Received 3 February 2023, accepted 28 February 2023, date of publication 6 March 2023, date of current version 13 March 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3253788

APPLIED RESEARCH

AirLoop: A Simulation Framework for Testing of UAV Services

JESSICA GIOVAGNOLA[1,2], **JUAN B. MORO MEGÍAS**[2], **MIGUEL MOLINA FERNÁNDEZ**[1,2], **MANUEL PEGALÁJAR CUÉLLAR**[3], **AND DIEGO P. MORALES SANTOS**[2]
[1]Infineon Technologies AG, 85579 Neubiberg, Germany
[2]Department of Electronic and Computer Technology, University of Granada, 18071 Granada, Spain
[3]Department of Computer Science and Artificial Intelligence, University of Granada, 18071 Granada, Spain

Corresponding author: Jessica Giovagnola (Jessica.Giovagnola@infineon.com)

This work was supported in part by the European Union (EU)-Funded Project Airborne Data Collection on Resilient System Architectures (ADACORSA) (www.adacorsa.eu) through the Electronic Components and Systems for European Leadership (ECSEL) Joint Undertaking (JU) support from the European Union's Horizon 2020 Research and Innovation Programme and Germany, The Netherlands, Austria, Romania, France, Sweden, Cyprus, Greece, Lithuania, Portugal, Italy, Finland, Turkey, under Agreement 876019; in part by the Spanish Ministerio de Economía y Competitividad under Project TED2021-129949A-I00; and in part by the Junta de Andalucía under Project P20_00265.

ABSTRACT Sensor fusion is a critical aspect in autonomous drone navigation as several tasks, such as object detection and self-pose estimation, require combining information from heterogeneous sources. The performance of these solutions depends on several factors, such as the characteristics of the sensors and the environment, as well as the computing platforms, which can heavily impact their accuracy and response time. Carrying out such performance evaluations through real flight tests can be a resource-demanding, time-consuming, and, at times, risky process, which is why researchers often rely on simulation environments for testing and validating sensor fusion algorithms. The simulation environment should provide photorealistic environmental features, as well as a comprehensive set of sensors, in order to allow to test the most extensive set of sensor fusion algorithms. This paper presents AirLoop, an AirSim-based flight simulator for Hardware-in-the-Loop and Software-in-the-Loop algorithm testing and validation. AirLoop extends the sensor setup provided by AirSim with an FMCW RADAR sensor simulation, which has been evaluated based on the Infineon Technologies BGT60TR13C RADAR. Furthermore, this work provides several Software-in-the-Loop (SITL) and Hardware-in-the-Loop (HITL) demonstrations, including interfacing with the Pixhawk 2 flight controller and an extensive evaluation of the communication of the engine with the NVIDIA Jetson Nano, which has been evaluated in various use cases, including dataset creation, object detection, Path Planning, and Simultaneous Localization and Mapping (SLAM).

INDEX TERMS Drone simulations, sensor fusion, Jetson Nano, RADAR simulation, HITL, SITL.

I. INTRODUCTION

In the past few years, Unmanned Aerial Vehicles (UAVs), commonly known as drones, have received growing attention from different industries due to their wide spectrum of applications, ranging from agriculture to last-mile delivery and surveillance [1]. However, creating drone-aided services needs the design, implementation, and testing of navigation algorithms.

Navigation algorithms combine information from different sources to serve various tasks such as environmental perception, obstacle avoidance, trajectory planning and tracking, localization, and mapping [2]. Such systems can be deployed on onboard embedded computing platforms or on base-station computers that communicate with the drones. Therefore, the developed algorithms need to be evaluated in real-time, in terms of both accuracy and response time and the

The associate editor coordinating the review of this manuscript and approving it for publication was Julien Le Kernec.

performance of the computing platforms needs to be tested according to their computational capacity and latency.

However, the testing and validation on real flight scenarios can be a costly, time-consuming, and rather a risky process [3]. First, it might be necessary to assemble a custom drone to host the computing platforms, the sensor setup, and power supplies. Such a task usually requires high cost and deep previous knowledge of the hardware specifications for each specific application.

Moreover, autonomous drone flights require certified pilots and the availability of adequate locations where it is allowed to perform beyond-visual-line-of-sight operations. Furthermore, possible failures of the tested system might harm public safety or can lead to the destruction of the drone itself.

The performance of the algorithms under test is subject to a number of factors that cannot be controlled by the user in real life scenarios, such as sensor failure and environmental conditions.

Therefore, we can state that efficient testing and validation of sensor fusion algorithms based on real drone flights requires addressing a number of open problems beyond the scope of the evaluation of the deployed systems.

Consequently, drone flight simulation frameworks can be a valuable alternative that allows the evaluation of sensor fusion pipelines in a repeatable and controllable way with the sole need of standard hardware components (e.g., personal laptop), without the need for physical system prototypes [4]. Such simulators should provide a sufficiently detailed model of the drone together with models of a suitable set of sensors ensuring proper environmental perception and self-state estimation [5]. Furthermore, the drone model and the simulated sensors should interact with diverse environments to emulate different use case scenarios. Since the developed sensor fusion algorithms might run on dedicated hardware, it should be possible to interface the simulator with external computing platforms and to perform Hardware-in-the-Loop (HITL) and Software-in-the-Loop (SITL) simulations.

In this paper, we present AirLoop: a UAV simulation framework based on the well-known platform AirSim, which can easily run on any sufficiently powerful machine with either Windows, Linux, or the Robot Operating System (ROS). The simulator provides a quadcopter model that can receive actuator signals and interact with different photo-realistic environmental models including city parks, forestry, and urban environments.

The functionalities of the simulator are demonstrated with laptop-based simulations and in context of the validation of a basic drone architecture, where a Pixhawk 2 serves as a flight controller and an NVIDIA Jetson Nano executes different tasks such as object detection, data gathering, and Simultaneous Localization and Mapping (SLAM).

The available sensor setup consists of both proprioceptive sensors (IMU, GPS) and exteroceptive sensors (LiDAR, monocular camera, depth camera, RADAR). However, a state-of-the-art review showed that none of the off-the-shelf RADAR sensor simulations that could be interfaced with AirSim could re-create the raw sensor data with sufficient accuracy. Therefore, AirLoop provides a novel RADAR sensor simulation implementation that can re-create range-doppler images and Frequency Modulated Continuous Wave (FMCW) RADAR raw data starting from visual data.

The rest of the paper is structured as follows. Section II provides a literature review of the available drone simulation frameworks (Subsection II-A) and the available RADAR simulators (Subsection II-B). Section III describes the hardware setup leveraged for the simulator implementation and the experiments. Section IV discusses the implementation of the Simulation Framework. In more detail, Subsection IV-A lists the requirements, Subsection IV-B describes the system architecture, Subsection IV-C discusses the communication protocol between the engine and the NVIDIA Jetson Nano, and Subsection IV-D tackles the implementation of a RADAR sensor simulator integrated into the simulation environment. Section V provides instructions about the installation process (Subsection V-A) and reviews the experiments carried out with the simulator, including dataset creation (Subsection V-B), object detection (Subsection V-C), RADAR model evaluation (SubsectionV-D) and two separate flight demonstrations (Subsection V-E). Finally, section VI is dedicated to the conclusion.

