumed to have rightward velocity. The multirotor, with $c_d = c_l = 0$, maintains this motion by keeping its collective thrust u mostly vertical to balance gravity, but angled slightly rightward. Meanwhile, the fixed-wing points its collective thrust nearly rightward, with a slightly upward angle so that its wings generate lift $-c_l v_z$ to balance gravity.

The airspeed calculation depends on the collective thrust through proportionality $c_s \in \mathbb{R}_+$ in addition to the forward velocity $v_x(t) \in \mathbb{R}$. This is to account for aerobatic fixed-wings capable of hovering by blowing enough air over their own control-surfaces. Wind can also easily be incorporated if measured.

The objective of the trajectory optimization is to minimize the integrated tracking error between the UAV's position and the desired position $r(t) \in \mathbb{R}^3$, plus regularization terms on orientation and control effort, and penalty terms for the constraints.

$$J^* = \min_{u,\omega} \frac{1}{2t_n} \int_0^{t_n} j \, dt \tag{3.7}$$

$$\begin{aligned}
j \;&= ||p - r||^2_{\Gamma_p} && \text{(position error)} \\
&+ ||Rv - \dot{r}||^2_{\Gamma_v} && \text{(velocity error)} \\
&+ 2\big(\gamma_x(1 - R_x^\intercal \frac{\dot{r}}{||\dot{r}||}) + \gamma_z(1 + R_z^\intercal \frac{g}{||g||})\big) && \text{(alignment regularization)} \\
&+ \gamma_u(u - \underline{u})^2 + ||\omega||^2_{\Gamma_\omega} && \text{(effort regularization)} \\
&+ \kappa_u\big(\min\{0, u - \underline{u}\}^2 + \max\{0, u - \bar{u}\}^2\big) && \text{(thrust constraint penalty)} \\
&+ \kappa_{v_a} \min\{0, v_a - \underline{v}_a\}^2 && \text{(airspeed constraint penalty)}
\end{aligned}$$

The alignment regularization term entices the UAV to fly facing forward and right-side-up. Its weights can be made small as it just needs to resolve some arbitrary freedom in the orientation state (e.g. multirotors can translate invariant to their yaw).

This planner concisely meets the “one size fits all” goal because its dependence on UAV type/configuration is smoothly parameterized in the straight-forward way summarized in Table 3.1. Furthermore, hovering, loiters (orbits), and of course path tracking are all in a sense “emergent” behaviors specifiable through the same interface: path $r(t)$.

For example, if hovering is deemed physically feasible by the model, the optimization will “select” it to track a stationary point, else the closest trajectory in an L_2 sense will be a loiter. Finally, if the user wants to mandate a specific type of loiter (say they are nervous about whether the optimization will determine it autonomously), they can just explicitly specify that actual path.

Type	c_l	c_s	ω_z
Multirotor	0	0	$\pm$
Conventional Fixed-Wing	+	0	$\pm$
Aerobatic Fixed-Wing	+	+	$\pm$
Delta Fixed-Wing	+	0	0

Table 3.1: Simple parameter choices that configure the planner for various UAV types. The ω_z column expresses that for delta (rudderless) fixed-wings, one can just remove ω_z as a decision variable by assuming it is nominally zero.

MPC is currently the dominant state-of-the-art for robot motion planning, owing to the modern processors powerful enough to carry it out and advancements in efficient optimization routines. We solve our trajectory optimization problem via differential dynamic-programming (DDP), which is a second-order method with a rich theoretical foundation and history of successful application [4]. By Taylor-approximation of the dynamics, DDP allows for a natural handling of the $\mathbb{SO}3$ orientation part of the state.

3.2 Attitude Regulator

To achieve the angular velocity ω^* commanded by the planner, the regulator computes a torque $\tau(t) \in \mathbb{R}^3$ that will influence the angular velocity dynamic (Euler equations),

$$\dot{\omega} = M^{-1}(\tau + \eta - \omega \times M\omega) \tag{3.8}$$

where $M \in \mathbb{R}_+^{3\times3}$ is the positive-definite inertia tensor in body coordinates and $\eta(t) \in \mathbb{R}^3$ is the net uncontrolled aerodynamic torque in body coordinates. It is difficult to model η without extensive data collection that is usually out-of-scope for autonomy-focused programs, so we will rely strictly on feedback to compute τ.

