FlyZone: A Testbed for Experimenting with Aerial Drone Applications

Mikhail Afanasov*‡, Alessandro Djordjevic*, Feng Lui*, and Luca Mottola*†

*Politecnico di Milano (Italy), ‡Credit Suisse (Poland) †RI.Se SICS Sweden

ABSTRACT

FlyZone is a testbed architecture to experiment with aerial drone applications. Unlike most existing drone testbeds that focus on low-level mechanical control, FlyZone offers a high-level API and features geared towards experimenting with application-level functionality. These include the emulation of environment influences, such as wind, and the automatic monitoring of developer-provided safety constraints, for example, to mimic obstacles. We conceive novel solutions to achieve this functionality, including a hardware/-software architecture that maximizes decoupling from the main application and a custom visual localization technique expressly designed for testbed operation. We deploy two instances of FlyZone and study performance and effectiveness. We demonstrate that we realistically emulate the environment influence with a positioning error bound by the size of the smallest drone we test, that our localization technique provides a root mean square error of 9.2*cm*, and that detection of violations to safety constraints happens with a 50*ms* worst-case latency. We also report on how FlyZone supported developing three real-world drone applications, and discuss a user study demonstrating the benefits of FlyZone compared to drone simulators.

CCS CONCEPTS

• **Computer systems organization** → *Robotics*; *Embedded software*; *Dependable and fault-tolerant systems and networks*; • **Software and its engineering** → *Development frameworks and environments*;

KEYWORDS

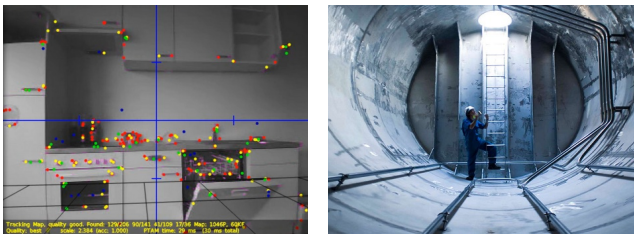
drones, testbeds, localization, dependability

ACM Reference Format:

Mikhail Afanasov, Alessandro Djordjevic, Feng Lui, and Luca Mottola. 2019. FlyZone: A Testbed for Experimenting with Aerial Drone Applications. In *the 17th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys '19), June 17–21, 2019, Seoul, Republic of Korea.* ACM, NY, NY, USA, 13 pages. https://doi.org/10.1145/3307334.3326106

MobiSys '19, June 17–21, 2019, Seoul, Republic of Korea

ACM ISBN 978-1-4503-6661-8/19/06…$15.00
https://doi.org/10.1145/3307334.3326106



(a) The parameters of the existing implementation are optimized for a domestic environment [15].

(b) Oil tanks lack the visual features of Fig. 1a, and require a new set of parameters to efficiently operate SLAM.

Figure 1: Target environments for SLAM.

1 INTRODUCTION

Aerial drones represent a new breed of mobile computing. They enable sophisticated applications largely unfeasible with any other technology, such as gathering high-resolution imagery in an automatic fashion [38], and collecting fine-grained environmental data in near-inaccessible areas [8]. Experimenting with aerial drone applications, however, is currently an ordeal.

Motivation. Representative of an oil company eventually asked us to prototype a drone system to perform automatic visual inspections of oil tanks [40], using low-cost drones. These are hostile environments, as gases originating from chemical residuals abound.

Using drones therefore represents a viable alternative, but the necessary functionality is non-trivial. Oil tanks are GPS-denied environments and cannot be instrumented beforehand. Autonomous navigation is to be achieved using visual or dead-reckoning techniques [27]. The former tend to be more precise [16], but their performance is sensitive to the environment visual features.

We started off from an existing implementation of simultaneous localization and mapping (SLAM) for the AR.Drone 2.0 [15, 58]. Note that this was already way more mature than many existing implementations of autonomous navigation functionality: it was extensively tested using real drones [15], and offered optimized parameters for the target drones.

We created oil tank mock-ups and started experimenting. Initial tests were disastrous; the existing implementation turned out to be very inaccurate in navigating the environment. In total, we broke seven drones crashing against walls and ceilings, not to count damages to objects nearby. The reason is intuitively shown in Fig. 1: the parameters driving the recognition of visual landmarks were tuned for objects of shape, color, and size different from the inner side of an oil tank.

The implementation we used offers several knobs to tune SLAM. Experimenting with different parameter values, however, was extremely laborious as every inaccurate setting eventually resulted

in a crash. This required fixing broken parts, checking the system sanity, and rebooting the application with different parameters.

Challenge. Oil tank inspections are merely an example. Developers of aerial drone *applications* are confronted with similar issues in a range of diverse domains, including ambient intelligence [19] and search-and-rescue missions [33].

Existing techniques enabling autonomous behaviors rarely work out of the box, as they are typically tested only in simulation, or require significant application- and hardware-specific customization [51]. Many drone applications may also benefit from multiple collaborating drones [8]. Investigating distributed interactions further complicates matters. Developers are thus confronted with how to verify that their implementations meet the requirements at stake.

Drone simulators exist [4, 34], which are however simplified compared to reality, are unable to model application-specific sensors, and are often limited to waypoint GPS-driven navigation. Robot simulators [20, 22, 24, 29, 45] often focus on aspects such as swarm behaviors, targeting scenarios with thousands of unsophisticated resource-constrained devices. In comparison, drones are much more powerful platforms, able to operate in a stand-alone fashion. Moreover, simulators may require developers to use different languages compared to actual systems, which duplicates development efforts.

Because of these reasons, experimenting with drone applications tends to take place right in the target settings [43]. This is often a recipe for disaster. Ensuring the safety of objects and people when running prototype implementations is extremely difficult, while the consequences of bugs may be catastrophic [57]. The ever-changing regulations on the use of civil drones compound the problem [18]. The result is that most drone applications are still manually operated [57].

Contribution. As further discussed in Sec. 2, drone testbeds exist. However, the vast majority of them focuses on *low-level mechanical control*, using expensive *motion capture systems* for localization and *highly-engineered* drones [32, 35, 62]. The outcomes of the experimentation are therefore difficult to translate into application-specific operation in a given environment. Developers of drone applications rather require a means to run *high-level application* functionality by emulating the *features of the target deployment setting*, using *commercially-available* drones.

FlyZone fills this gap. It provides drone developers with an application-level API and ways to emulate environment influences, such as wind or pressure gradients, while developer-provided safety constraints are automatically monitored that mimic the presence of physical obstacles. For example, using FlyZone a developer may test an application's reaction to lateral forces on the drone, or express constraints on what trajectories a drone is allowed to fly based on a virtual representation of oil tanks. Any violation to these constraints prompts FlyZone to reclaim control of the drone and execute developer-provided fail-over actions.

Achieving such a functionality requires to address conceptual as well as technical challenges:

(1) To ease the transition from testbed to deployments, we must decouple the testbed infrastructure from the application. To this end, we design a lightweight system architecture and a set of dedicated application-level APIs, described in Sec. 3.

Figure 2: Components in drone platforms. *The ground control station let users configure high-level mission parameters, the autopilot software implements the low-level motion control aboard the drone.*

(2) Realistically emulating the environment influence is extremely difficult, and hides subtle interactions with the flight control logic. To address this, we conceive a custom technique to reproduce the effect of external forces, described in Sec. 4.
(3) Drone localization in a testbed is a different problem than robot localization in a target application. Moreover, FlyZone must not interfere with other localization systems used by the main application. We illustrate in Sec. 5 a custom visual localization technique that addresses these needs.

FlyZone provides several benefits. Unlike simulators, the additional development effort due to FlyZone is arguably small, while fidelity increases. Experimentation occurs using the same application code and drone platforms to be deployed in the target setting. In addition to portability across drone platforms, our design facilitates replicating the testbed infrastructure at different sites in a fully customized fashion. To that end, we do offer a well-defined set of procedures and scripts to automate this effort [3]. FlyZone is entirely built with off-the-shelf commercial hardware, which facilitates obtaining the equipment and reduces costs.

Two working installations of FlyZone currently exist, located at Politecnico di Milano, Italy and University of Virginia, US. Both are actively used by researchers at either institution. Sec. 6 describes these prototype installations, which are also instrumental to evaluate the performance of FlyZone. The results we present in Sec. 7 indicate, for example, that we realistically emulate the environment influence with a positioning error bound by the size of the smallest drone we test, that our localization technique provides a root mean square error of only $9.2cm$, and that safety violations are detected in under $50ms$.