II. STATE OF THE ART

This section provides an overview of the currently available UAV flight simulators. In more detail, Subsection II-A describes and compares the main general simulation frameworks for drone flight simulation, while Subsection II-B is specifically focused on the RADAR sensor simulation and its possible integration with the simulation system.

A. DRONE FLIGHT SIMULATION ENVIRONMENTS

A flight simulation framework should contain a model of the drone dynamics, the sensors, and the environment. Each simulated component can be modeled with a different level of detail, reaching a trade-off between computational speed, physical accuracy, and photorealism. In this dissertation, we discuss only the simulation environments that, after a preliminary state-of-the-art review, have been tested first-hand by us, as they could better fit the scopes mentioned in Section I. A more comprehensive overview of the currently available UAV simulation frameworks is available at [6].

The simulators have been tested either on Windows or Linux using a personal laptop equipped with Intel(R) Core(TM) i5-8250U processor, 8GB Random Access Memory (RAM), 128GB Solid State Drive (SSD) + 1TB Hard Drive Disk (HDD), and a GeForce MX 130 2GB Video RAM. The HDD hosts a Linux partition, while Windows runs on the processor.

AirSim [7] is an open-source and cross-platform simulation environment from both aerial and ground vehicles created by Microsoft. It can be easily installed on different operating systems: we tested AirSim's functioning on

Windows 10. AirSim is developed as a plugin for the rendering platform Unreal Engine [8], meaning that it can be interfaced with any of its environments. Unreal Engine provides a diversified set of pre-implemented environments that the user can customize through a Graphical User Interface (GUI). A Python Application Programming Interface (API) allows the drone to receive commands and enables data retrieval for future elaboration using external software tools.

As for the drone dynamics model, AirSim utilizes a custom kinematic physics engine called FastPhysics, where the multirotor is modeled as a rigid body equipped with an arbitrary number of actuators that generate force and torque. FastPhysics expresses the state of the multi-rotor through the following parameters: position, orientation (quaternions), linear and angular velocity, linear and angular acceleration.

AirSim is characterized by outstanding photorealism due to the leverage of advanced rendering techniques and detailed environmental designs. For this reason, a rather powerful machine is necessary to run the simulations smoothly and to guarantee a reasonable frame rate. Furthermore, a wide set of sensors is available, including visual sensors (e.g., monocular camera, depth camera, image segmentation), barometer, IMU (here consisting of an accelerometer and a gyroscope), magnetometer, and GPS. The sensor models are implemented as C++ libraries and their parameters and characteristics can be adjusted (e.g., Field of View and resolution for the camera). However, the addition of a new sensor model from scratch is not contemplated in the conception of the simulator, which makes the modification of the sensor setup quite complicated as it has to be done through the direct modification of the involved libraries. However, AirSim comes with a wide set of pre-implemented sensors allowing it to cover a wide set of combinations. However, sufficiently accurate FMCW RADAR simulation is not included in the standard version of the simulator frameworks.

In addition, AirSim offers the possibility to easily obtain pre-labeled data from the simulated exteroceptive sensors, which is a handy feature to have ground-truth labels for training and testing supervised learning algorithms.

AirLearning [9] is an extension of AirSim intended for the deployment and testing of Reinforcement Learning policies in the context of autonomous drone navigation through Hardware-in-the-Loop (HITL) simulations.

Like Airsim, Flightmare [10] is another engine-based simulator using the Unity rendering engine [11]. Flight is made up of several software components that are decoupled from each other, namely:

- *Flightmare library*, containing the drone dynamics and the sensors;
- *Flightmare Rendering Engine*, providing photo-realistic Unity environments. The interface with the quadrotor dynamics is implemented using the messaging library ZeroMQ;
- *Reinforcement Learning Algorithms and Examples*, a Python wrapper for deep RL algorithms and examples;
- *Robot Operating System (ROS) Wrapper* for Flightmare Library.

Flightmare provides four different environments: *Industrial Warehouse*, *Garage* and *Nature Forest* and a set of exteroceptive sensors, i.e., Camera (RGB Camera, Depth Camera, Semantic Segmentation and Optical Flow), Collision Detector and Optical Flow. To the current state, it is not possible to add or modify the environments nor the sensors [12].

The installation can be carried out via the *pip* library or the Robot Operating system.

Although the environmental rendering is less resource-demanding than Airsim, the execution of Flightmare is considerably more CPU-intensive: the authors recommend a machine with a 12-core CPU to perform a smooth simulation.

A rather big family of simulators is based on the Gazebo framework [13], [14]. Gazebo allows to simulate the flight of individual or multiple drones within user-defined indoor or outdoor environments, similarly to AirSim. Multiple APIs are supported to interface the drone model with external software tools.

As for the physical modeling, it is possible to choose between different drone dynamic models and environmental models, allowing to simulate with different levels of abstraction. Despite accomplishing a fair level of graphical rendering, Gazebo does not outperform AirSim in photorealism.

A broad set of sensors is available, as in AirSim. FMCW RADAR simulation is not available. The drone frame, sensors, and actuators are modeled as a set of joints and links in a *.urdf* format file. Therefore, adding or removing sensors is still possible, even if with a higher effort in comparison with AirSim. Gazebo supports multiple platforms, such as Windows, Linux, Mac, iOS, and ROS.

The last simulator framework we tested is called RotorS [15]: a Micro Aerial Vehicle (MAV) simulation framework based on ROS and Gazebo. ROS comes up with a comprehensive set of open-source libraries and tools to support the development of robotics applications. The main idea of ROS is to provide a basic framework with basic and core functionalities where different developers can implement solutions concerning different components of the robotic system. Each executable software model is called *node* and can communicate with other nodes within the system through a custom messaging protocol based on communication channels called *topics*. The critical advantage of ROS lies in the possibility of dividing the code into single reusable blocks that can be easily interfaced together via a single messaging tool [16].

The RotorS simulator consists of a ROS package equipped with plug-and-play libraries implementing various functionalities such as controllers, basic environments called worlds, interfacing with joystick controllers, and example launch files. The libraries are written in C++, but ROS nodes can be written in Python, too.

When executed, the ROS nodes contained in RotorS can send messages to Gazebo, enabling system visualization.

TABLE 1. Comparison and baseline selection of the State-of-the-Art simulators.

Features	AirSim	AirLearning	Flightmare	RotorS	Asimovo
Simulation velocity	✓	✓	✓	✓	✓
Sensor availability	✓	≅	✗	✓	✗
Extensibility	≅	≅	≅	✓	✓
Gathering data velocity	≅	≅	✓	✓	✓
Labelling	✓	✓	✗	✗	✗
Photorealism	✓	✓	≅	✗	✗

The set of sensors included in RotorS comprises a camera, a depth camera, and an IMU.

Unlike AirSim, RotorS and the other Gazebo-based simulators do not provide pre-labeled data. However, this functionality could be achieved using the *projection* [17].

We have tested and evaluated the performance of the simulators above based on the parameters displayed in Table 1.

Based on the evaluation stated above, we decided to utilize AirSim as a base to build up our own simulation framework.