Consider the following Lyapunov ("error energy") function,

$$\ell = \frac{1}{2}||\omega^* - \omega||^2 \tag{3.9}$$

Its rate-of-change (considering our dynamic) is,

$$\dot{\ell} = (\omega^* - \omega)^\intercal\big(\dot{\omega}^* - M^{-1}(\tau + \eta - \omega \times M\omega)\big) \tag{3.10}$$

We employ the usual feedback-linearizing control law [8],

$$\tau = M\big(\dot{\omega}^* + K(\omega^* - \omega)\big) + \omega \times M\omega - \hat{\eta} \tag{3.11}$$

where $K \in \mathbb{R}_+^{3\times3}$ is a positive-definite "proportional gain" tensor in body coordinates and $\hat{\eta}(t) \in \mathbb{R}^3$ is an adaptation we will define shortly. Substituting Eq. 3.11 into 3.10,

$$\dot{\ell} \;=\; -(\omega^* - \omega)^\intercal K(\omega^* - \omega) \;+\; (\omega^* - \omega)^\intercal M^{-1}(\hat{\eta} - \eta) \tag{3.12}$$

The leftmost term is negative-definite and thus always assists with decreasing ℓ (the error energy). If the rightmost term can be made negative-definite as well, then the error will definitely decay ($\omega^* - \omega \to 0$). We cannot specify $\hat{\eta}$ to guarantee this without the difficult modeling of η, but we can employ feedback yet again to adapt $\hat{\eta}$ online in a heuristically helpful way known as the "MIT rule" [7].

It is reasonable to assume that η is a function of not just the UAV's angular velocity, but also its translational velocity (and the wind vector, if measured). Specifically, the aerodynamic forces will generate a torque normal to the plane spanned by the UAV's velocity and its natural direction of forward flight (assumed to be x). This is the torque that either passively stabilizes the UAV's velocity into alignment with its forward direction (i.e. fights sideslip), or acts to flip the UAV backwards (if the aerodynamic center is ahead of the COM). Thus we finitely parameterize $\hat{\eta}$ as,

$$\hat{\eta}(\omega, v; W) = W\xi = W\begin{bmatrix}\omega \\ x \times v \\ 1\end{bmatrix} \tag{3.13}$$

with parameters $W(t) \in \mathbb{R}^{3\times7}$ (initialized to zero) to be learned online via gradient descent of the error energy rate-of-change. We compute the gradient of Eq. 3.12,

$$\nabla_{\mathrm{W}}\dot{\ell} = (\omega^* - \omega)^\intercal M^{-1}\nabla_{\mathrm{W}}\hat{\eta} \tag{3.14}$$

$$= M^{-1}(\omega^* - \omega) \otimes \xi \tag{3.15}$$

We adapt W to oppose this gradient with "learning-rate" $\Gamma \in \mathbb{R}_+^{3\times3}$ in order to make Equation 3.12 pursue negative-definiteness.

$$\dot{W} = -\Gamma M^{-1}(\omega^* - \omega) \otimes \xi \tag{3.16}$$

Equations 3.11, 3.13 and 3.16 together fully define the controller. Note that the affine 1-component of ξ serves to generate an ordinary "integral term." Thus, by choice of gains alone, this controller can reduce to the ubiquitous fixed-gain PI-loop that is used for most autopilot's angular velocity control.

Furthermore, if M is unknown (say the user doesn't have a mechanical CAD model of their UAV), it can be "absorbed into the gains" – i.e. just let M be an identity matrix and compensate the performance via tuning of K and Γ.

Finally, note that during this controller's development, the adaptive form in Equation 3.13 was originally a multilayer perceptron. However, during simulation tests, no additional performance was seen using hidden layers, so the form was reduced to the linear one shown here. Simulation results are shown in Figure 3.2.