Finally, Sec. 8 reports on our experience using FlyZone for developing prototype drone applications that reached real deployments. We also discuss the results of a user study comparing the use of FlyZone with a widespread drone simulator for developing one of these applications. Further, we illustrate how FlyZone is unexpectedly doubling as a training facility for professional drone pilots seeking to acquire the official license [17], as much as we discuss limitations of our work.

2 BACKGROUND

FlyZone supports drone developers experimenting with *high-level application* functionality. Therefore, it differs from existing literature, which mainly investigates low-level mechanical control.

Platforms. Drone platforms are often architected as in Fig. 2. Application software runs at a ground-control station (GCS), which is

typically a standard computer that communicates with the drone using a long-range radio. On the drone, the high-level commands from the GCS are translated into low-level motor operation by the autopilot software [7], which runs on a dedicated embedded computing unit [46].

Platforms also exist where the application logic runs aboard the drone, on what is called "companion computer" [64]. This allows application developers to create sophisticated autonomous functionality, such as vision-based navigation [14], that can run independent of external infrastructure. The companion computer is typically a Linux box [50, 52, 64] that connects to the autopilot board through a UART interface.

Developers use robot operating systems [51] or custom drone APIs [2]. The high-level application-specific functionality running on the GCS or the companion computer continuously exchanges data with the autopilot. This is necessary as applications need to adapt to the instantaneous run-time conditions, such as environment influences or the appearance of obstacles. Handling these cannot be delegated to the autopilot but in the simplest scenarios, such as waypoint navigation.

FlyZone supports both GCS-based platforms and companion computers. When using a GCS, the FlyZone hardware to be mounted on the drone adds no run-time overhead and little weight. When using a companion computer, FlyZone is sufficiently lightweight to be deployed on that and rely on the same connection to the autopilot board. Thus, no additional hardware is required.

Testbeds. While we focus on *application-level* experimentation using *commercially-available* drones, existing drone testbeds concentrate on low-level mechanical control, using highly-engineered drones backed by computing clusters and expensive motion capture systems [32, 35, 62]. The objective of the experimentation is not to test application implementations, but to investigate fine-grained flight regulators that enables demonstrations such as drones throwing and catching balls [54] or building physical structures [28].

Few robot testbeds exist to experiment with application-level functionality. Nonetheless, for example, Saeed et al. [56] do not offer a generic application-level API and use a single ceiling camera to track ground robots. Their technique aims to tell the robots apart from the surrounding, and is thus inherently site-specific. The median localization error of 50*cm* with robots in fixed positions is arguably too large to ensure safety of flying drones, and much greater than what we show in Sec. 7.

Common to these solutions, and unlike FlyZone, is the tight integration of the testbed software and hardware with the main application. Our design keeps the two as decoupled as possible, to facilitate transitioning from testbed operation to deployment.

Localization. In general, localizing *drones in a testbed* is a distinct problem compared with *robot localization in a given application scenario*. In the former, absolute accuracy and speed are key, while the environment may be freely instrumented. Localizing drones in a given application scenario, instead, typically assumes little existing infrastructure, whereas a limited loss of accuracy is accepted as long as application requirements are met.

The question is whether any of the latter techniques may be employed for drone localization in a testbed. For example, Hirose et al. [23] use pictographs to localize ground robots. Their technique is very precise, yet the processing times are prohibitive for accurate control of flying drones. AprilTag [41] and AprilTag 2 [65] are closest to the approach we use for testbed-level localization. While their accuracy is comparable to ours, Sec. 7 demonstrates that the processing times are inappropriate for use in FlyZone.

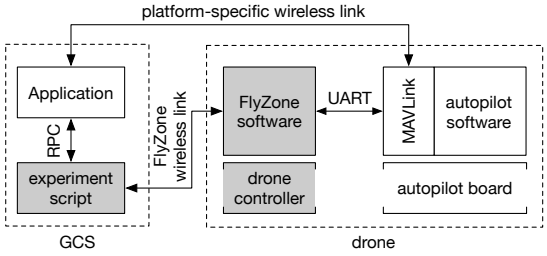


Figure 3: FlyZone architecture. *Components in grey are FlyZone-specific.*

A large body of work exists in SLAM [14]. An example for commercially-available drones is the work of Engel et al. [15], who report a localization accuracy comparable to ours. Their technique targets unknown environments and limits the degrees of freedom. However, in a testbed, the environment is known and the drone must be able to move freely. Similar considerations apply to the use of optical flow methods [21].

Works exist using ultra-wide band (UWB) technology with drones. The key issue here is the presence of outliers [25]. For operation in a testbed, these may threaten the ability to promptly react whenever the application loses control. Moreover, UWB-based solutions do not provide orientation information per se, forcing the use of techniques that likely introduce errors that accumulate over time [6].

Laser scanners aboard the drones [55] provide best accuracy among existing solutions. However, they tend to impact a drone's lifetime because of the added weight. We rather aim at providing a solution with minimal added overhead. Finally, solutions exist for camera tracking of mobile entities. For example, Ctrax [10] tracks swarms of walking flies. These systems are, however, designed to meet very different goals than ours. Ctrax, for example, is designed to provide a quantitative behavior analysis tool to the neuroethology community.

3 ARCHITECTURE

The design of FlyZone has two objectives. First is to decouple the testbed infrastructure from the main application. Second is to support both GCS-based platforms and companion computers.

Fig. 3 shows a lightweight architecture that achieves these objectives. The picture shows the case where the application runs on a GCS; minimal variations apply if the application is deployed on a companion computer.

Drone controller. We deploy an additional single-board computer on every drone, termed as *drone controller*. This is responsible for executing the testbed commands to emulate the influence of the environment; for example, by steering the drone in arbitrary directions and for checking violations to safety constraints. It runs a minimal Linux installation and a custom FlyZone software that exchanges flight commands and telemetry data with the autopilot,

using the MAVLink [49] protocol though a UART interface. The controller also serves part of the localization system, as described in Sec. 5, so it is informed of the location.

An alternative to this design would be to rely on the autopilot board. Doing so would require altering the time-sensitive control loops responsible of flight control, which is in general not advisable [43] and bound to be platform-specific. Moreover, autopilot boards are extremely resource-constrained, and unlikely to have unused resources. The cost for our choice is a possible reduction in flight times due to the additional weight of the controller and its battery. Currently available technology, however, offers a range of low-cost options [5, 52], most of them imposing limited additional weight and operating for hours using small powerbanks.

Experiment script. The controller receives commands from, and forwards telemetry data to an *experiment script*. This specifies the actions developers are interested in, for example, to create given environment situations, and the safety constraints for the experiment. The script may be deployed on the GCS or on a companion computer. In the former configuration, a FlyZone-specific wireless link is to be used to connect to the controller.

Only one instance of experiment script exists in the system. When deploying multiple drones, the FlyZone API allows one to apply different sequences of actions to different drones, possibly depending on their mutual interactions. If communication between the main application and the experiment script is required, developers use a remote procedure call stub we provide. In this case, the code for using the stub would be the only place in the main application that needs changes when moving to an actual deployment.

APIs. FlyZone is implemented in Java. Testbed operations may be invoked from an experiment script in Java or from application code using other languages, provided the corresponding stub is available. Right now, we provide stubs for Java, Python, and C++. We only summarize the key traits of the FlyZone API here, but further details are available [3].

Key to writing FlyZone experiment scripts is an interface called `Drone` we provide. It provides controls for the single drone using a coordinate system determined by the localization technique we illustrate in Sec. 5. When invoking any of these operations, the corresponding controller accordingly instructs the autopilot. This interface includes three kinds of operations.

First, it provides basic functionality such as taking off or moving to a certain coordinate. These operations are useful to create a given initial situation. For example, when testing obstacle avoidance techniques, developers may want to boot the system by placing drones according to specific patterns relative to the obstacles.

The second kind of operations are useful to emulate the environment influence onto drones, such as wind. These operations accept data structures mapping every location in the testbed to a vector of external forces whose effect is reproduced by FlyZone onto the drone, as explained in Sec. 4 This functionality is useful when experimenting with applications targeting outdoor deployments; for example, search-and-rescue operations.

The third kind of operation allows developers to state safety constraints that mimic the presence of physical obstacles or protect drones and their surroundings. For the visual inspection of oil tanks described earlier, developers rely on this to create virtual fences corresponding to the shape of tanks. When invoking any of these operations, the constraints are installed on the controller, which monitors them continuously.