B. RADAR SIMULATION

We now describe the implementation of a simulated Frequency Modulated Continuous Wave (FMCW) RADAR sensor integrated into the proposed simulation framework. To the current state, no RADAR simulation model that can deliver synthetic raw data is available in AirSim. However, a few RADAR simulation environments are available in the literature.

ViRA is a real-time FMCW RADAR simulation Framework based on Unity. It comes up as a stand-alone Unity project with a simple scene, but it can also be inserted into an existing Unity project. The engine acts as a client that connects to ROS or a TCP/IP server to fetch data. The main configurable parameters are the Field of View (FOV), the number of chirps per frame, the number of receiver antennas, the bandwidth, the lower frequeny, the sample frequency, and the radiation pattern mask.

The radiation pattern mask is a property of the emitting antenna that defines the variation of the radiated power as a function of the departure direction of the electromagnetic wave. Such information can usually be found in the datasheet.

The input of the ViRA signal generation pipeline is a depth image. Every pixel of the image is processed to obtain a signal contribution. The signal response of the RADAR is obtained by summing the individual contributions. In more detail, the signal generation models:

- Wave penetration effects, based on the information extracted from occluded objects
- Multiple antenna receivers in a uniform linear array fashion. The linear configuration enables the extraction of the azimuth and the elevation angle
- Reflection coefficient, to better recreate the reflection of RADAR waves, which depends on the objects and the material's electric properties. A roughness and a normal value are associated with every pixel of the detected object.
- Radiation pattern modeling, i.e., a 2D gray-scale image is applied to the original picture to model the radiation pattern mask.

The framework was validated by configuring the system corresponding to the BGT60TR13C Infineon FMCW RADAR [18] and comparing the simulated results with the real measurements.

In [19], a millimeter wave RADAR implementation for AirSim is proposed. However, its output does not consist of simulated raw data but of a color-coded point cloud obtained from the depth image. The processing pipeline requires the following inputs: a depth image, a segmentation image, and a surface normal image. A depth image contains both visual and depth information by assigning to each pixel a numerical value identifying the distance between the sensor and the represented object. On the other hand, in a segmentation image each pixel holds a value between 0 and 255 depending on the mesh each object has. Therefore, assuming that a single object corresponds to a single mesh, choosing a specific value allows one to detect a single object in the environment. Finally, a sufrace normal image is a gray-scale image containing information about the normality of the surfaces with respect to the camera view.

Currently, an FMCW RADAR sensor model that can give as an output simulated raw RADAR data is not available in AirSim. A real-time RADAR simulator based on Unreal Engine for the Automotive industry called Advanced Millimeter Wave RADAR has been developed by the company OTSL but has yet to be released [20].

Other RADAR simulation options are available as in [21]. However, they are only based on MATLAB scripts without the possibility of being easily integrated into a game engine-based simulator.

III. HARDWARE SETUP

The experiments executed in this paper refer to the validation of a basic drone flight architecture with Hardware-in-the-Loop (HITL) and Software-in-the-Loop (SITL) simulations with the support of a personal laptop.

As mentioned in Section II, the state-of-the-art review was supported by a Windows laptop with a Linux partition equipped with Intel(R) Core(TM) i5-8250U processor, 8GB Random Access Memory (RAM), 128GB Solid State Drive (SSD) + 1TB Hard Drive Disk (HDD), and a GeForce MX 130 2GB Video RAM.

However, a more powerful laptop has been chosen to carry out the final experiments of the proposed simulator to improve the performance in terms of frame rate and rendering smoothness. Thus, the experiments described in Section V have been performed on an ASUS TUF GAMING F15 with 6 cores, 16GB RAM, and a GeForce RTX 3060 6GB Video RAM works well for our needs.

The basic flight architecture we consider in this work comprises a flight controller and an embedded computing platform that is in charge of carrying out sensor fusion tasks. In more detail, a well-known commercial-off-the-shelf flight controller has been selected, i.e., a PH ORANGE SET Pixhawk Orange Cube + Pixhawk 2.1. The flight controller offers good connectivity options for additional peripherals, a triple redundancy IMU system, and all the features needed to fly any remotely controlled aircraft, helicopter, or multirotor. It controls the power delivered by the motors (propellers speed) and performs the low-level control of the vehicle. Since one of the experiments requires the flight controller to be initialized with the help of an actual GPS sensor, the Here + GPS module was leveraged for this purpose.

An NVIDIA Jetson Nano 4GB serves as an onboard computer on which several sensor fusion algorithms are carried out. Its reduced dimensions and its relatively high computational power makes it ideal for developing Artificial Intelligence applications on edge.

IV. GENERAL SIMULATOR FRAMEWORK

In this section, we describe the general framework of our simulator. We first state the user and technical requirements in Subsection IV-A. Then, we describe the general architecture of the system in Subsection IV-B, the communication protocol with the NVIDIA Jetson Nano (Subsection IV-C). Finally, subsection IV-D described the proposed RADAR simulation.

A. REQUIREMENTS

The simulator shall provide an effective tool to benchmark sensor fusion algorithms deployed on a laptop or edge devices to carry out autonomous navigation tasks. This means that two main functions have to be accomplished. First, the simulator should enable the creation of synthetic labeled datasets. In other words, multi-sensor data should be retrieved from the simulation environment and stored to be processed offline. Second, commands from other devices, such as the flight controller and the NVIDIA Jetson Nano, should be sent to the simulator and interact online with the drone model and the environment.

Given the problem statement, we formulated some user functional requirements and technical requirements the simulator should accomplish to address the problem correctly.

The user functional requirements are stated below:

- High fidelity representation in order to reduce the gap between real and simulated data.
- Graphical user interface displaying the drone's state and the surrounding environment.
- Integration of a quadcopter and possibility to send commands manually (through the keyboard or a controller) or through dedicated scripts.
- Availability of multiple maps reflecting the reference use cases, i.e., last-mile delivery and last-mile delivery in urban environments.

From the user functional requirements stated above, we extrapolated the following technical requirements:

- Remote connection via Local Area Network (LAN) to allow external clients to retrieve state information from the simulator.
- The high-end laptop should be able to train AI algorithms that run on the onboard computer in real-time.
- The data retrieved by the simulator should be either stored offline on the high-end computer to serve the creation of a dataset (from now on, we will refer to this use case as the *offline version*) or retrieved and processed by external software or hardware clients (we will refer to these applications as the *online version*).
- The simulator has to provide semantic information about the surrounding environment to guarantee ground-truth labeling of the acquired data.
- The simulated drone should respond to the changes in the environment with sufficient speed. Although the simulation clock speed can be slowed down, the FPS rate should be high enough.
- Availability of the following sensors: camera, depth camera, LiDAR, IMU, GPS, and FMCW RADAR.

B. SYSTEM ARCHITECTURE

In this subsection, we explain the system architecture. The simulation system can be subdivided into three components:

- Graphical engine or simulator, AirSim in this case
- System Under Test (SUT), which extracts the state information from the engine, i.e., sensor data
- Client Under Test (CUT), which executes the algorithms in real-time.

As mentioned in Subsection IV-A, we distinguish between two versions of the system, namely *offline version*, in charge of retrieving and storing data, and *online version*, capable of carrying out real-time demonstrations.