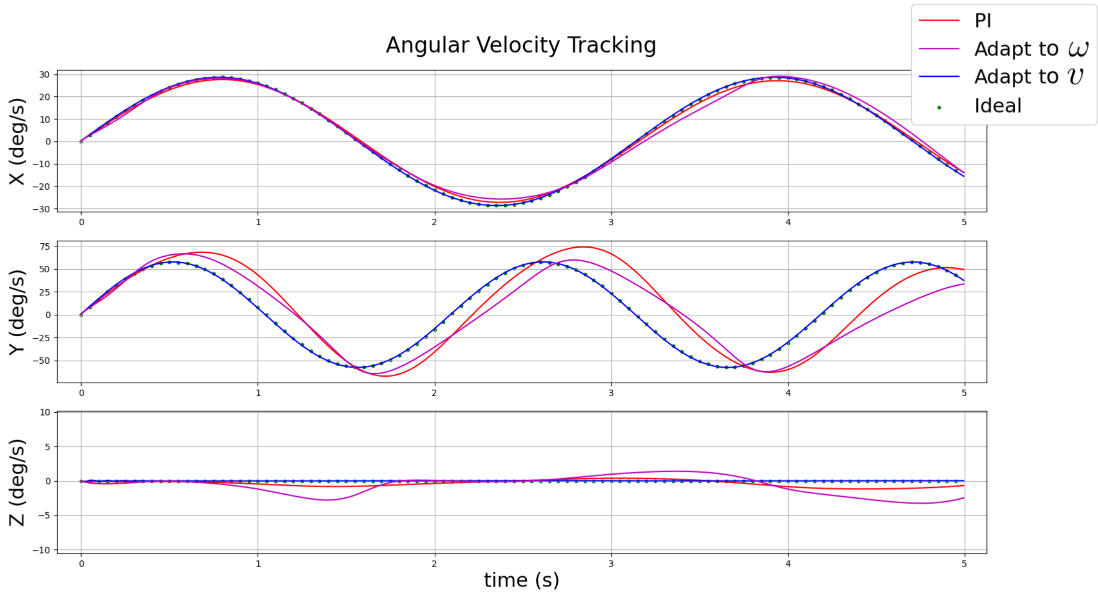

Figure 3.2: Simulation results for a fixed-wing UAV tracking an oscillatory angular velocity trajectory in flight using various controllers. The typical fixed-gain PI-loop ($\xi = 1$) tracks the ideal not too well. Using a gradient-based adaptation with $\xi = [\omega \ 1]^\intercal$ actually somewhat degrades performance. Finally, our proposed controller with $\xi = [\omega \ x \times v \ 1]^\intercal$ tracks very effectively.

3.3 Actuation Allocator

With the collective thrust u^* and torque τ^* now chosen (by the planner and regulator respectively), the allocator must decide on actuator effort signals that will best cause the UAV to experience a net actuator wrench (at the COM) of,

$$w^* = \begin{bmatrix} u^* \\ 0 \\ 0 \\ \tau^* \end{bmatrix} \tag{3.17}$$

The contribution that thrusters (motors with propellers) make to the net actuator wrench is relatively simple to model. As shown in Figure 3.3, a quadratic polynomial nicely maps actuator effort (PWM percent duty-cycle) to force magnitude [10]. We'll denote the scalar coefficients of this offset-free quadratic as c_1 and c_2. Reaction torque may also be measured with a slightly more sophisticated setup where the thruster is mounted to a torsional load-cell, and a linear fit is made from force to reaction torque with scalar coefficient c_r. The wrench from thruster i is then,

$$w_i = (c_{1_i}\zeta_i + c_{2_i}\zeta_i^2) \begin{bmatrix} \delta_i \\ l_i \times \delta_i + c_{r_i}\delta_i \end{bmatrix} \tag{3.18}$$

where $\zeta_i \in [0, 1]$ is the i^{th}-thruster's effort value, δ_i is its direction in body coordinates, and l_i is the vector from the COM to its mounting point in body coordinates. Conveniently, for trimmed mono-prop or counter-rotating bi-prop UAVs we have $c_{r_i} = 0$, $\delta_i = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix}^\intercal$, and $l_i \times \delta_i = 0_3$ (unless you want to utilize each propeller of the bi-prop independently for yaw authority – a neat trick to emulate a rudder).