Finally, interfaces are provided to process real-time updates of drone position and navigation data including speed, acceleration, roll, pitch, yaw, and battery level. Classes implementing this interface are also notified of violations to the safety constraints, so corrective actions may be applied.

4 ENVIRONMENT INFLUENCE

When emulating the environment influence, the drone controller issues commands to the autopilot to steer the drone *as if* it was subject to the corresponding forces. Applications cannot distinguish these dynamics from actual environment influence, as both are detected at the higher levels through MAVLink telemetry packets.

Two main challenges exist: *i)* how to tune the commands from the drone controller to the autopilot to yield a realistic behavior, discussed in Sec. 4.1, and *ii)* how to handle the interplay between these commands and those issued by the application when it reacts to the perceived environment influence, discussed in Sec. 4.2. The following description builds on fundamentals of flight dynamics that, in the interest of brevity, we include in an accompanying technical report [3].

4.1 Drone Dynamics

Intuitively, to tackle the first challenge, we are to reverse-engineer the physical behavior of the drone. External forces are normally applied to the drone by the environment. In response to that, the drone moves in certain ways. We need to do the opposite: we want to proactively move the drone in a way that reproduces its physical response to external forces. We thus need a model to tell how the drone moves when subject to external forces of a given strength and duration, so the drone controller can replay those movements.

Design options. We provide two solutions to this problem. On one hand, we rely on the vast literature [66] on aircraft dynamics and adapt existing detailed models to FlyZone. Note that we need a model for general navigation, unlike task-specific models [54].

Our models require some, still reasonable, effort in parameter estimation. This is needed only once and solely in case FlyZone does not integrate a given type of drone already. We currently support 24 different drones, ranging from custom quad-, hexa-, and octocopters to commercial devices such as DJI Spark, Mavic Pro, and the whole Phantom and Inspire series.

Should a developer not be willing to invest effort in parameter estimation for a detailed model, we provide a ready-to-use alternative that approximates the drone as a point in space corresponding to its center of mass. A linear model determines the parameters of the commands issued by the drone controller to the autopilot.

Such an approach only requires knowledge of the approximate weight at take-off. Considering the shirking sizes of drones and their relative dimensions compared to the surrounding space, they may be considered as a point in a three-dimensional space. Moreover, weight distribution is both even and concentrated in the middle of the drone, where cameras and batteries are installed. Although these assumptions are not unreasonable, ease of use comes at the

cost of reduced realism, as this model disregards factors such as drag and inertia due to the specific body shape.

We describe next the detailed models and omit the approximate ones for brevity. Further details are available nonetheless [3]. Both modeling approaches are appropriate for *rotor symmetrical* drones. Fixed-wing and Y6 configurations [1] require different approaches.

Model structure. Aerial drones are 6-degree of freedom (DOF) rigid bodies [66]. Long-established approaches to model their dynamics are the Euler-Lagrange and Newton-Euler methods. The former is more compact, but the latter lends itself to incorporate external forces, which is our goal.

With the Newton-Euler method, two frames of reference are used: one is fixed to the Earth and termed as *navigation* reference; the other is fixed with the body of the drone. Let $\mathbf{p}^n$ and $\mathbf{v}^n$ be the drone position and velocity vectors in navigation coordinates (denoted with n), and $\mathbf{w}^b$ be its angular rate in body coordinates (denoted with b). Its dynamics are described as the instantaneous change in position and velocity of its center of mass in the navigation frame, plus the instantaneous change in angular rate in the body frame. By applying fundamental laws of kinematic, we write

$$\dot{\mathbf{p}^n} = \mathbf{v}^n \tag{1}$$

$$\dot{\mathbf{v}^n} = m^{-1}\mathbf{C}_b^n\mathbf{F} \tag{2}$$

$$\dot{\mathbf{w}^b} = \mathbf{J}^{-1}\mathbf{M} \tag{3}$$

where $\mathbf{F}$ and $\mathbf{M}$ are the sum of forces and torques on the drone, m and $\mathbf{J}$ are its mass and inertia, and $\mathbf{C}_b^n$ transforms body coordinates into navigation coordinates.

The solution to eq. (1)-(3) are the necessary and sufficient information to steer the drone as if it was subject to given external forces. We translate the solution to low-level MAVLink commands the drone controller issues to the autopilot. We describe next how to obtain the necessary drone-specific inputs and parameters. Position information are provided by the FlyZone localization system, described in Sec. 5.

Model inputs. We apply existing methods to compute the torque vector $\mathbf{M}$ in eq. (3) [11]. The force vector $\mathbf{F}$ in equation (2) is a combination of drag forces $\mathbf{F}_d$, motor thrust $\mathbf{F}_m$, and gravity $\mathbf{F}_g$. We compute $\mathbf{F}_m$ based on the relation between input current and output thrust for brushless DC motors [9], which typically equip aerial drones. We obtain input currents based on the transformation parameters used in electronic speed converters (ESCs). These are in charge of translating the pulse width modulation (PWM) signal from the autopilot into a current flow to the motors. The ESC firmware is typically open-source, even for commercial drones [12]; thus the translation parameters are generally available. The gravity vector $\mathbf{F}_g$ is straightforward.

The remaining force vector represents drag forces $\mathbf{F}_d$, which we express as

$$\mathbf{F}_d = -\mathbf{C}_{\mathbf{d},\mathbf{F}}\mathbf{C}_n^b|\mathbf{v}^n - \mathbf{v}_w^n|(\mathbf{v} - \mathbf{v}_w) \tag{4}$$

where $\mathbf{C}_{\mathbf{d},\mathbf{F}}$ is the diagonal drag coefficient matrix and $\mathbf{v}_w^n$ is the velocity vector of external forces applied to the drone [31]. Using FlyZone, the latter is input to a given experiment and represents the knob developers use to set the environment influence. They may provide this information as a constant velocity vector that applies throughout the field, using meteorological software [48] to generate detailed three-dimensional wind maps, like in Fig. 4a, or by synthetically creating wind patterns modeling specific situations, such as narrow passages or airflows around objects [38].

Parameter estimation. The remaining unknown quantities are the drone mass m, inertia $\mathbf{J}$, and diagonal drag coefficient matrix $\mathbf{C}_{\mathbf{d},\mathbf{F}}$. We compute m by weighing individual components. We use a three-dimensional drone model in Blender combined with the BGE Advanced Physics Library and custom scripts we develop to estimate $\mathbf{J}$ and $\mathbf{C}_{\mathbf{d},\mathbf{F}}$. Blender is a state-of-the art open-source 3D editing software, often used to create 3D models of drones for optimizing their aerodynamics. The BGE Advanced Physics Library is included in the Blender distribution and is used to understand the physical properties of Blender models. These software packages are included in the FlyZone distribution.

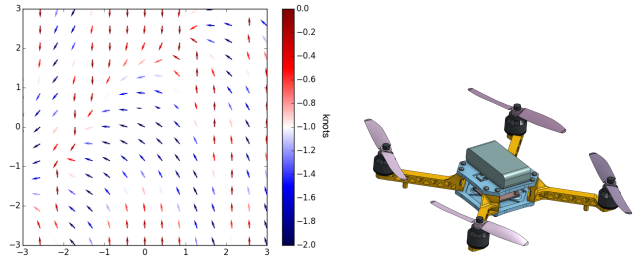
Throughout this process, some approximations are inevitable to keep the problem tractable. We use 3D models as exemplified in Fig. 4b, which omit the presence of smaller components such as ESCs, antennas, and the like. Further, our scripts compute propeller drag only based on length and maximum rotation speed, similar to existing literature [9], rather than considering the specific propeller shape. Finally, the drone aerodynamic drag and rotational drag coefficients are computed separately and then arithmetically combined, whereas they are known to have limited, but still non-zero influence on each other [13].

4.2 Executions

With either drone model providing the necessary inputs to the autopilot to emulate the environment influence, the next question is how to manage the interplay between these inputs and the application inputs to the autopilot, whenever it reacts to these influences.

There are, in fact, subtle interactions that may occur. The drone controller issues commands to the autopilot at a preconfigured rate. As soon as the application detects the environment influence, that may also issue further commands to the autopilot to counteract the corresponding effects. Say any two of these commands reach the autopilot within an interval ϵ smaller or equal to the flight control period. The behavior would be at best undefined, as *two different inputs* are to be processed for the same control loop iteration.

Design options. One way to address these situations is to modify the autopilot implementation. Whenever two commands are received within ϵ, the autopilot first computes the corresponding



(a) Wind map in the plane generated using QGis software. **(b) Quadcopter Blender model.**

Figure 4: Inputs for environmental influence and parameter estimation.

control loop outputs separately, then linearly combines the resulting force and torque vectors before outputting the PWM signals.