1) OFFLINE VERSION

The main goal of the *offline version* consists of creating a dataset containing raw data from different sensors. A schema of this system configuration is available in Fig. 1. As it is possible to notice, no CUT is involved.

The dataset is stored in a commonly used state-of-the-art format employed in the ASL dataset [22]. The dataset comprises a set of folders containing the raw data files as *.csv* files and a *.yaml* file providing the sensor calibration information. Our simulated data contains ground truth semantic information, too. This type of information is stored in a *.xml* file, similarly to the Pascal Visual Object Classes (VOC)

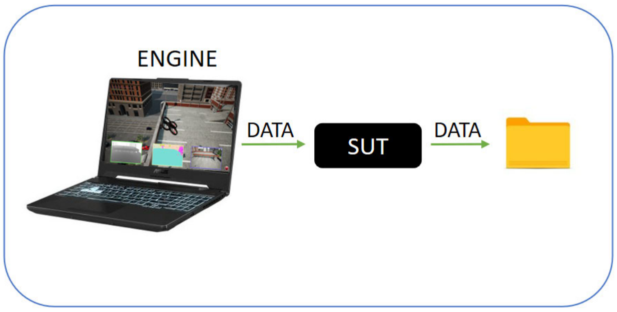


FIGURE 1. Offline version schema. SUT receives data from the simulator.

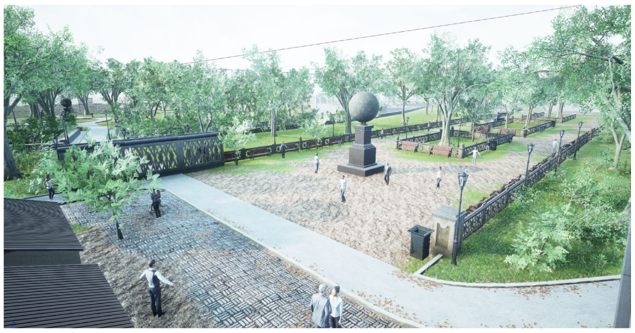


FIGURE 2. Scene created to record data based on the *City Park Environment*.

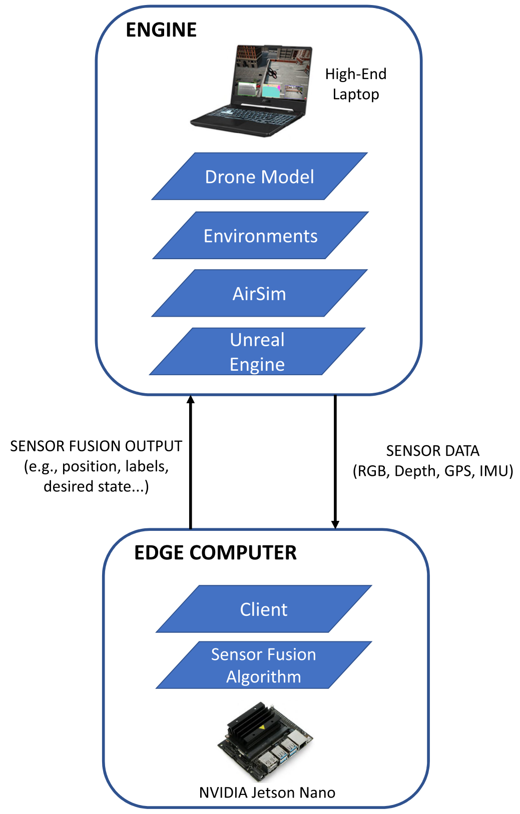


FIGURE 3. System architecture - online version.

dataset [23], a popular format for image datasets used for object recognition tasks.

A custom environment has been created based on an existing environment provided by Unreal Engine, called *City Park Environment* as displayed in Fig. 2. The environment has been enriched with different features, such as buildings, trees, and people.

The data acquisition task is done through a Python script running on the laptop. The script takes as input the destination directory where the data will be stored and the number of desired frames. The sensor acquisition task has been optimized in terms of acquisition frequency and synchronization with the use of six processes. The sensor information is fetched from five processes, while the sixth process controls the drone by describing a preset trajectory. The five processes, one per sensor, are mutually independent and synchronized to align with the slowest sensor before proceeding with the acquisition of the next frame.

2) ONLINE VERSION

In the *online version*, the data is gathered and processed in real-time to adapt the drone model's behavior to the environmental conditions. Following the nomenclature adopted in [9], we can subdivide the overall processing latency into three contributions:

- t_1: The SUT latency to extract the state information from the simulator.
- t_2: The time taken by the algorithm in the CUT to process the input data.
- t_3: The duration of the control action. The desired state of the CUT is converted in a low-level action with the AirSim flight controller API. The duration is determined by a specific function parameter or the necessary time to perform the action, such as moving to a reference (x, y, z) position with a certain speed.

The time contribution t_1 strongly depends on the computing platform as it is a CPU-consuming task. The hardware architecture was chosen in order to minimize t_1.

Two different variants of the *online version* have been implemented and compared to maximize the FPS. The system architecture is schematized in Fig. 3.

The SUT and CUT can run on the same or different machines. This creates different variants of the software architecture, which are discussed in the following paragraphs.

a: SUT AND CUT IN THE SAME MACHINE

In this schema, which is displayed in Fig. 4, the onboard computer is in charge of extracting the sensor data and running the algorithms.

The data gathering and the execution of the algorithm are based on a multithreaded script in which four threads implement the data gathering (SUT), and a fifth thread implements the CUT performing the tested algorithm. The multithreading approach had been privileged over the multiprocessing approach as the only shared variable consists of a

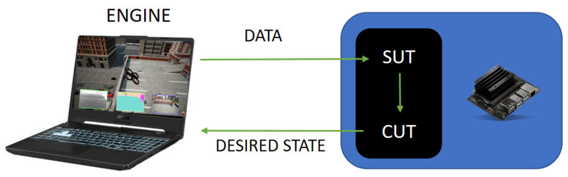


FIGURE 4. SUT and CUT running on the same machine.

list where sensor data is stored by the threads in charge of the data-gathering and retrieved by the one in charge of the sensor fusion. Each data-gathering thread reads the sensor data and stores it in a pre-defined position in the list. The sensor fusion is executed sequentially and returns as an output the desired state, which is, in turn, sent back to the simulator.

b: SUT AND CUT IN DIFFERENT MACHINES

This version, schematized in Fig. 5, discharges the onboard computer of the data extraction task, which is carried out by the high-end laptop. Therefore, the SUT retrieves data from the simulator and sends it to the CUT. After the CUT has elaborated the data, the SUT receives the CUT response and applies the control action to the drone model. In this configuration, the time contribution t_1 can be further subdivided into two parts:

$$t_1 = t_{ENGINE \to SUT} + t_{SUT \to CUT} \tag{1}$$

Since the SUT and the engine are deployed on the same machine, the time contribution $t_{ENGINE \to SUT}$ is reduced compared to the version described above.