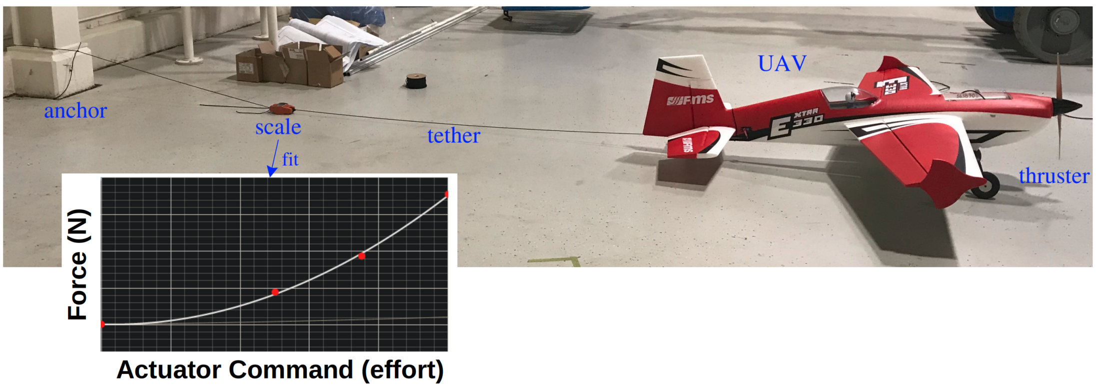

Figure 3.3: A simple test to characterize a UAV's thrusters. A few {effort (PWM), force} data points are collected and a quadratic is fit.

Control-surfaces are far more difficult to model. The usual paradigm for autopilots is to not model them, but instead to tune "mixing factors" that specify what proportionality of the torque is dedicated to each control-surface. (This is usually simple to do because on standard fixed-wing UAVs, each control-surface serves an independent rotational degree of freedom: aileron for roll, elevator for pitch, rudder for yaw). We can adopt the same computation in this wrench framework by approximating the wrench of the j^{th}-control-surface with normalized deflection $\zeta_j \in [-1, 1]$ as,

$$w_j = c_j \zeta_j \begin{bmatrix} 0_3 \\ \delta_j \end{bmatrix} \tag{3.19}$$

where δ_j is the "torque contribution direction" for that surface. E.g., $\delta_{\text{ailerons}} = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix}^\intercal$. A large c_j magnitude will result in a smaller ζ_j being allocated to surface j.

The control allocation problem can then be posed as,

$$\min_{\zeta} \left|\left| w^* - \Big(\sum_i w_i + \sum_j w_j \Big) \right|\right|^2 + \gamma_\zeta ||\zeta||^2 \tag{3.20}$$

$$\text{s.t.} \quad 0 \le \zeta_i \le 1 \quad \forall i \in \{\text{thrusters}\} \tag{3.21}$$

$$-1 \le \zeta_j \le 1 \quad \forall j \in \{\text{surfaces}\} \tag{3.22}$$

The L_2-regularization entices redundant actuators to be allocated in a way that minimizes power usage. This convex problem is solved by sequential quadratic programming (SQP).

3.4 State Estimator

All of the models presented so far are simplified and tailored for the purpose of the specific algorithms employing them. For example, the planner's aerodynamic model captures just enough phenomena to make sense of aerial maneuvering, but it relies on the repeated recalculations of the MPC-loop to actually be useful. The model otherwise has very limited predictive power.

A process model for state estimation generally needs predictive power, even if such predictions are probabilistic. Of course, this again begs for expensive data-collects and/or CFD to characterize all anticipated aerodynamic regimes and uncertainties. The robotics community circumvents this by leveraging the availability of high-quality inertial measurement units (IMUs). The accelerometer can effectively be used as f/m in Equation 3.2, and the gyroscope as ω in Equation 3.3.