We argue that this solution would be greatly impractical. Autopilots are time-sensitive software functionality. This kind of modifications are likely to be very difficult to carry out, as they require intimate knowledge of the autopilot implementation and run the risk of affecting the stability of control loops [43]. In addition, most commercial drones do not allow one to freely modify the autopilot implementation.

We rather adopt a few key precautions to rule out these situations, requiring no changes to the autopilot implementation. First, we increase the autopilot control rate to the maximum supported setting, that is, we run the flight control loop as often as possible. We do this to make ϵ as small as possible, at the price of a slight increase in energy consumption of the autopilot board. This is negligible compared to the energy draw of motors [7]. Crucially, a smaller ϵ decreases the likelihood of the above situations, still without completely ruling them out. Unlike loading a new autopilot firmware, this is possible also on commercial drones [12].

If the autopilot implementation supports priorities for MAVLink packets, we set the ones emulating the environment influence to the lowest priority. Such a setting gives priority to application packets whenever those fall within the same ϵ as the ones from the drone controller. It has no effect otherwise. If MAVLink priorities are not supported, we reduce the size of the MAVLink input packet queue to one. Thus, the first such packet that arrives is the one that the autopilot processes. Any other packet is silently discarded. Using either approach, we could fully integrate the aforementioned 24 different drone types.

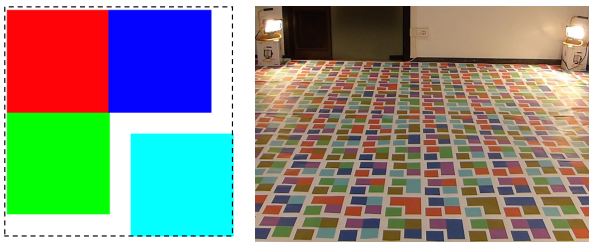
The combination of these techniques may result in a slight reduction of realism, yet Sec. 7 demonstrates that this is minimal in concrete applications.

5 LOCALIZATION

Drone localization must happen with accuracy, speed, and limited processing overhead. We are expecting FlyZone installations to be located indoor, so GPS is generally not applicable. Moreover, Sec. 2 argues how motion capture systems are expensive and laborious to install, whereas application-level robot localization is inapplicable.

We opt to design a custom visual localization technique that is entirely based on off-the-shelf technology. Our technique uses *visual tags* like the one in Fig. 5a, deployed on the ground at known positions, as shown in Fig. 5b. The tags are dynamically recognized through a camera connected to the drone controller and attached to the bottom of the drone, pointing to the ground. The tags provide localization and orientation information in the plane. Note that such a technique is solely meant for testbed operation and is replaced with an application-specific localization system when deploying the system, for example, using GPS or SLAM [15]. We obtain altitude information from the autopilot software, based on the readings of ultrasound sensors part of a drone's IMU and employed to this end also in real deployments.

We use Java to implement the localization pipeline we described next, using OpenCV [42]. The pipeline may run on a central computer that simply receives the video feed from the drone controller, or entirely on the latter. Using a central computer, we may leverage



(a) Visual tag. **(b) Tag space.**

Figure 5: Localization system in FlyZone.

GPU acceleration, whereas using the drone controller we can provide positioning information locally to the drone, thus emulating the presence of an on-board GPS sensor. The software components implementing the localization pipeline are fully decoupled from the rest of the testbed architecture; this enables replacing the visual techniques we describe next with alternative solutions depending on where the testbed is installed. When installing FlyZone outdoor, for example, GPS may be simply used.

Next, Sec. 5.1 discusses the tag design and placement. In Sec. 5.2 we illustrate how we derive positions.

5.1 Tag Design

Every tag is composed of a number of squared *tiles* of different colors and corresponds to a numerical value.

To decode the value, we employ a positional notation and a one-to-one mapping between colors and numerical values. The tag in Fig. 5a represents a specific point in such a space; the corresponding numerical value is obtained by decoding the numerical values of the individual tiles in a clockwise manner starting from the lower-right corner. The latter is identifiable independent of camera orientation as one of the tiles is intentionally misplaced. In this example, the tag in Fig. 5a represents the value 3201 in base-4. Every value corresponds to a position in the plane.

Dimensioning. For a given FlyZone installation, we are to determine how many colors c one should employ and how many tiles n should be included in a single tag. This is a function of the area covered by the testbed on the ground plane, camera resolution and angle of view, as well as maximum and minimum flying height.

We derive analytical expressions tying the values of c and n to these three parameters, in a way that *i)* ensures the camera constantly sees multiple tags, which provides redundancy, and *ii)* maximizes the accuracy in decoding individual tags. Such a derivation, which we report in a companion report [3], eventually leads to a Pareto front of optimal solutions for c and n.

To find a practical design point, we proceed as follows. Say k is the minimum probability a camera guarantees in distinguishing two colors, as shown in Sec. 6. We can experimentally find the maximum number of colors c_{max} corresponding to such probability. Based on our analytical derivations [3], c_{max} corresponds to a minimum value n_{min} of the number of tiles.

Dually, the number of tiles per tag is limited above because of the camera's field of view at the minimum flying height. Such a maximum number of tiles n_{max} corresponds to a minimum number of colors c_{min} [3]. The upper and lower bounds for c and n

determine a narrower band of admissible values in the Pareto front, where the selection is simpler. Sec. 6 demonstrates this based on an actual installation of FlyZone.

Placement. Once appropriate values for c and n are identified, it remains to be determined *where* each of the possible c^n tags is placed within the considered area.

A random or sequential placement may, in fact, be sub-optimal. As mentioned above, chances are that if one tile is incorrectly recognized in a tag, the same tile is also incorrectly recognized in any adjacent tag. This may happen because uneven lighting conditions or camera distortions are likely to apply to a slice of the camera field. This is why we should avoid placing tags with the same or similar tiles near each other. The more diverse are the tiles in adjacent tags, the less likely is an incorrect recognition of a tile to propagate to multiple tags within the camera's field.

We turn the placement of tags into an optimization problem. Based on the color distance [47] between *tiles*, the color distance between any two *tags* T_1 and T_2 is

$$dist(T_1, T_2) = \sqrt{\sum_{i=1}^{n} (color_{iT_1} - color_{iT_2})^2} \quad (5)$$

where $color_i$ indicates the color value of the i-th tile according to the clockwise ordering described in Sec. 5.1. The expression above is equivalent to considering a tag to be an n-dimensional vector in the color space.

Based on eq. (5), we formulate an optimization problem that seeks to *maximize color distance between adjacent tags*, including diagonals. The inputs are:

1) Decision variables to represent the locations; therefore, there are at most c^n of these.
2) Bounds for every such variable to assume a value between 0 and $c^n - 1$.
3) A constraint that forces every variable to assume a unique value, as we cannot re-use the same tag.

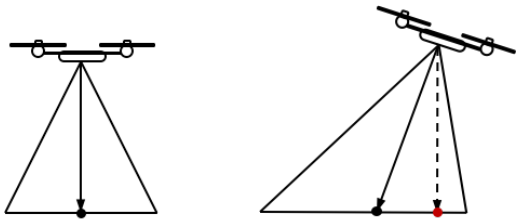
The assignments to the decision variables encode the tag placement. We use MiniZinc [37] as solver; with the installations in Sec. 6, MiniZinc returns a near-optimal solution in a few hours.

5.2 Deriving Positions

First, we need to recognize tags in the camera field. We use standard image-processing techniques. First, we apply a thresholding operator to convert the image to black and white. Next, we apply a contour finding operator to identify the tiles, including the misplaced one to determine the orientation. Finally, we map this information back to the original image, read the tiles' colors, and decode the value of the tag.

Recognizing individual tags is not sufficient. Some of the tags in the camera field may be incorrectly recognized. Moreover, the view of the camera is parallel to the ground only if the drone hovers stably, as in Fig. 6a, but drones move in space by inducing pitch and roll. As shown in Fig. 6b, this changes the perspective of the camera on the ground, distorting the image. Installing a gimbal to correct this would increase costs and weight.

To address these issues, we proceed with two additional steps once individual tags are recognized:



(a) Hovering. *The image center corresponds to location.*

(b) Moving. *The drone location is not at image center.*

Figure 6: The camera perspective on the tag space changes when drones move.