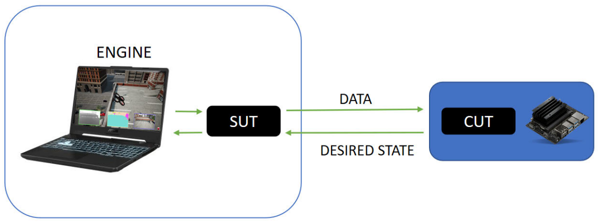


FIGURE 5. SUT and CUT running on different machines.

The pipeline is divided into two scripts, as the SUT and CUT run on different machines. The script implementing the SUT comprises one thread per sensor in charge of the data gathering, while the last thread retrieves the desired state information from the CUT and applies it to the simulator.

The thread in charge of retrieving the GPS data is taken as a reference to set a timestamp.

The CUT is implemented in a single sequential function that receives data, runs the tested algorithm, and sends the data back to the SUT.

C. COMMUNICATION WITH NVIDIA JETSON NANO

In order to enable communication between the simulation environment and the NVIDIA Jetson Nano, a TCP-based communication protocol was implemented. TCP is a connection-oriented protocol that guarantees data delivery in a FIFO fashion, i.e., the data is received in the same order it was sent. Therefore, TCP is a reliable protocol thanks to its flow control, congestion control, and the possibility of retransmission of lost packets [24].

The high-end laptop and the onboard computer are part of a LAN and are physically connected via an Ethernet cable. Using a LAN is meant to reduce the latency and fluctuations that are more likely to happen when using a wireless network.

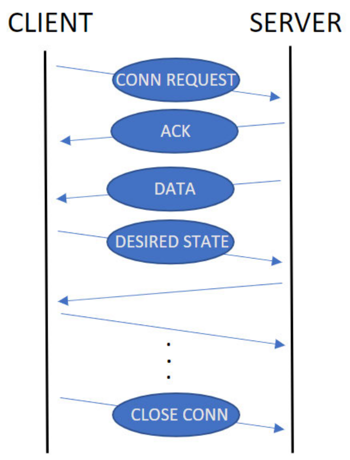


FIGURE 6. Communication protocol on TCP sockets - schema.

In the schema reported in Fig. 4, the engine and the onboard computer are connected through the AirSim API, involving Remote Procedure Calls (RCPs) over the TCP/IP network. The RCP technique aims to build a distributed system that allows a machine to call a subroutine running on another machine without the necessity to know that it is remote [25]. The API uses a messaging library called *MessagePack-RCP* that enables object serialization. At the start-up of the simulator, a default port is opened, which can be changed within the simulation settings and stored in a *.json file*.

This settings file contains the fundamental configuration parameters loaded when launching the simulation, such as the sensor configurations, the number of vehicles, and the physics engine.

On the other hand, when SUT and CUT are running on different machines, the two platforms communicate via socket. Fig. 6 displays a graphical representation of the protocol. First, the server binds to and listens to a port. Then, the client attempts to connect with the server and establishes a connection. The server can start sending sensor data and retrieving the desired state, and vice versa for the client. Finally, after the desired number of frames have been recorded, both client and server cut the connection.

The SUT and the engine are connected with RCPs, while SUT and CUT are connected via TCP sockets. Since the sensor data do not always have a fixed length (e.g., point clouds from LiDAR data can have different dimensions depending on the detection), the exchanged messages report the length of the messages in bytes in a header.

D. SIMULATED RADAR

We now describe the implementation of an FMCW RADAR sensor model integrated into the framework. The simulated sensor generates raw data instead of range-velocity 2D plots. This allows us to simulate the actual hardware; our model can be parametrized according to high-level RADAR specifications. Furthermore, the raw data can be leveraged to conduct research on signal processing techniques.

The state-of-the-art review discussed in Section II showed that ViRa is the only FMCW simulation based on a game engine currently available in the scientific literature. Therefore, our simulation is based on an adaption of its principles and equations to AirSim and its Python API. The input data for the RADAR simulator are based on the image set presented in [19].

The RADAR simulation pipeline is schematized in Fig. 7. The generation and processing of raw data lead to a significant latency, which would excessively slow down the data acquisition if carried out online. Consequently, the relevant input data, i.e., the timestamped normal and depth images, are post-processed right after the data acquisition.

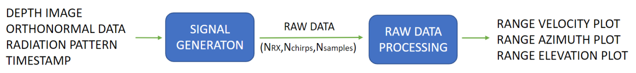


FIGURE 7. Simulated RADAR pipeline schema.

The pipeline can be subdivided into two main functional blocks, each implemented in a separate Python script. The first functional block is in charge of outputting simulated RADAR signals, while the second block extracts the range, velocity, and angle information is extracted through the preprocessing method proposed in [26], which returns Doppler images as the one shown in Fig. 8.

1) RADAR SIGNAL GENERATION

This paragraph describes the procedure to produce synthetic RADAR raw data. The inputs to the RADAR signal generation pipeline are:

- *Stereo image*, i.e., a picture containing both visual and depth information based on stereo-vision principles
- *Orthonormal image*, containing information about the orthogonal surfaces with respect to the objects represented in the simulation
- *Segmentation image*, a binary image whose pixels contain the value 1 if they contain information related to a relevant target (we selected trees, people, houses), 0 otherwise. It will act as a mask by being multiplied by the stereo image and the orthonormal image.
- *Radiation pattern*, i.e., a 2D gray-scale picture. We are using the same radiation pattern provided by ViRA (Fig. 9).
- *Timestamp*.

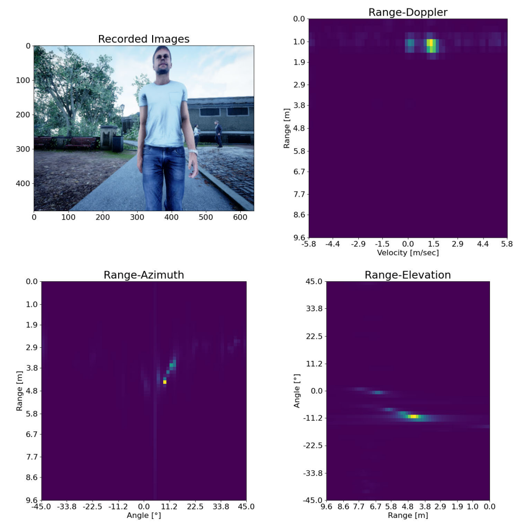


FIGURE 8. Simulated RADAR features, person with positive velocity detected during the flight.

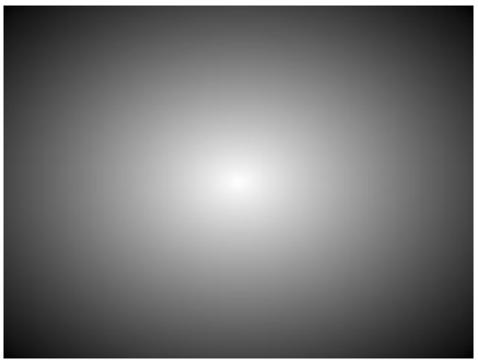


FIGURE 9. Radiation pattern mask in the RADAR simulator [27].

