DOI: https://doi.org/10.1145/3417084.3417088

THE FlyZone TESTBED ARCHITECTURE FOR AERIAL DRONE APPLICATIONS

Mikhail Afanasov *Politecnico di Milano, Italy*
Luca Mottola *Politecnico di Milano, Italy, RI.SE Sweden*

Illustration, istockphoto.com

Editor: Steven Ko

Aerial drones represent a new breed of mobile computing. Compared to mobile phones and connected cars that only opportunistically sense or communicate, aerial drones offer direct control over their movements. They can thus implement functionality that were previously beyond reach, such as collecting high-resolution imagery, exploring near-inaccessible areas, or inspecting remote areas to gather fine-grain environmental data.

The current practice of experimenting with autonomous aerial drone applications is, however, resembling the early times of computer programming, as it involves significant and time-consuming trial-and-error. The one difference is that drones crashing or failing potentially represent a physical threat to objects and persons. Moreover, gaining access to the target deployment for testing early implementations is and will be difficult, as regulations worldwide are increasingly stricter on the formal and technical guarantees that one has to provide.

Motivation. We have first-hand experience on the issues arising when experimenting with autonomous aerial drone applications. Representatives of an oil company eventually asked us to prototype a drone system to perform automatic visual inspections of oil tanks [11], using low-cost drones. These are hostile environments, as gases originating from chemical residuals abound.

Using drones is thus a viable alternative, but the necessary functionality is anything but trivial. Oil tanks are GPS-denied environments and cannot be instrumented beforehand. Autonomous navigation is to be achieved using visual or dead-reckoning techniques. The former tends to be more precise, but their performance is sensitive to the environment's visual features.

We started from an existing implementation of simultaneous localization and mapping (SLAM) for the AR.Drone 2.0 [5,18]. This was much more mature than many existing implementations of autonomous navigation functionality: it was extensively tested using real drones [5] and offered optimized parameters for the target drones. We created mock-ups of oil tanks 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.

The reason is qualitatively shown in Fig. 1 and Fig. 2: the parameters driving the recognition of visual features in the environment were tuned for objects of shape, color, and size different from the inside 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. After 2+ months of experimentation with no dedicated support, the system was still not even remotely working in an accurate way.

Challenge. Developers of aerial drone applications are confronted with similar issues in a range of diverse domains, including ambient intelligence [7] and search-and-rescue missions [9]. Existing techniques enabling autonomous behaviors rarely work out of the box, as they are typically tested only in simulation or require significant application- and/or hardware-specific customization. As a result, well-tested and reliable system implementations are largely lacking. Many drone applications may also benefit from multiple collaborating drones. Investigating distributed interactions further complicates matters.

Drone simulators exist (e.g., SITL [3]), which are, however, simplified compared to reality, unable to model application-specific sensors, and are often limited to waypoint GPS-driven navigation. Robot simulators 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.

Few attempts exist at providing system support for experimenting with drone applications [16] in a real-world setting, as most existing facilities are designed for different purposes, for example, to study low-level mechanical control using highly engineered drones backed by computing clusters and expensive motion capture systems. The objective is not to test application implementations, but to investigate fine-grained flight regulators that enable demonstrations, such as drones throwing and catching balls.

Because of these reasons, experimenting with drone applications tends to take place right in the target settings, if and when that is at all possible or allowed. 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 [17]. The ever-changing regulations on the use of civil drones compound the problem [6].

FlyZone. Developers of drone applications require a means to run high-level application functionality by emulating the features of the target deployment setting, using commercially available drones.

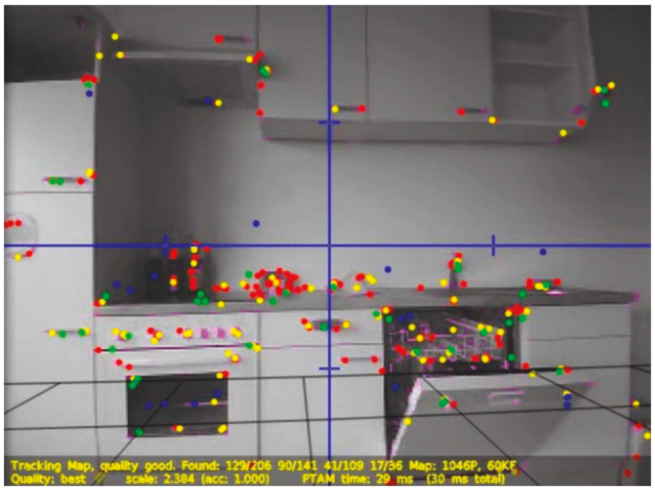
FlyZone fills this gap. It provides drone developers with three key features: i) an application-level API that allows them to write code as close as possible to the one to be deployed, ii) ways to emulate environment influences, such as wind or pressure gradients, and iii) the ability to specify safety constraints, which are automatically monitored to 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. The drone is safe, the objects nearby are as well, while developers will not need to collect the pieces of whatever they broke before being able to investigate the problem.