Such IMU-driven process models fused with all other sensors (barometer, magnetometer, etc.) via an extended or unscented Kalman filter is considered the de-facto standard for UAV state estimation [12] [14]. Moving-horizon estimation (MHE) is growing in popularity for simultaneous localization and mapping (SLAM) on UAVs, but SLAM is still an active research field that Unitair is intended to sit beneath, not solve. If the

Unitair estimator is used directly with exteroception sensors like a LIDAR, it will require a pre-mapped environment and leverage the Gaussian particle-filter (GPF) technique detailed in [14] to meld nicely with the Kalman filter.

Unitair's estimator uses an extended Kalman filter (EKF), rather than unscented. This is because there is a tremendous amount of sparsity in the model Jacobians that can only be leveraged by an EKF to dramatically reduce computational load and increase bandwidth. (The largest Jacobian is only 10% dense). We argue that incorporating measurements at a higher rate is well worth the loss of second-order convergence theorized for the UKF.

The estimator tracks the following variables,

- $p \in \mathbb{R}^3$: Position in world-coordinates of the body-origin
- $v \in \mathbb{R}^3$: Velocity in body-coordinates of the body-origin relative to the world-frame
- $a \in \mathbb{R}^3$: Acceleration in body-coordinates of the body-origin relative to the world-frame
- $q \in \mathbb{SO}3$: Orientation of body-coordinates relative to world-coordinates, as unit-quaternion
- $\omega \in \mathbb{R}^3$: Angular velocity in body-coordinates of body-frame relative to world-frame
- $b_a \in \mathbb{R}^3$: Bias of the inertial acceleration input
- $b_\omega \in \mathbb{R}^3$: Bias of the inertial angular velocity input
- $\nu \in \mathbb{R}^3$: Velocity in world-coordinates of the local wind relative to the world-frame

The body-coordinates are defined by the IMU. Transformation of this state to COM-centered coordinates for the purpose of control is handled with a COM parameter in the code. Letting a prime denote the discrete Δt-advance operator, the process model is,

$$p' = p + R_q(v\Delta t + a\tfrac{\Delta t^2}{2}) \tag{3.23}$$

$$v' = v + a\Delta t \tag{3.24}$$

$$a = (\mu_a - b_a - \tfrac{\epsilon_a}{\sqrt{\Delta t}}) + R_q^\intercal g - \omega \times v \tag{3.25}$$

$$q' = qe^{\omega\Delta t} \tag{3.26}$$

$$\omega = \mu_\omega - b_\omega - \tfrac{\epsilon_\omega}{\sqrt{\Delta t}} \tag{3.27}$$

$$b_a' = b_a + \epsilon_{b_a}\sqrt{\Delta t} \tag{3.28}$$

$$b_\omega' = b_\omega + \epsilon_{b_\omega}\sqrt{\Delta t} \tag{3.29}$$

$$\nu' = \nu + \epsilon_\nu\sqrt{\Delta t} \tag{3.30}$$

where all ϵ variables are Gaussian white noise (Euler-Maruyama discretized), R_q is the vector-rotation operator corresponding to q, μ_a is the accelerometer measurement, and

μ_ω is the gyroscope measurement. Analytical Jacobians are provided in [12].

Note that the discretized evolution of the attitude state (as well as the computation of its Jacobian and innovation) follow the Lie-algebraic formulation described in [13]. EKFs that employ this formulation are sometimes referred to as "multiplicative" or "invariant" and are the dominant paradigm for high-performance attitude estimation.

The International Standard Atmosphere relations are used to model the proportionality between the barometer measurements and p_z (altitude). The World Magnetic Model is used to predict the local magnetic field, which is then subjected to hard/soft distortions and rotated to model the magnetometer measurements. The airspeed sensor is modeled as reading $v - \nu$ projected onto its direction and clipped positive. Motion-capture and GPS are currently treated as "loosely-coupled position measurements" [12].