Furthermore, the simulation model requires as inputs a set of parameters consisting of the ones required in ViRa, with the addition of the chirp duration time T_c. These parameters influence the RADAR maximum range, velocity, and resolution. The parameters we used to enable mid-range object detection, consistently with our use case scenarios, are shown in Table 2. Just like ViRa, we take as a reference the BGT60TR13C Infineon FMCW RADAR.

TABLE 2. Simulated RADAR sensor configuration parameters.

Symbol	Parameter	Value
FOV	Field of View	90°
N_{chirps}	N° chirps per frame	64
$N_{samples}$	N° samples per chip	64
N_{RX}	Number of receiver antennas	3
B	Bandwidth	$0.4GH_z$
T_c	Chirp Duration	$390.975\mu s$
f_c	Lower Frequency	$60.8GH_z$

The technical characteristics of the simulated RADAR we obtained are shown in Table 3.

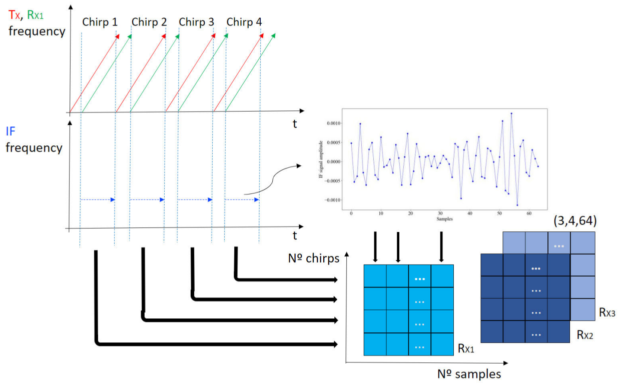


FIGURE 10. RADAR raw data structure example.

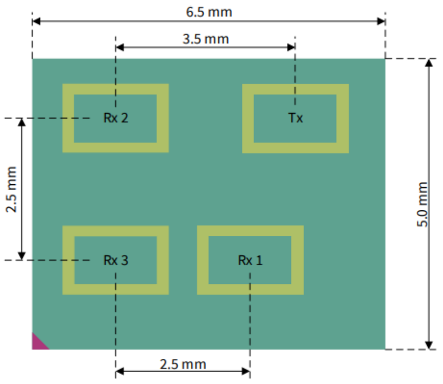


FIGURE 11. BGT60TR13C FMCW RADAR layout [18].

TABLE 3. Simulated RADAR technical characteristics.

Parameter	Value
v_{max}	$3.15m/sec$
v_{res}	$0.1m/sec$
d_{res}	$0.38m$
d_{max}	$12m$

The raw data consists of a matrix of dimensions $(N_{RX}, N_{chirps}, N_{samples})$ containing the voltage amplitude if the IF signal low filtered and ADC sampled. Fig. 10 shows the data structure of a frame of dimensions (3, 4, 64). Each matrix contains information from one receiver antenna, where each row represents a chirp and each column is a sample of the chirp.

A signal contribution is computed for each pixel of the input stereo image, then each value of the signal amplitude is given by the sum of the individual pixel contribution. Due to the high complexity of the operations ($\sim 10^9$ for a 320×240 pixels image), the operation was parallelized using GPU features with the *Numba* Python library [28], shrinking the computation time from tens of seconds to $\sim$ 1 second.

We will review the fundamental RADAR FMCW expressions as in [29] to better understand the signal generation pipeline. The chirp signal can be described with the following equation:

$$s_T(t) = Ae^{j(2\pi f_c t + \pi S t^2)} + A^* e^{-j(2\pi f_c t + \pi S t^2)}, \quad 0 < t < T_c \tag{2}$$

where $(.)^*$ denotes the complex conjugate. Considering a single object at a distance d moving at a velocity v, the received signal is a scaled and time-delayed copy of $s_T(t)$ where TOF is $\tau = \frac{2(d+vt)}{c}$, with t the Δ of time considered.

$$s_R(t) = Be^{j(2\pi f_c (t-\tau) + \pi S (t-\tau)^2)} + B^* e^{-j(2\pi f_c (t-\tau) + \pi S (t-\tau)^2)}, \quad \tau < t < T_c \tag{3}$$

This signal is then processed under an electronic mixer and a low-pass filter to remove high-frequency mixing product, giving a signal modeled as follows:

$$s_{IF}(t) = AB^* e^{j(\tau(2\pi f_c + 2\pi S t - \pi S \tau))} + A^* B e^{-j(\tau(2\pi f_c + 2\pi S t - \pi S \tau))}, \quad \tau < t < T_c \tag{4}$$

Equation (4) is the main expression we compute. The first term can be expressed as $2\pi f_c \tau = 4\pi \frac{d_o + vt}{\lambda_{max}}$ where $\lambda_{max} = \frac{c}{f_c}$. It can be observed that the velocity varies as a function of $\lambda_{max} \sim 1^{-3} m$. Thus, although the change in distance due to the velocity vt barely affects the range detected, the velocity can be extracted because vt is in the scale of the maximum wavelength.

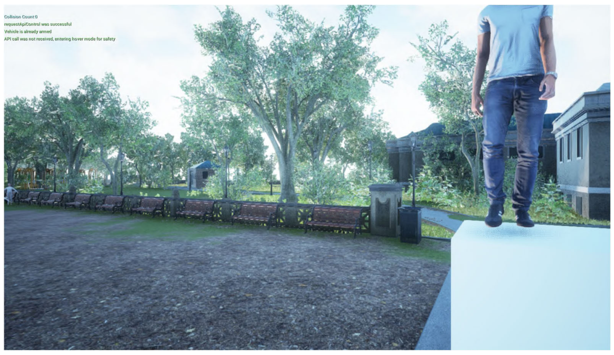
Three receiver antennas are simulated as in the BGT60TR13C FMCW RADAR, whose layout is displayed in Fig. 11 to extract the horizontal (azimuth) and vertical (elevation) angles of the objects. Due to the distance between antennas, the emitted electromagnetic waves arrive at the receiver with a time delay, resulting in a relative phase difference.

We simulate the three antennas. Referring to the nomenclature of Fig. 11, the first antenna is taken as a reference, which means that no phase shift is associated with its signal. The antenna R_{X2} is shifted in the horizontal and vertical direction with respect to R_{X1}; therefore, the relative phase difference can be expressed as [30]:

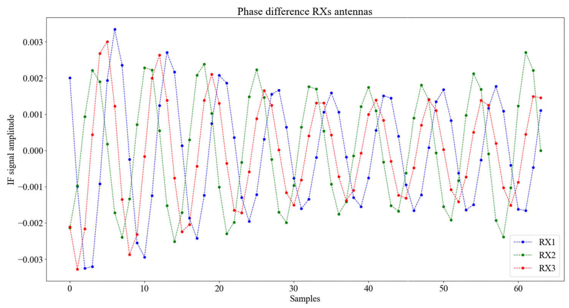
$$\Delta\theta_2 = \frac{2\pi d_{antenna}}{\lambda_{max}} sin(\theta) cos(\phi) + \frac{2\pi d_{antenna}}{\lambda_{max}} sin(\phi) \tag{5}$$

where θ is the azimuth angle, ϕ is the elevation angle and $d_{antenna}$ is the distance between receiver antennas. The phase difference can be observed in the resulting IF signal. Fig. 12 shows the sensor view ad the IF signal amplitude (64 samples) of the first chirp or each antenna. In this figure, the only detected object is the person. Since the first antennas to receive the signal are R_{X2} and R_{X3}, the signal received by R_{X1} is represented as behind in time. The same happens with R_{X2} and R_{X3} in the vertical plane. Fig. 13 shows the same behavior in the case of negative azimuth and elevation angles.