Achieving this functionality requires addressing conceptual as well as technical challenges. These include i) decoupling the testbed infrastructure from the application, to ease the transition from testbed to deployments; ii) realistically emulating the environment influence, and iii) tackling the drone localization problem in the testbed as opposed to robot localization in a target application. In the following, we provide a brief account on how we address these challenges.

Unlike simulators, with FlyZone, the experimentation occurs using the same application code and drone platforms to be deployed in the target setting. Our testbed design also facilitates replicating the testbed infrastructure at different sites in a fully customized fashion. To that end, we offer a well-defined set of procedures and scripts to automate this effort [2]. FlyZone is entirely built with off-the-shelf commercial hardware, which facilitates obtaining the equipment and reduces costs.

Three working installations of FlyZone currently exist, located at Politecnico di Milano (Italy), University of Virginia (US), and Warwick University (UK). They are actively used by researchers at either institution and are also instrumental to evaluate the performance of FlyZone. We demonstrate, for example, that we realistically emulate the environment influence with a positioning error bound by the size of the smallest drone we test. In the following, we also succinctly report on our experience using FlyZone for developing prototype drone applications that reached real deployments. Further details about the design, implementation, performance, and use of FlyZone are available [2].



```
Tracking Map, quality good. Found: 129/206 90/141 41/109 17/36 Map: 1046P, 60KF
Quality: best        scale: 2.384 (acc: 1.000)       PTAM time: 29 ms   (30 ms total)
```

FIGURE 1. The parameters of the existing implementation are optimized for a domestic environment [5].

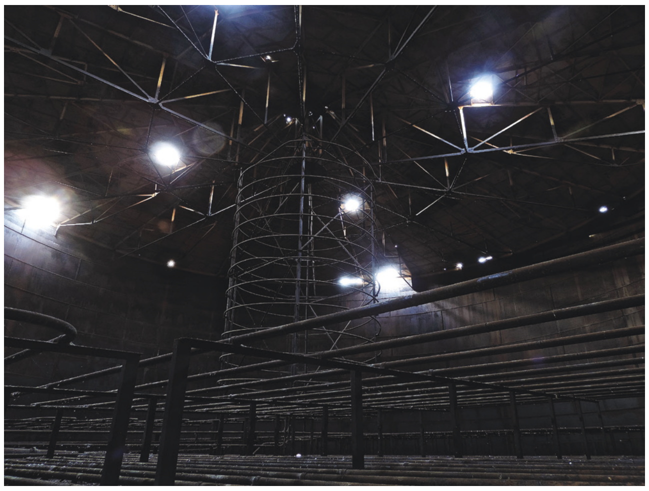


FIGURE 2. Oil tanks lack the visual features of Fig. 1, and require a new set of parameters to efficiently operate SLAM.

FlyZone IN A NUTSHELL

Three aspects concur to the design of FlyZone: its architecture design, how we emulate the environment influence, and how we perform drone localization in the testbed.

Architecture

The FlyZone architecture seeks to decouple the testbed infrastructure from the main application. Fig. 3 shows how we achieve these objectives in the case where the application runs on a Ground Control Station (GCS); minimal variations apply if the application is deployed on a companion computer aboard the drone.

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 [14] protocol though a UART interface. The controller also serves part of the localization system, as described next, so it is informed of the location.

The controller receives commands from and forwards telemetry data to an experiment script. This specifies the actions that developers are interested in; for example, to create given environment situations, what kind of environment influence that developers want to emulate, as well as the safety constraints for the experiment. The experiment script accesses FlyZone through an API currently available in Java, Python, and C++ [2].