1) **Reference mapping:** based on the set of recognized tags in an image, we reconstruct the slice of the coordinate system as seen by the drone. The latter allows us to spot the incorrectly recognized tags. We know, in fact, that tags follow a specific spatial sequence as determined in Sec. 5.1.
2) **Compensation:** based on IMU information, we recreate the frame of reference as seen by the camera, correcting the distortions induced by pitch and roll. This process is repeated for every tag in the image. The results are then averaged to obtain final positioning and orientation information.

The output of the process is a tuple of three values: coordinates $\langle x, y \rangle$ and orientation α in the horizontal plane. The third coordinate is obtained from the autopilot software that already compensates the readings of the ultrasound sensor on the drone's IMU.

6 INSTALLATION

We offer a few practical considerations on deploying an instance of FlyZone at a target site. These are based on the two existing installations, covering a space of about $80sqm$ and $144sqm$ respectively.

We use Parrot's AR.Drone 2.0 as well as custom quadcopters and hexacopters based on Ardupilot and PixHawk. The AR.Drone 2.0 is GCS-based, whereas we deploy a companion computer on the custom drones. We use a RaspberryPI Zero W with a RaspiCam V2 as drone controller and input for localization. The RaspberryPI is powered by a separate battery. The total added weight is 89 grams.

Tags. We determine the maximum number of colors the camera can distinguish with a minimum probability k, which we set to 0.9. We perform a sequence of experiments to check the RaspiCam's capability to distinguish c different colors at distances from the minimum and maximum flying height at the target site[1].

It turns out the camera detects six different colors with almost absolute certainty. Given $c_{max} = 6$, we calculate $n_{min} = 4$ [3]. This way, we obtain the upper (lower) bound for c (n). With similar experiments, we determine the camera's field of view and accordingly identify an upper (lower) bound for n (c). Based on analytical derivations [3], we calculate $n_{max} = 17$ and a corresponding $c_{min} = 2$.

The optimal values for $\langle c, n \rangle$ thus range from $\langle 2, 17 \rangle$ to $\langle 6, 4 \rangle$. We choose $c = 6$ and $n = 4$ because smaller values for c do not improve the detection probability; as that is already equal to 1 for

[1]These experiments only require the camera and sample print-outs of test colors at maximum color distance [47]; the camera is not on the drone. The lighting conditions during these experiments are described next.

Component	Price (€)	Quantity
Printed tag space	225	1
400W halogen lights	13	4
RPi Zero W and RaspiCam V2	60	8
Desktop computer	1500	1
Miscellaneous	100	
Total	2500	

Figure 7: Approximate cost of FlyZone. *The prototype supports up to eight drones. The total estimated cost is ≈ 2500 €, excluding the drones to be used in the testbed.*

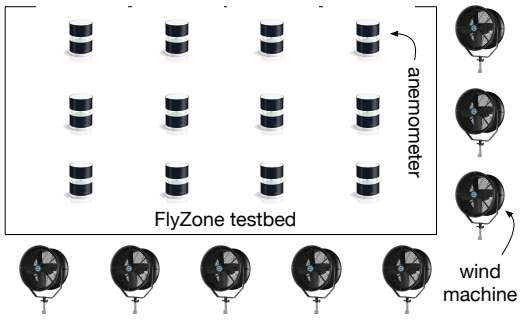


Figure 8: Evaluating realism in emulating environment influence: testbed setup to gain ground truth.

$c = 6$. Conversely, larger values for n are detrimental to the the robustness of tag recognition, as illustrated in Sec. 5.1. We thus calculate all relevant quantities and generate the tag space [3].

Deployment. We use plastic nets to protect the surroundings of the experimentation area. To ensure proper illumination of the tag space, we employ four halogen light sources placed at the corners of the testbed area, below the minimum flying height and pointed to the center. We use halogen bulbs as they do not produce strobing light. This setup guarantees that the angle of direct light is too small to cause glaring, and that the drones never cast their shadows on the tag space.

Fig. 7 indicates the bill of materials for one of the FlyZone installations. The figures for the other installation are similar. The cost varies with how many drones need to be simultaneously supported, which is eight in this case. The machine we use for the localization pipeline when not running on the drone controllers is equipped with an Intel Xeon E3 v1270 CPU and a cheap NVIDIA GPU. The total—excluding drones—is comparable to a high-end modern laptop.

7 EVALUATION

We evaluate the performance of FlyZone along two dimensions, using the installation at Politecnico di Milano, Italy. In Sec. 7.1, we investigate how realistic is the emulation of environment influence; Sec. 7.2 reports on the performance of localization and detection of safety violations.

7.1 Environment Influence

Quantifying how realistically we emulate the environment influence is a challenge per se. The key problem is how to gain some form of measurable ground truth.

Setting. Fig. 8 intuitively describes the physical setup we design to this end. We rent eight 17" wind machines of the type used in professional movie making. These offer accurate power and orientation settings, which we use to create repeatable wind patterns in the three dimensions. To measure the effect before any drone is deployed, we install 48 portable anemometers measuring flow speed and direction, uniformly in the three-dimensional testbed area. They are installed on top of thin poles, which minimally affect air flows. We linearly interpolate their single data points to create a three-dimensional wind map of the kind in Fig. 4a.

As a form of ground truth, we run the test applications described next against the actual influence of the wind machines, using the same settings used for creating the windmaps, but without the poles and anemometers in the field. For comparison, we input the windmaps to FlyZone to recreate the corresponding environment effect. We experiment with both the DETAILED model and the APPROXIMATE one, described in Sec. 4.

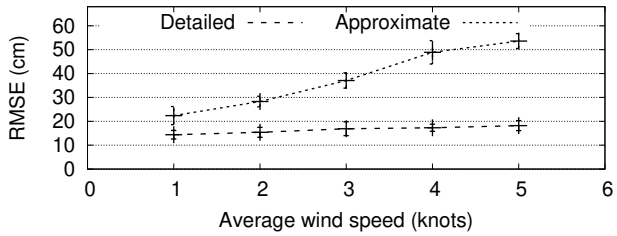
We develop a single-drone application called TRAJECTORY that directs the drone along regular three-dimensional paths, such as cubes and helices. The application may run in two different modes: in COMPENSATION mode, it tries to counteract the effect of wind machines to maintain the shape of the trajectory; in SIMPLE mode, it does nothing to that end. In addition, we emulate a search-and-rescue application developed by external users of FlyZone using five drones, as described in Sec. 8, and call it SEARCH.

We track the drones using an OptiTrack motion capture system [63]. We compute the Root Mean Square Error (RMSE) of drone positions and orientations between the path flown in the ground truth setting and when using FlyZone. We consider this as a measure of FlyZone realism. When emulating the environment influence, the drone controller inputs MAVLink commands to the autopilot twice per second. We experiment with a total of 62 different wind machines configurations, creating a variety of air flow patterns. We repeat the same experimental setting ten times to factor out inaccuracies in the setup, using either the AR.Drone 2.0 or our custom PixHawk-based hexacopter. Each experiment lasts 20 minutes. We total 400+ hours of tests.

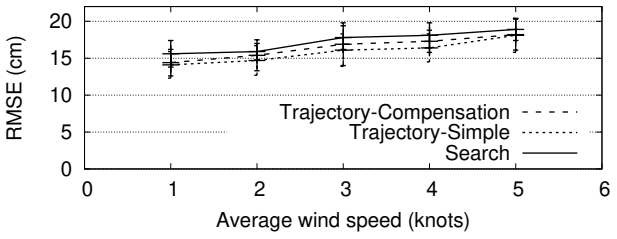
Results. Fig. 9 plots the RMSE results in drone positions; we obtain similar trends for orientation. We test average wind speeds up to 5 knots; current regulations advise not to fly beyond these conditions [17, 61].

Fig. 9a investigates the position performance against variable average wind speeds. The DETAILED model yields realistic performance and is minimally affected by increasing intensity of external forces. The absolute RMSE are constantly lower than the physical dimensions of the smallest drone we use. The APPROXIMATE model has comparable performance at low average wind speeds, but quickly looses realism as the latter grows. At 5 knots, the RMSE becomes comparable with the size of the AR.Drone 2.0. These trends are expected; the more intense are the external forces, the more the physical features that the APPROXIMATE model does *not* represent bear an impact.

Fig. 9b plots the position performance as a function of the tested application. The results are very similar and follow the same trend, demonstrating the general applicability of our approach. The slight differences are due to the interplay between drone controller and



(a) Realism depending on model.



(b) Realism depending on application.