The amplitude of the IF signal is also simulated according to the RADAR equations. The electromagnetic waves

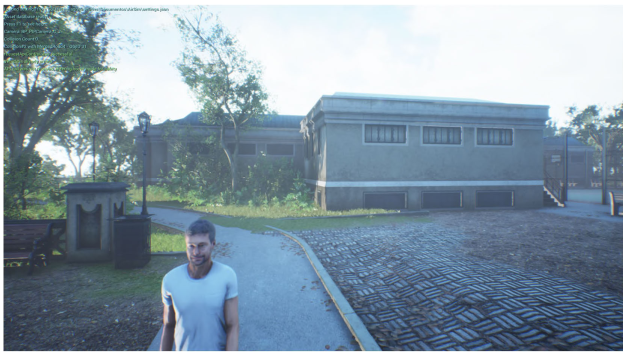


(a)

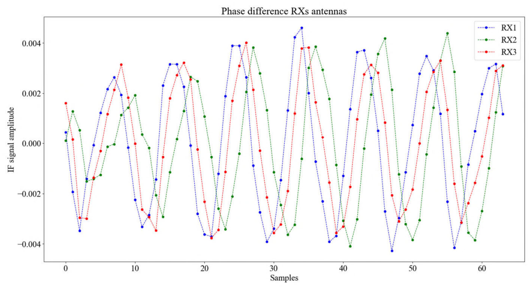


(b)

FIGURE 12. Person with positive azimuth and elevation angles (a) and corresponding IF Signal amplitude of the 3 RX antennas (b).



(a)



(b)

FIGURE 13. Person with negative azimuth and elevation angles (a) and corresponding IF Signal amplitude of the 3 RX antennas (b).

propagate with a spherical waveform. The radiated power density is given by $\frac{P_E G_E}{4\pi d^2}$, where G_E is the emitter antenna gain, and P_E is the emitted power. The power reflected by the object is the radiate power density multiplied by σ, which is the RADAR Cross-Section (RCS). The RCS measures the targetB4s ability to reflect the RADAR signal to the RADAR. The power density at the RX antenna is $\frac{P_E G_E}{4\pi^2 d^4}$, and the power captured by the antenna is given by the multiplication of the power density by the effective aperture area $\frac{G_R \lambda^2}{4\pi}$, where G_R is the receiver antenna gain [31]. Therefore, the relation between receiver power P_R and emitter power P_E is given by:

$$\frac{P_R}{P_E} = \frac{\lambda^2 G_E G_R \sigma}{4\pi^3 d^4} \tag{6}$$

In order to adapt the RADAR equation to the simulated input data (i.e., depth image), we impose that $\frac{P_E}{4\pi d^2} = 1$. This is because in depth images, distant objects will appear as smaller (which means occupying a smaller number of pixels) and, since we compute the RADAR response by summing the individual pixel contributions, the response of the further targets would otherwise be unrealistic. By imposing the condition above, the signal power per pixel decreases quadratically with the distance, i.e., in the same way as the pixel area increases. Therefore, the attenuation information is included with the reduction of the pixel area of the object in the image. The output of a RADAR consists of a voltage amplitude, which we obtain by:

$$V_R = \sqrt[2]{G_E G_R \sigma}\,\frac{\lambda}{4\pi d} \tag{7}$$

V. EXPERIMENTS AND RESULTS

A. INSTALLATION

The proposed simulation framework is based on AirSim; therefore, installing the simulator on an adequate machine is necessary to test its features. We tested our simulator in Windows 10, but AirSim is also available for Linux and MAC. In addition, it can also be interfaced with the Robot Operating System (ROS) through a wrapper. We have chosen the ASUS TUF GAMING F15 with 6 cores, 16GB RAM, and a GeForce RTX 3060 6GB Video RAM, as it allows us to carry out the simulations smoothly. We consider this machine a reference for the minimum system requirements as long as the CPU and RAM are fully used during the simulator execution.

AirSim can be easily installed following the official documentation at [32]. AirSim is, in turn, a plugin that has to be deployed in an Unreal project [33]. The scenarios have been created based on the Unreal Engine packages called *City Park Environment Collection*, *Scanned 3D People Pack*, *Modular Military Operation Urban Training Environment - Civilian Pack*, which can be downloaded from the Epic Games Launcher application.

The AirSim repository contains a Python API within the folder called B4*PythonClient*. The scripts containing the

TABLE 4. Drivers and software versions in Windows 10.

Drivers and software versions in Windows 10						
GPU drivers	CUDA Controller	CUDA Toolkit	cuDNN	Python	TensorFlow	Pytorch
511.65	11.6.99	11.5	8.3.1	3.7	2.8.0	1.10.2

TABLE 5. FPS performance comparison - *Offline Version*.

FPS offline version						
Script	RADAR	*Low*	*Medium*	*High*	*Epic*	*Cinematic*
without storage	✓	7.51	5.86	4.8	4.56	4.6
without storage	✗	25.5	20.3	18.74	18.4	17.63
with storage	✓	7.17	5.5	4.78	4.51	4.4
with storage	✗	20.1	19.4	17.2	16.17	15.17

implementations of our demonstrations have been added to this folder.

Training deep learning algorithms in the high-end laptop requires properly configuring the NVIDIA graphic card driver and libraries. The drivers and software version in Windows 10 are shown in Table 4

B. DATA ACQUISITION

1) OFFLINE MODE

In this subsection, we evaluate the *offline version*, which is intended for the recording of a synthetic multisensor dataset.

The dataset is obtained by recording the drone sensor data while traveling along a pre-planned square trajectory loop in a park scene with trees, buildings, and people. The drone's linear velocity is set to 3 m/s to avoid aggressive drone movements when changing directions. The dataset consists of 500 frames recorded all over the path.

The *Environment display* offers five different graphic quality levels, ranging from *Low* to *Cinematic*. The graphic quality depends on various parameters, such as the view distance, shadow rendering, textures, or shading.

The overall latency can be divided into the following:

- t_1: latency to generate and extract the state information from the simulator
- t_2: latency to save data into the SSD.

Since latency t_2 mostly depends on the operating system, we evaluate the overall performance based on the time employed to generate and extract the data from the simulator without considering the storage time. As the RADAR is the most computationally demanding sensor to simulate, we evaluate the FPS with and without the RADAR sensor. Table 5 shows the FPS calculated in each quality mode, calculated by dividing the number of frames (500) by the total elapsed time.

Only a tiny drop in the FPS can be observed by adding the data storage phase. The medium quality provides the best trade-off between achievable acquisition frequency and graphics fidelity. However, the graphics rendering in the recorded dataset is delimited by the camera image resolution (320 × 240 pixels). Consequently, getting higher engine resolution is not relevant in this scope. The other experiments described in the following sections are carried out in *Low* quality mode.