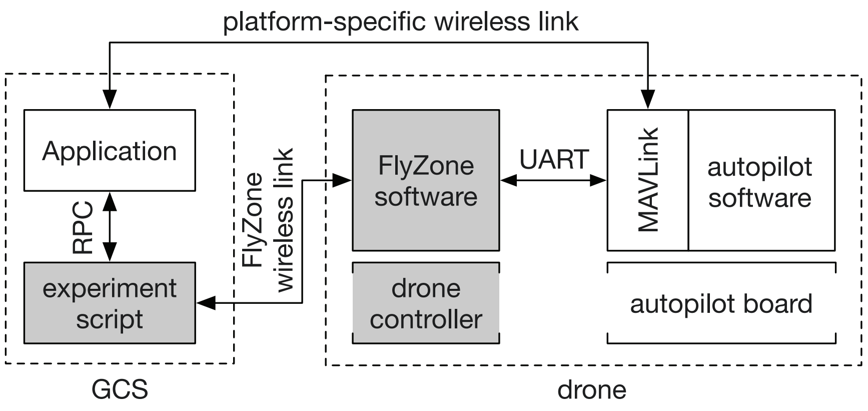


FIGURE 3. FlyZone architecture. Components in grey are FlyZone-specific.

Environment Emulation

Intuitively, to emulate the environment influence on a drone, we 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.

To do so, we make the drone controller issue commands to the autopilot to steer the drone as if it were subject to the corresponding forces [1]. Applications cannot distinguish these from actual environment influence, as both are detected as changes in the readings of navigation sensors. This functionality is enabled by a model of the physical drone dynamics that determines how the drone would move when subject to given external forces.

We rely on the vast literature on aircraft dynamics and adapt existing detailed models to FlyZone. Note that we need a model for general navigation, unlike task-specific models. 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 28 different drones, ranging from custom quad-, hexa-, and octocopters to commercial ones, such as DJI Spark, Mavic Pro, and the whole Phantom and Inspire series.

Model structure. Aerial drones are 6-degree of freedom (DOF) rigid bodies. We use the Newton-Euler method, where 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 p^n and v^n be the drone position and velocity vectors in navigation coordinates (denoted with n), and w^b be its angular rate in body coordinates (denoted with b). A drone's 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

$$p^n = v^n \qquad (1)$$

$$v^n = m^{-1} C^n{}_b F \qquad (2)$$

$$w^b = J^{-1} M \qquad (3)$$

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

A solution to eq. (1)-(3) represents the necessary and sufficient information to steer the drone as if it were subject to given external forces. We translate the solution to MAVLink commands that the drone controller issues to the autopilot. We describe next how to obtain the necessary drone-specific inputs and parameters.

Model inputs. We apply existing methods to compute the torque vector M in eq. (3) [4]. The force vector F in equation (2) is a combination of drag forces F_d, motor thrust F_m, and gravity F_g. We compute F_m based on the relation between input current and output thrust for brushless DC motors, which typically equip aerial drones. The gravity vector F_g is straightforward.

The remaining force vector representing drag forces F_d is a function of drag coefficients and of the velocity vector of external forces applied to the drone. Using FlyZone, the latter is input to a given experiment and represents the knob that 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 [13] to generate detailed three-dimensional wind maps, like in Fig. 4, or by synthetically creating wind patterns modeling specific situations, such as narrow passages or airflows around objects.

Parameter estimation. The unknown quantities to use the model are the drone mass, inertia, and drag coefficients. Estimating a drone mass is straightforward. We use a three-dimensional drone model in Blender combined with the BGE Advanced Physics Library and custom scripts that we develop to estimate the latter two. 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.

Throughout this process, some approximations are inevitable to keep the problem tractable. For example, we use 3D models as exemplified in Fig. 5, 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, rather than considering the specific propeller shape.

Localization

Drone localization must happen with accuracy, speed, and limited processing overhead. We are expecting FlyZone installations to be located indoors, so GPS is generally not applicable. Motion capture systems are expensive and laborious to install. Techniques in traditional robotics also generally target different requirements and settings. For example, the deployment of tags in a testbed is dense and uniform in a known environment, unlike the sparse and uneven deployment seen whenever visual tags are attached to objects to help ground robots roam unknown environments. Moreover, we aim at optimizing not only accuracy, but also processing speed to ensure FlyZone can promptly react to safety violations and to scale in the presence of multiple drones.

Based on these considerations, 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. 6, deployed on the ground at known positions, as shown in Fig. 7. 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, whereas the main application relies on its own separate localization system, for example, using GPS for outdoor operation or SLAM in GPS-denied environments [5]. We obtain altitude information from the autopilot software, based on the readings of ultrasound sensors, part of a drone's IMU.