The process model is continually integrated with the most recent IMU measurements (the integration timestep can be much smaller than the IMU update rate). At each loop, any queued sensor measurements are fused via the EKF correct step in the order they arrived. Measurement timestamps are used to discard extremely outdated information.

3.5 Simulation

Because all of Unitair's algorithms are run on a single SBC, software-in-the-loop (SIL) and hardware-in-the-loop (HIL) simulations are the same thing. The simulator itself can be run on the UAV's SBC, or the Unitair algorithms can be run on a development machine (the same loop-rates are achieved in either case).

Unitair can in principle be used with any UAV simulator, but we provide immediate integration with Stokesim: an efficient, MITLL-developed UAV simulator.

The Unitair repository contains a Stokesim plugin as well as a handful of example Stokesim configuration files for different UAV types. Please see the Stokesim repository for more details on what the simulator itself provides or how plugins work.

3.6 Organization

The Unitair source-code can be found here:
`https://github.com/Unitair-Autopilot`

A major tenet of Unitair is that the source-code should be organized in a way that reflects the algorithm architecture, so that it is extremely easy to transition from a mathematical description of Unitair to its implementation. By design, Unitair's algorithms can support a wide variety of UAV types with simple runtime parameter choices, and thus there is not a landslide of "features." The four main algorithms (planner, regulator, allocator, and estimator) are each primarily broken into four files:

- <algorithm>.hpp: Header file declaring a class Algorithm and supporting structures
- <algorithm>.cpp: Implementation file for the associated header
- <algorithm>_test.cpp: Stand-alone executable unit-test of the implementation (also serves as a demo of basic usage)
- <algorithm>_node.cpp: ROS node that instantiates an Algorithm and connects it to the other ROS nodes according to Figure 1.1.

Note that the *only* files that depend on ROS are the *_node.cpp files. Thus, Unitair can easily be extended to other middlewares by writing alternative "nodes" (or whatever they might be called), with all processing code held pristine.

Every declaration is currently documented with Doxygen. However, the creation of a detailed documentation wiki with walk-throughs is mandatory during the rest of Unitair's development.

Unitair's dependencies are minimal, requiring only some de-facto libraries for C++: Eigen, yaml-cpp, and ROS (just for the nodes). (ROS itself depends on Eigen and yaml-cpp so just installing ROS is actually sufficient). The algorithms and drivers are implemented cleanly and efficiently from scratch, thus *tailoring* them for the purpose of Unitair (as opposed to roping in a variety of general libraries that would make it harder to find "where the math actually happens" and increase the source complexity by requiring the reader to become familiar with many more APIs).

At the root directory, Unitair provides a ROS package complete with launch and parameter files. The launch files are applicable to any system, while the parameter files serve as examples for specific UAVs. Users are expected to,

1. Install the Unitair ROS package on their SBC (a typical catkin build)
2. Copy into their mission-specific repository the example Unitair parameter files for a UAV most similar to theirs
3. Modify the copied parameters to suit their actual system
4. Call Unitair's launch files with the required cfg_path parameter set to their mission-specific directory.

Each Unitair node provides a custom reconfigure ROS service that can be used to make that node reload the parameters at cfg_path. Thus on-the-fly tuning is simply performed by modifying the parameter files and calling the respective service, even mid-flight. Because the parameter files themselves are modified to do this, every parameter change made is immediately tracked by the user's version control (presumably git). This makes for very clean parameter calibration and tweaking in the field.

Unitair uses custom ROS messages to remain self-contained and clear (by mimicking Figure 1.1 as closely as possible). Unitair provides a Python node referred to as the "visualizer" that can be run on the user's groundstation computer to relay relevant Unitair messages as ROS visualization messages which can be displayed in RViz.

Unitair comes with a high-level state-machine written in Python referred to as the "manager." It provides an interface for a human to specify Unitair's mode (essentially just "standby" or "path-tracking") and to input paths. For more complex autonomy, the user is expected to replace the manager with their own nodes that live on top of Unitair. This is what the "Mission" block of Figure 1.1 represents.