Figure 9: FlyZone performance in emulating the environment influence.

application. The TRAJECTORY-SIMPLE case shows the lowest average RMSE because the application does nothing to counteract the environment influence, so the situations of Sec. 4.2 never occur. Conversely, SEARCH uses a complex application logic that constantly tries to correct the trajectory. Nonetheless, the solution we describe in Sec. 4.2 to handle the interplay between this and the drone controller yields a minimal decrease in realism.

The absolute values as well as the trends hitherto discussed are similar for the AR.Drone 2.0 and our custom hexacopter. The two are strikingly different in physical terms; the AR.Drone 2.0 weighs $380g$ and is $58cm$ wide, whereas our custom hexacopter weighs $1.2Kg$ and is $94cm$ wide. This indicates that the models of Sec. 4 remain accurate also with different types of drone.

Key to the performance in emulating environment influences are both the models and the accuracy of position information, which we investigate next.

7.2 Localization and Safety Violations

The localization system in FlyZone is a foundation for many testbed functionality. In addition to increasing realism when emulating the environment influence, high-frequency drone tracking improves how promptly we recognize violations to safety constraints.

Setup. We consider the same predefined flight paths we used in Sec. 7.1. We trace the location updates returned by FlyZone and compare against two baselines.

One baseline is AprilTag 2: a state-of-the-art visual technique widely employed for application-level localization [65]. We apply the same placement technique and reference mapping of FlyZone, as in Sec. 5. This factors out any aspect not related to the tag design. The comparison is meant to demonstrate that a custom technique is necessary for testbed operation. The other baseline is the OptiTrack system [63] used in Sec. 7.1, which we employ as ground truth. Note that FlyZone and AprilTag 2 employ two different types of

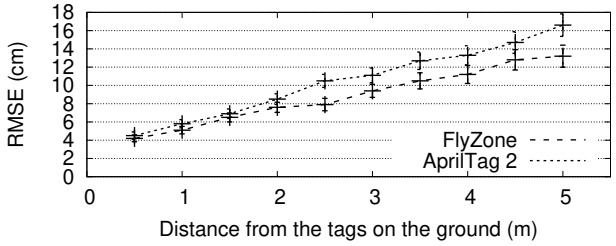


Figure 10: Localization accuracy of FlyZone compared with AprilTag 2 with a single drone, at varying distances from the tags.

tags. Therefore, they can be compared only relative to the common baseline provided by the OptiTrack system.

We perform a total of 180 experiments at different heights and speeds between $2m/s$ and $6m/s$, each lasting 20 minutes. We deploy up to eight drones in separate experiments to measure the performance whenever the camera view is partially occluded by other drones. We collect 90.000 data points in total. We compute the RMSE compared to OptiTrack as a measure of accuracy, and the number of Frames Per Seconds (FPS) as an indication of processing performance compared to AprilTag 2. As each frame in the video input yields a single location update, the FPS correspond to the rate of location updates.

Note that the video input for localization comes from the drone controller. Therefore, the results we describe next equally apply when using any other drone with the same drone controller.

Results: localization. The accuracy performance of AprilTag 2 is comparable, or at times moderately worse than FlyZone. For example, FlyZone shows an RMSE of $9.2cm$ ($2.1°$) in location (orientation) against $11.8cm$ ($3.8°$) for AprilTag 2. The worst-case location outlier is $12.9cm$ ($15.1cm$) for FlyZone (AprilTag 2). Compared with UWB in an almost identical setting [25], the performance of either system is markedly better in accuracy, and one order of magnitude lower in worst-case outlier. UWB localization alone cannot provide orientation information.

Fig. 10 plots the localization RMSE in the ground plane, against a variable distance from the tags. This is the main parameter determining the performance of visual localization techniques, including AprilTag 2 [65]. As the distance from the tags increases, the error of AprilTag 2 grows slightly faster than FlyZone. Similar observations apply to orientation.

In a multi-drone scenario, we investigate the same figures as a function of the percentage of frames where a drone appears in the camera's field of another drone. Note we can control this parameter by forcing trajectories to cross a given number of times. Also consider that multiple drones appearing in the same frame are considered as separate instances, as they typically occlude different slices of the tag space.

Fig. 11 reports the localization RMSE in the ground plane; we observe again similar trends for orientation. FlyZone shows better resilience than AprilTag 2 to partial occlusions of the camera field. FlyZone's performance only marginally worsens with an increasing fraction of partially occluded frames compared to AprilTag 2, which are not designed to incorporate any form of intra-tag redundancy. This shows in the increasing RMSE and in the increasing

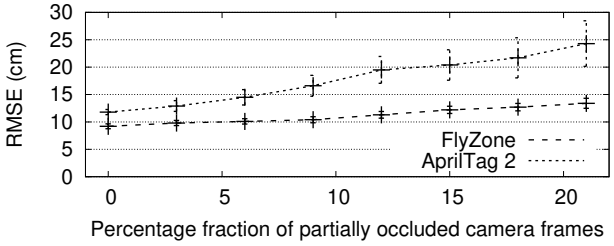
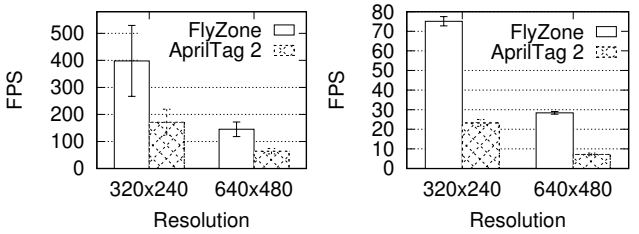


Figure 11: Localization accuracy of FlyZone compared with AprilTag 2 in the presence of partial occlusions of the camera view.



(a) Desktop computer. (b) Drone controller.

Figure 12: FPS depending on image resolution, as returned by FlyZone or AprilTag 2.

standard deviation around the RMSE of AprilTag 2 compared to FlyZone. Intuitively, with eight drones flying simultaneously, the scenario is already quite crowded.

Crucially, Fig. 12 shows the FPS performance of FlyZone compared with AprilTag 2. It shows that FlyZone is twice (three times) as fast than AprilTag 2 when the localization pipeline runs on a desktop computer (drone controller). Such an improvement is enabled by the simpler design of FlyZone tags, which are straightforward to recognize and decode, thus requiring fewer computing resources. Using a smaller resolution to improve performance, at the cost of lower accuracy, does not change the trends in Fig. 12. The desktop computer mounts a low-end GPU, which offers an order of magnitude performance advantage over the RaspberryPI Zero W. Where to deploy the localization pipeline is thus a trade-off between FPS and the ability to provide positioning information at the drone controller.

The localization technique in FlyZone thus offers only slight accuracy improvements over AprilTag 2 in a single drone scenario. However, it shows marked advantages in accuracy with multiple drones and in processing speed, which are key for a multi-drone testbed and instrumental to check safety constraints efficiently.

Results: safety. We investigate the latency to detect safety violations using a variable number of AR.Drone 2.0 flying at variable speeds during a dummy experiment, until one of them is purposely directed towards a forbidden area. The OptiTrack system determines when the drone's location should trigger the violation. We measure the time until the experiment script is notified, using NTP for time synchronization. We repeat every experiment 10 times and average results.

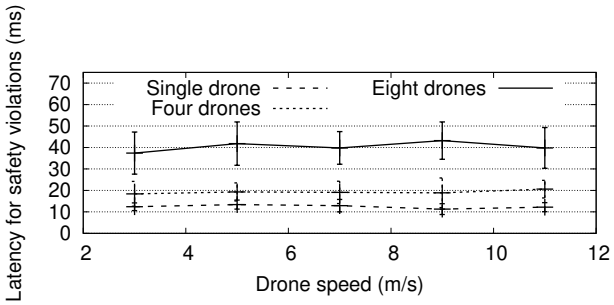


Figure 13: Latency in detecting safety violations.

Whenever the drone controller runs the localization pipeline, the detection of safety violations happens locally. In this situation, we measure a nearly constant latency of about 40*ms*, regardless of the speed and number of drones. Differently, Fig. 13 shows the latency in detecting safety violations whenever the localization pipeline runs on a desktop computer. Speed still plays no role, yet the number of drones becomes a factor as the WiFi network is increasingly loaded, due to funneling real-time video from the drone controller.

The absolute values in Fig. 13 are, however, limited even with eight drones. In the worst case, a drone flying at $11m/s$—top speed for the AR.Drone 2.0—travels about half a meter in around 50*ms*. This is within the safety margins for obstacle avoidance [26]. To further improve this figure, one may use different WiFi networks over different channels if possible or use packet prioritization.