We use Java to implement the entire localization pipeline, using OpenCV. 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 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 tags are generated and placed on the ground in ways that maximize both accuracy of detection and processing speed. We achieve this by formulating both tag generation and placement as a multi-objective optimization problem [2]. The scripts and solving tools are provided as part FlyZone, and can automatically generate a customized tag space depending on input information, such as camera accuracy and physical constraints of the space. This way, we facilitate replicating the testbed at places other than the existing installations.

FlyZone IN ACTION

To provide quantitative evidence, we discuss next the accuracy in emulating the environment influence. This integrates the functioning of the FlyZone architecture, the models used for environment emulation, and the localization technique. It is thus representative of the testbed performance as a whole. We also discuss our experience in using FlyZone to develop real-world drone applications.

Environment Emulation

Quantifying how realistically FlyZone emulates the environment influence is a challenge per se. The key problem is how to gain some ground truth.

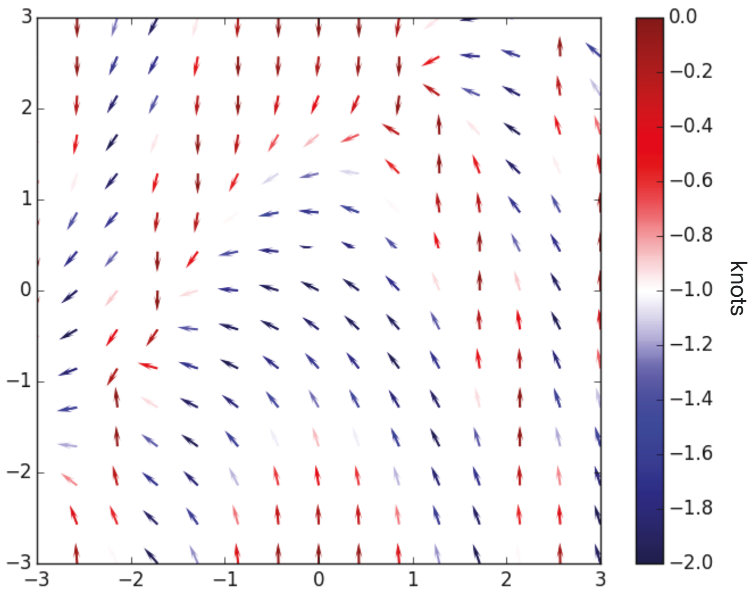


FIGURE 4. Wind map generated using QGis software.

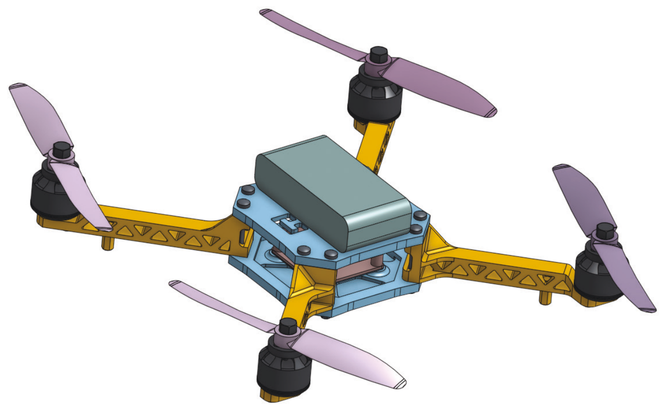


FIGURE 5. Quadcopter Blender model.

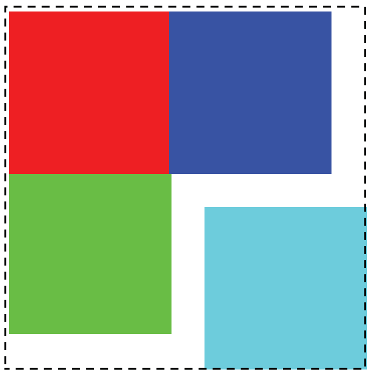


FIGURE 6. Visual tag.

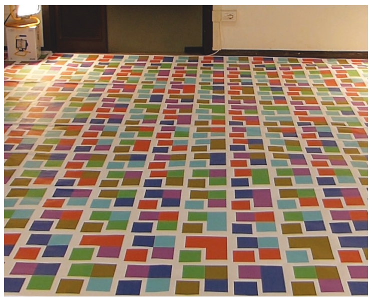


FIGURE 7. Tag space.