4 Conclusion

We have presented Unitair, an autopilot designed to alleviate the majority of GNC-related development issues faced by UAV autonomy programs at MITLL. Unitair is currently at the end of its two year development as a TI. All of the algorithms have been implemented and tested full-stack on hardware as shown in Figure 4.1.

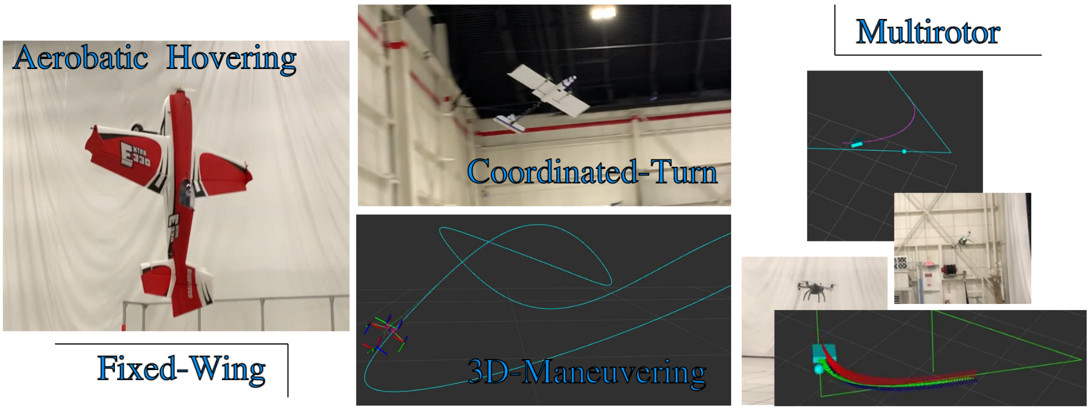

Figure 4.1: Various flight tests of Unitair.

5 References

[1] Lutkebohle, “Determinism in Robotics Software” at ROSCON, Vancouver, 2017.

[2] Kay, “Real-time control in ROS and ROS 2” at ROSCON, Hamburg, 2015.

[3] FAA, “Unmanned aircraft systems,” 2019. Retrieved online.

[4] Mayne, David Q. “Differential dynamic programming - a unified approach to the optimization of dynamic systems.” Control and Dynamic Systems. Vol. 10. Academic Press, 1973. 179-254.

[5] Tassa, Yuval, Nicolas Mansard, and Emo Todorov. “Control-limited differential dynamic programming.” IEEE ICRA, 2014.

[6] Falanga, Davide, et al. “PAMPC: Perception-aware model predictive control for quadrotors.” 2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS). IEEE, 2018.

[7] Mareels, Iven MY, et al. “Revisiting the MIT rule for adaptive control.” Adaptive Systems in Control and Signal Processing 1986. Pergamon, 1987. 161-166.

[8] Nezvadovitz, “Observer-Side Parameter Estimation For Adaptive Control.” (2017).

[9] Johansen, Tor A., and Thor I. Fossen. “Control allocation - a survey.” Automatica 49.5 (2013): 1087-1103.

[10] Yomchinda, Thanan. “Simplified Propeller Model for the Study of UAV Aerodynamics using CFD method.” 2018 5th Asian Conference on Defense Technology (ACDT). IEEE, 2018.

[11] Humpherys, Jeffrey, Preston Redd, and Jeremy West. “A fresh look at the Kalman filter.” SIAM review 54.4 (2012): 801-823.

[12] Kok, Manon, Jeroen D. Hol, and Thomas B. Schön. “Using Inertial Sensors for Position and Orientation Estimation.” Foundations and Trends in Signal Processing 11.1-2 (2017): 1-153.

[13] Hertzberg, Christoph, et al. “Integrating generic sensor fusion algorithms with sound state representations through encapsulation of manifolds.” Information Fusion 14.1 (2013): 57-77.

[14] Bry, Adam, et al. “Aggressive flight of fixed-wing and quadrotor aircraft in dense indoor environments.” The International Journal of Robotics Research 34.7 (2015).