8 APPLICATION EXPERIENCE

We report on the experience using FlyZone for developing applications that eventually reached real-world deployments, and on a user study comparing FlyZone with a simulator. The observations we draw provide evidence that the quantitative results we described in Sec. 7 apply to realistic settings. We conclude by discussing alternative uses and limitations.

Oil tanks. The application outlined in the Introduction was the original motivation for designing FlyZone. We used safety constrains to indicate where the drone was allowed to fly inside our mock-up oil tank. Every violation to these constraints was detected by FlyZone before we had lost control of the drone. The experiment script stopped the main processing, moved the drone back to the initial position, changed relevant parameters, and restarted the test. These experiments could ran in a semi-automatic fashion.

Once we could rely on this functionality, it took five days of work to find efficient SLAM parameters. This is very little time compared to the two months we spent hopelessly trying to identify suitable values without being able to prevent mishaps. Our final prototype was eventually demonstrated in public, autonomously navigating mock-up oil tanks of arbitrary shapes and colors [30]. Across 16 hours of such experiments, the system always correctly navigated the planned paths showing no unintended behaviors.

Obstacle avoidance. Within a graduate course on embedded software, we gave students a task to develop an ambient intelligence application that uses drones to find lost items in a house. The items send a radio beacon the drone can locate with room-level accuracy.

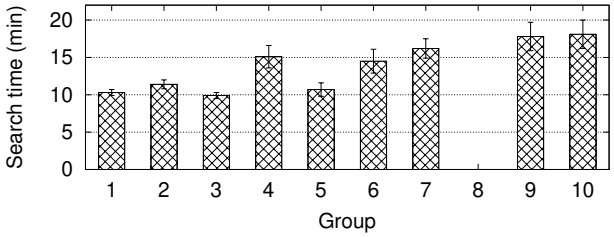


Figure 14: Search times for the search-and-rescue application for different groups in the final challenge trials.

The students started with the same SLAM system we used for oil tank inspections [58]. The default parameters were already optimized for a domestic environment, as shown in Fig. 1a. The challenge was to extend it to: *i)* drive navigation based on radio beacons, and *ii)* avoid obstacles. To that end, the students equipped the drone with a BLE radio and a mmWave sensor [59].

It turned out that the functioning of SLAM was affected by the presence of obstacles. Whenever flying *above* one, the ultrasound sensor reported a decrease of height, due to the signal bouncing on the object rather than the ground. The mapping step of SLAM was then led to think that the perspective on entire scene suddenly changed, producing an unstable behavior.

FlyZone's safety constraints were key again. The students created mock-ups of a domestic environment and specified the space occupied by physical obstacles as not allowed for drones. This helped them extend the existing implementation to handle this specific case. FlyZone supported the development of the additional functionality. The effort was significant, as the additional code eventually accounted for more than half of the final implementation.

Such a development effort would be impossible without FlyZone. The students entered the final stage of TI's Innovation Challenge with this work [60].

Search and rescue. We ran a student challenge comparing the use of FlyZone with the SITL [4] simulator, the de-facto standard for simulating MAVLink-based drone platforms. The students worked in pairs to create a prototype search-and-rescue application using a custom hexacopter with a Raspberry PI 3 companion computer and an ARVA radio receiver for finding people under snow [44]. The objective was to minimize search times. We recruited a total of 20 last-year M.Sc. graduate students with multi-course expertise in software engineering and embedded systems. For development, half of the students used FlyZone, the other half used SITL.

Unlike the previous examples, the students could not rely on an existing implementation and started from a textbook description of a gradient descent algorithm [53]. They were also required to extend the system to multiple collaborating drones to reduce search times [8, 36]. This functionality had to be developed from scratch.

Development times were generally in favour of the groups using SITL, who invested about 33% fewer hours. It goes without saying that no testbed may ever match the ease of use of a simulator. Looking at the actual system performance, however, turned things in favor of the groups using FlyZone.

We measured the search times in the final challenge trials based on four individual runs of the prototypes in a rugby field. The drone and the ARVA transmitter were initially placed at opposite ends. This site was unknown to the students until they turned in the final implementations. We used the same digital anemometers of Sec. 7.1 to make sure the running conditions were comparable across groups.

Fig. 14 plots the results. Group one to five, who used FlyZone, show better performance than the other groups who used SITL, but group #4. Group #8, who used SITL, never completed the search. The application logic was very similar among the different groups, as it was based on the same search algorithm. The parameter tuning made the difference. Using FlyZone to emulate the environment influence led the groups to eventually obtain more efficient parameters able to withstand the environment effects.

Other (non-)uses. FlyZone at ⟨OMISSIS⟩ is also helping a local piloting school train pilots towards obtaining the official license for flying professionally [17].

Although this was never among our goals, FlyZone's features are useful in this case too. We are running no application; the drone is manually controlled. We wrote an experiment script that specifies safety constraints to make sure even the most novice pilot can do no harm. The same script triggers different "trials" to check whether the pilot can deal with environment influences, for example, due to wind gusts. FlyZone is currently the only indoor infrastructure that pilots can use to learn how to fly in realistic conditions.

FlyZone has limitations in scope. As we focus on application-level experimentation, testing changes in the autopilot software is not possible and we consider the autopilot as trusted. Moreover, specific functionality may necessarily require experimenting in the target site. An example is the detection of drones based on signatures in their wireless transmissions [39]. The features that enable such detection are inherently a function of the deployment setting; therefore the results of experimentation in a testbed—FlyZone or any other—unlikely translate to real deployments.

9 CONCLUSION

FlyZone is a testbed infrastructure to support developing aerial drone applications. Its unique features include the ability to emulate the environment influence, which we achieve with a positioning error bound by the size of the smallest drone we test, and the automatic monitoring of safety constraints that mimic obstacles, whose violations we detect in under 50*ms*. A custom visual localization techniques, providing positioning errors as low as 9.2*cm*, enables this performance, while a lightweight testbed architecture that maximizes decoupling from the main application facilitates transitioning from testbed to real deployments. We provide evidence of FlyZone effectiveness based on three real-world aerial drone applications we developed with FlyZone and a user study comparing FlyZone support with the SITL simulator.

Acknowledgments. We thank Kamin Whitehouse for feedback on the FlyZone development. This work was partly supported by the Italian Ministery of Education, University, and Research through the cluster project "ICT Solutions to Support Logistics and Transport Processes (ITS)" of and by VINNOVA, the Swedish Innovation Agency through project "DePILOT".