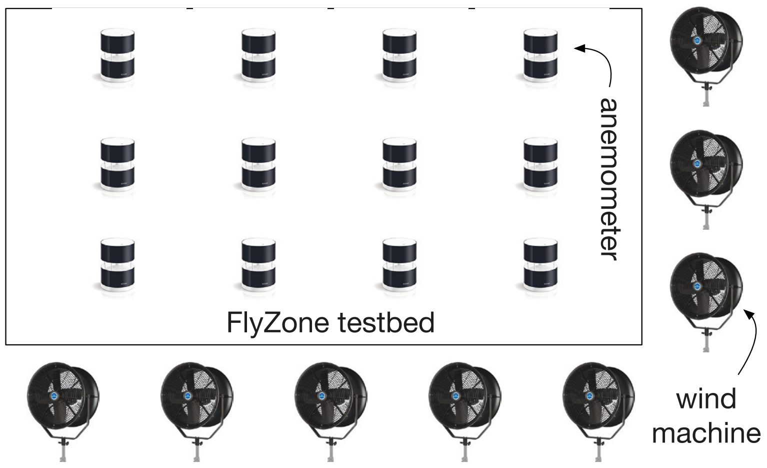


FIGURE 8. Evaluating realism in emulating environment influence: testbed setup.

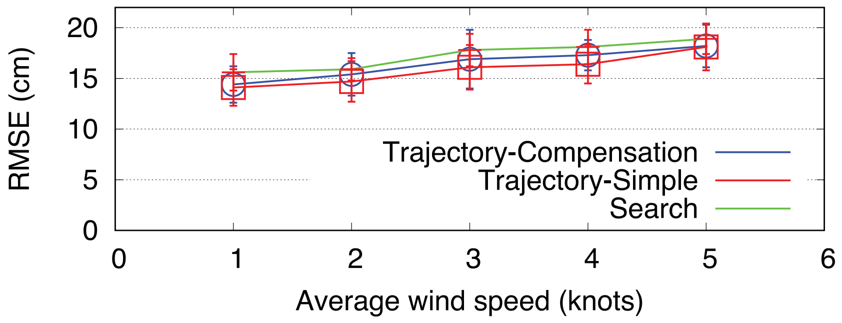


FIGURE 9. FlyZone performance in emulating the environment influence.

Setting. Fig. 8 graphically describes the setup we design to this end. We rent eight 17" wind machines of the type used in professional movie making. They 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 uniformly install 48 portable anemometers measuring flow speed and direction. We linearly interpolate their measurements to create a three-dimensional wind map, as in Fig. 4.

As a form of ground truth, we run test applications against the actual influence of the wind machines, using the same settings for creating the wind maps, but without the anemometers. For comparison, we input the windmaps to FlyZone to recreate the environment effect. We develop a single-drone application called *trajectory*, which directs the drone along regular three-dimensional paths. The application may run in two modes: in compensation mode, it tries to counteract the effect of wind machines to maintain the trajectory; in simple mode, it does nothing to that end. We also emulate a search-and-rescue application using five drones [10] and call it search.

We track the drones using an OptiTrack motion capture system [19]. 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. 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. We total 400+ hours of tests.

Results. Fig. 9 plots the RMSE results in drone positions; we obtain similar trends for orientation. The absolute RMSE are constantly lower than the physical dimensions of the smallest drone we use. The trajectory-simple case shows the lowest average RMSE because the application does nothing to counteract the environment influence, so the interplay between drone controller and application creates no discrepancies compared to reality. Conversely, search uses a complex application logic that constantly tries to correct the trajectory; this creates slight differences between the emulated behavior in the testbed and the real behavior one would expect in the field.

Experience

Oil tanks. The application outlined earlier 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 run 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 [8].

Search and rescue. We ran a student challenge comparing the use of FlyZone with the SITL [3] 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 companion computer and an ARVA radio receiver for finding people under snow [12]. The objective was to minimize search times. We recruited 20 MSc graduate students with multi-course expertise in software engineering and embedded systems. Half of the students used FlyZone, the other half used SITL.

The students started from a textbook description of a gradient descent algorithm [15]. They were also required to extend the system to multiple collaborating drones to reduce search times [10]. This functionality had to be developed from scratch. Development times were generally in favor of the groups using SITL, who invested about 33% fewer hours. Admittedly, 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. This site was unknown to the students until they turned in the final implementations. We again used digital anemometers to ensure comparable conditions.

All groups were using FlyZone but one showed better performance than the groups who used SITL, resulting in about 37% shorter search times. 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.

Training pilots. The FlyZone installation at Politecnico di Milano (Italy) is also helping a local piloting school train pilots working toward the official license for flying professionally.

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 that 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.

CONCLUSION

FlyZone is a testbed architecture 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. A custom visual localization technique enables this performance, while a lightweight testbed architecture that maximizes decoupling from the main application facilitates transitioning from testbed to real deployments. We are currently engaged in using FlyZone to test the performance of autonomous navigation algorithms, whose performance was hitherto only measured in simulation, against the emulated environment influence. We argue this is a fundamental step towards the concrete real-world use of this technology. ■

Acknowledgements

This research was partially supported by the EU commission with H2020 project 5G-DIVE (grant agreement 85988), by the Italian Ministry of Education, University, and Research through the cluster project "ICT Solutions to Support Logistics and Transport Processes (ITS)", and by the Swedish Innovation Agency through project "DePILOT", "DePILOT Reloaded", and "DepDrone".

Mikhail Afanasov has a PhD in Computer Science and is a Technical Lead at UBS. Prior to this, he was working as a Senior Developer at Credit Suisse and as a Research Fellow at Politecnico di Milano, Italy.

Luca Mottola is an Associate Professor at Politecnico di Milano, and a senior researcher at RISE SICS, Sweden. His interests focus on modern networked embedded systems. He has received numerous awards including the Google Faculty Award, the Cor Baayen Award, the ACM SigMobile Research Highlight, and Best Paper Awards at ACM MOBISYS and ACM/ IEEE IPSN.

REFERENCES

[1] E. Bregu, N. Casamassima, D. Cantoni, L. Mottola, and K. Whitehouse. Reactive control of autonomous drones. 2019. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services (MOBISYS)*.
[2] M. Afanasov, A. Djordjevic, F. Lui, and L. Mottola. Flyzone: A testbed for experimenting with aerial drone applications. 2019. In *Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services (MOBISYS).*
[3] Ardupilot. SITL simulator. goo.gl/PLfx1g.
[4] G. De Croon, K. De Clercq, R. Ruijsink, B. Remes, and C. De Wagter. Design, aerodynamics, and vision-based control of the delfly. 2009. *International Journal of Micro Air Vehicles*, 1(2).
[5] J. Engel, J. Sturm, and D. Cremers. 2012. Camera-based navigation of a low-cost quadrocopter. In *Intelligent Robots and Systems* (IROS).
[6] Federal Aviation Administration. Regulations for unmanned aircraft systems. goo.gl/cahhbk.
[7] D. Floreano and R. J. Wood. Science, technology and the future of small autonomous drones. (2015). *Nature*, 521(7553):460.
[8] MEETmeTONIGHT – Milano. Face research and researchers. www.meetmetonight.it.
[9] F. Michahelles, P. Matter, A. Schmidt, and B. Schiele. 2003. Applying wearable sensors to avalanche rescue. *Computer and Graphics*, 27(6).
[10] L. Mottola, M. Moretta, C. Ghezzi, and K. Whitehouse. 2014. Team-level programming of drone sensor networks. In *Proceedings of ACM SENSYS*.
[11] NTR Labs. UAVs for oil tank inspections. goo.gl/aHfPK8.
[12] Pieps. ARVA Transceivers. goo.gl/tPywra.
[13] QGis Project. Open-source geographic information system. goo.gl/L6EnaL.
[14] QGroundControl. MAVLink: Micro air vehicle communication protocol. goo.gl/fMPw0D.
[15] N. Ratliff, M. Zucker, J. A. Bagnell, and S. Srinivasa. 2009. Chomp: Gradient optimization techniques for efficient motion planning. In *Proceedings of ICRA*.
[16] A. Saeed et al. Up and away: A visually-controlled easy-to-deploy wireless UAV cyber-physical testbed. 2014. In *Proceedings of International Conference on Wireless and Mobile Computing, Networking and Communications.*
[17] Scientific American. Five epic drone flying failures—and what the FAA is doing to prevent future mishaps. goo.gl/tIXfHH.
[18] Technical University of Munich. ROS package: AR.Drone camera-based autonomous navigation. goo.gl/5NSxDD.
[19] VICON Systems. OptiTrack. goo.gl/A1mGwN.