REFERENCES

[1] 3D Robotics. [n. d.]. Y6 RTF Drone. goo.gl/7sqRF5.
[2] 3DRobotics. [n. d.]. Dronekit. goo.gl/IGX2t5.
[3] Mikhail Afanasov, Alessandro Djordjevic, Feng Lui, and Luca Mottola. 2018. FlyZone: A Testbed Infrastructure for Aerial Drone Applications. goo.gl/uPWnp1. Technical report.
[4] Ardupilot. [n. d.]. SITL simulator. goo.gl/PLfx1g.
[5] BeagleBoard.org. [n. d.]. BeagleBone Black. goo.gl/RBZQGL.
[6] J. Borenstein and L. Feng. 1996. Measurement and correction of systematic odometry errors in mobile robots. *IEEE Transactions on Robotics and Automation* 12, 6 (1996).
[7] E. Bregu et al. 2016. Reactive Control of Autonomous Drones. In *Proceedings of ACM MOBISYS*.
[8] Wolfram Burgard et al. 2000. Collaborative multi-robot exploration. In *Proceedings of ICRA*.
[9] Anežka Chovancová, Tomáš Fico, L'uboš Chovanec, and Peter Hubinsk. 2014. Mathematical modelling and parameter identification of quadrotor (a survey). *Procedia Engineering* 96 (2014).
[10] Ctrax. [n. d.]. The Caltech multiple walking fly tracker. goo.gl/QHgzVU.
[11] GCHE De Croon, KME De Clercq, R Ruijsink, B Remes, and C De Wagter. 2009. Design, aerodynamics, and vision-based control of the DelFly. *International Journal of Micro Air Vehicles* 1, 2 (2009).
[12] DJI. [n. d.]. Phantom Drone. goo.gl/wLbVic.
[13] Wei Dong, Guo-Ying Gu, Xiangyang Zhu, and Han Ding. 2013. Modeling and control of a quadrotor UAV with aerodynamic concepts. In *Proceedings of World Academy of Science, Engineering and Technology*.
[14] Hugh Durrant-Whyte and Tim Bailey. 2006. Simultaneous localization and mapping: part I. *IEEE robotics & automation magazine* 13, 2 (2006), 99–110.
[15] Jakob Engel, Jürgen Sturm, and Daniel Cremers. 2012. Camera-based navigation of a low-cost quadrocopter. In *Intelligent Robots and Systems (IROS)*.
[16] Jakob Engel, Jürgen Sturm, and Daniel Cremers. 2014. Scale-aware navigation of a low-cost quadrocopter with a monocular camera. *Robotics and Autonomous Systems* 62, 11 (2014).
[17] Hillary B Farber. 2014. Eyes in the sky: constitutional and regulatory approaches to domestic drone deployment. *Syracuse L. Rev.* 64 (2014), 1.
[18] Federal Aviation Administration. [n. d.]. Regulations for Unmanned Aircraft Systems. goo.gl/cahhbk.
[19] Dario Floreano and Robert J Wood. 2015. Science, technology and the future of small autonomous drones. *Nature* 521, 7553 (2015), 460.
[20] B. Gerkey et al. 2003. The Player/Stage project. In *Proceedings of of ICAR*.
[21] Bruno Herissé, Tarek Hamel, Robert Mahony, and François-Xavier Russotto. 2012. Landing a VTOL unmanned aerial vehicle on a moving platform using optical flow. *IEEE Transactions on robotics* 28, 1 (2012).
[22] David Hiebeler et al. 1994. The swarm simulation system and individual-based modeling. In *Proceedings of Decision Support 2001: Advanced technology for natural resource management*.
[23] K. Hirose et al. 2011. Camera-based localization for indoor service robots using pictographs. In *Proceedings of IEEE/ASME International Conference on Advanced Intelligent Mechatronics (AIM)*.
[24] Bryan Kate, Jason Waterman, Karthik Dantu, and Matt Welsh. 2012. Simbeeotic: a simulator and testbed for micro-aerial vehicle swarm experiments. In *Proceedings of the International Conference on Information Processing in Sensor Networks (IPSN)*.
[25] Benjamin Kempke, Pat Pannuto, and Prabal Dutta. 2015. PolyPoint: Guiding Indoor Quadrotors with Ultra-Wideband Localization. In *Proceedings of HotWireless*.
[26] O Khatib. 1986. Real-time obstacle avoidance for manipulators and mobile robots. *International Journal of Robotics Research* 5, 1 (1986).
[27] Lindsay Kleeman. 1992. Optimal estimation of position and heading for mobile robots using ultrasonic beacons and dead-reckoning. In *Proceedings of the International Conference on Robotics and Automation*.
[28] Quentin Lindsey, Daniel Mellinger, and Vijay Kumar. 2012. Construction with quadrotor teams. *Autonomous Robots* 33, 3 (2012).
[29] Sean Luke, Claudio Cioffi-Revilla, Liviu Panait, Keith Sullivan, and Gabriel Balan. 2005. Mason: A multiagent simulation environment. *Simulation* (2005).
[30] MEETmeTONIGHT - Milano. [n. d.]. Face research and researchers. www.meetmetonight.it.
[31] Johannes Meyer, Alexander Sendobry, Stefan Kohlbrecher, Uwe Klingauf, and Oskar Von Stryk. 2012. Comprehensive simulation of quadrotor UAVs using ROS and Gazebo. In *International conference on Simulation, Modeling, and Programming for Autonomous Robots*.
[32] N. Michael et al. 2010. The GRASP Multiple Micro-UAV Testbed. *IEEE Robotics Automation Magazine* (2010).
[33] F. Michahelles, P. Matter, A. Schmidt, and B. Schiele. 2003. Applying wearable sensors to avalanche rescue. *Computer and Graphics* 27, 6 (2003).
[34] Jalil Modares, Nicholas Mastronarde, and Karthik Dantu. 2016. UB-ANC emulator: An emulation framework for multi-agent drone networks. In *Proceedings of the International Workshop on Wireless Network Testbeds, Experimental Evaluation, and Characterization*.
[35] D. I. Montufar et al. 2014. Multi-UAV testbed for aerial manipulation applications. In *Proceedings of International Conference on Unmanned Aircraft Systems (ICUAS)*.
[36] L. Mottola, M. Moretta, C. Ghezzi, and K. Whitehouse. 2014. Team-level Programming of Drone Sensor Networks. In *Proceedings of ACM SENSYS*.
[37] Nicholas Nethercote et al. 2007. MiniZinc: Towards a standard CP modelling language. In *International Conference on Principles and Practice of Constraint Programming*.
[38] F. Nex and F. Remondino. 2003. UAV for 3D mapping applications: A review. *Applied Geomatics* (2003).
[39] Phuc Nguyen, Hoang Truong, Mahesh Ravindranathan, Anh Nguyen, Richard Han, and Tam Vu. 2017. Matthan: Drone Presence Detection by Identifying Physical Signatures in the Drone's RF Communication. In *Proceedings of MOBISYS*.
[40] NTR Labs. [n. d.]. UAVs for oil tank inspections. goo.gl/aHfPK8.
[41] E. Olson. 2011. AprilTag: A robust and flexible visual fiducial system. In *Proceedings of ICRA*.
[42] opencv [n. d.]. OpenCV: open-source computer vision. opencv.org.
[43] A. Patelli et al. 2016. Model-based Real-time Testing of Drone Autopilots. In *Proceedings of DRONET*.
[44] Pieps. [n. d.]. ARVA Transceivers. goo.gl/tPywra.
[45] Carlo Pinciroli et al. 2012. ARGoS: A modular, parallel, multi-engine simulator for multi-robot systems. *Swarm intelligence* 6, 4 (2012).
[46] PixHawk.org. [n. d.]. PX4 autopilot. goo.gl/wU4fmk.
[47] Fatih Porikli. 2003. Inter-camera color calibration by correlation model function. In *Proceedings of the International Conference on Image Processing (ICIP)*.
[48] QGis Project. [n. d.]. Open-source geographic information system. goo.gl/L6EnaL.
[49] QGroundControl. [n. d.]. MAVLink: Micro air vehicle communication protocol. goo.gl/fMPw0D.
[50] Qualcomm Inc. [n. d.]. Qualcomm flight UAV platform. goo.gl/FrQa5e.
[51] M. Quigley et al. 2009. ROS: An open-source Robot Operating System. *ICRA Workshop on Open Source Software* (2009).
[52] Raspberry PI foundation. [n. d.]. The Raspberry PI single-board computer. goo.gl/5FjUdx.
[53] Nathan Ratliff, Matt Zucker, J Andrew Bagnell, and Siddhartha Srinivasa. 2009. CHOMP: Gradient optimization techniques for efficient motion planning. In *Proceedings of ICRA*.
[54] Robin Ritz et al. 2012. Cooperative quadrocopter ball throwing and catching. In *Proceedings of IROS*.
[55] Inkyu Sa and Peter Corke. 2014. Vertical infrastructure inspection using a quadcopter and shared autonomy control. In *Field and Service Robotics*.
[56] A. Saeed et al. 2014. Up and away: A visually-controlled easy-to-deploy wireless UAV Cyber-Physical testbed. In *Proceedings of International Conference on Wireless and Mobile Computing, Networking and Communications*.
[57] Scientific American. [n. d.]. Five Epic Drone Flying Failures—and What the FAA Is Doing to Prevent Future Mishaps. goo.gl/tIXfHH.
[58] Technical University of Munich. [n. d.]. ROS package: AR.Drone camera-based autonomous navigation. goo.gl/5NSxDD.
[59] Texas Instruments. [n. d.]. AWR 1642 mmWave sensor. goo.gl/Pf7kyf.
[60] Texas Instruments. [n. d.]. Innovation Challenge. goo.gl/k6QwUy.
[61] UAV Forecast. [n. d.]. Wind statistics. goo.gl/h5EkKY.
[62] M. Valenti et al. 2007. The MIT Indoor Multi-Vehicle Flight Testbed. In *Proceedings of the International Conference on Robotics and Automation*.
[63] VICON Systems. [n. d.]. OptiTrack. goo.gl/A1mGwN.
[64] Volta Inc. [n. d.]. 4GMetry PixHawk companion computer. goo.gl/627gfa.
[65] John Wang and Edwin Olson. 2016. AprilTag 2: Efficient and robust fiducial detection.. In *Proceedings of IROS*.
[66] Xiaodong Zhang, Xiaoli Li, Kang Wang, and Yanjun Lu. 2014. A survey of modelling and identification of quadrotor robot. In *Abstract and Applied Analysis*.