TABLE 6. *Online Version* performance comparison of connection versions.

Online version performance comparison of connection versions		
Version	Ports	FPS
SUT and CUT same machine	1	8.45
SUT and CUT in different machines	1	11.72
SUT and CUT in different machines	5	8.44

2) ONLINE MODE

The Data Acquisition procedure is also implemented in the *Online Mode* as it provides the input data to the sensor fusion algorithms.

Three different versions have been evaluated:

- *SUT and CUT running in the same machine*
- *SUT and CUT running on different machines*: SUT running on five threads and CUT running on a single thread connected through a single port
- *SUT and CUT running on different machines*: SUT runs on five different processes, and the CUT runs on five different threads connected through five ports.

The first two implementations were discussed in Section IV. As for the third version, five ports are opened both by the SUT and CUT. Each process in the SUT and thread in the CUT binds to a specific port. Four processes are involved in the data gathering, while the fifth applies the control commands to the drone. The processes are synchronized through a *Barrier* flag. Four processes of the CUT are in charge of storing the data retrieved from the SUT in a fixed position within a Python list. Finally, the fifth thread runs the tested sensor fusion algorithms and sends the desired state to the SUT.

The scene configuration is identical to the one described in the previous subsection, while the drone executes a 360 degrees yaw motion while recording 1000 frames.

The evaluation considers the time employed to gather the data without processing. The aim is to keep the t_2 and t_3 latencies constant for every experiment.

Table 6 summarizes the results. We can observe that the second implementation outperforms the others as it minimizes the t_1 latency.

C. OBJECT DETECTION

This section describes the deployment and testing of a people detection demonstration with the aid of the simulator. The object detection algorithm is based on a Convolutional Neural Network that was trained on the high-end laptop and deployed on one onboard computer. The process followed to build and deploy the object detection algorithm is summarized below:

- **Network architecture and design.** The design of a neural network structure can be a time-consuming task due to its complexity, so it can be preferable to leverage an existing model. In this case, the *mobilenet-Single-Shot multibox Detection* (SSD) [34] model has been selected. SSD-mobilenet is intended to perform object detection keeping memory consumption low while guaranteeing high accuracy [35]. The object detection task is divided into two steps, namely *Classification*, performed by the neural backbone *mobilenet*, and *Location/Detection* by the *SSD* structure.
  Before performing the object detection, the network must be trained based on rather large and varied datasets (containing around 10 thousand images). This can be a rather time-consuming process, which is fortunately not always necessary thanks to the leverage of Transfer Learning (TL) techniques. Transfer Learning consists of using a pre-trained network whose last(s) layer(s) are re-trained while the rest are frozen. TL saves training time and guarantees a good level of accuracy provided that the images contained in the user dataset and the original one has similar characteristics [36].
- **Dataset acquisition.** The dataset employed in the training of the last layers was acquired with the experiment described in Subsection V-B. The resulting dataset has been later adapted in its format to feed the SSD-mobilenet model. The frames are divided into three subsets, namely training (60%), validation (20%), and test (20%).
- **Model Training.** The training step has been carried out on the high-end laptop as it would guarantee higher accuracy and lower elapsed time thanks to the leverage of GPU features. A comparison between the training performance of the Jetson Nano and the high-end laptop is available in Table 7. The computing power is measured in Floating Point Operations Per Second (FLOPS).
- **Model Deployment.** Before the model deployment, the NVIDIA Jetson Nano followed the instruction in Table 4. Two different approaches have been considered to perform the inference.
  The first approach uses PyTorch to load the model and make predictions. We first create the *mobilenet-SSD* structure and then load the parameters. Before the simulated drone flight starts, performing an initial prediction on a previously loaded image is necessary because the first prediction takes way longer than the others. During the inference, the successive outputs three variables per detection: the box coordinates, the label of the class, and the detection probability. The drone image can be displayed synchronously in a window and the bounding box coordinates $(x_{min}, x_{max}, y_{min}, y_{max})$ allow to draw a rectangle over the detected object. The inference latency t_2 reduces the number of the achieved FPS.
  The second approach aims to optimize the prediction and to reduce latency t_2 using TensorRT [37], an SDK for high-performance deep learning inference. TensorRT optimizes the neural network model in order to reduce the inference time. Fig. 14 shows the camera image processed by the object detection algorithm using TensorRT.

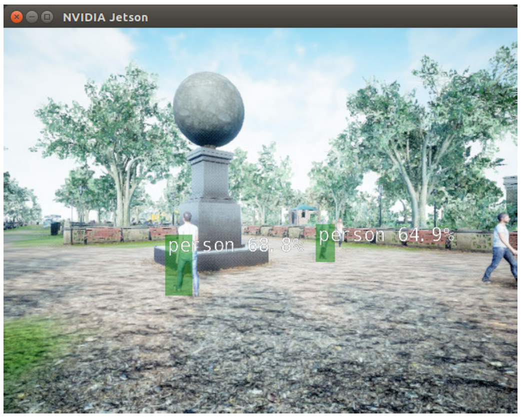


FIGURE 14. TensorRT inference in real time.

We run the SUT and CUT in different machines connected through a single port and evaluate the inference using PyTorch and TensorRT as shown in Table 8.

D. RADAR SIMULATION

The RADAR simulation has been validated by comparing the simulated data with real measurements collected with the Infineon BGT60TR13C RADAR sensor in a similar scenario. The real data acquisitions were recorded outdoors at the Infineon Technologies AG headquarters in Munich (Germany). A dataset of 500 frames was obtained at 13 FPS with the support of a laptop and a Python API. The RADAR data were processed following the same procedure mentioned in Subsection IV-D.

The real-life experiments have been carried out on two different scenarios where the RADAR is kept static at the height of 1.8 m and a person moves along two different trajectories:

- walking away from the sensor with constant speed along a straight line until a distance of 7 m is reached;
- walking towards the RADAR with constant speed starting from a distance of 7 m to 1 m.

The same scenarios and movements have been recreated in AirSim, where only the people and the ground have been included in the segmentation images. A separate dataset has been recorded for each scenario to ease the programming of the target's trajectory. The two datasets are composed of 100 frames, each collected at a frequency of 7 FPS. The RADAR model was tuned with the parameters shown in Table 2.

TABLE 7. High-end and on-board computer comparison.

High-end and on-board computer comparison		
Feature	ASUS TUF 15 FX506HM	NVIDIA Jetson Nano
Processor	Intel Core i5-11400H	Quad-Core Cortex-A57
GPU	NVIDIA GeForce RTX 3060 Laptop	NVIDIA Tegra X1
CUDA cores	3840	128
RAM	16GB	4GB
Computing power	11.4 FLOPS	472 GFLOPS

TABLE 8. AI performance comparison of inference versions.

Performance comparison of inference versions		
Version	Visualization	FPS
PyTorch	Visualization	4.31
	No visualization	4.6
TRT	Visualization	8.29
	No visualization	8.76

TABLE 9. NCC coefficients obtained in both scenarios.

NCC coefficients obtained in both scenarios		
Scenario	Frames evaluated	NCC coefficient
1	18	0.39
2	11	0.42

The real and simulated data were compared based on the range-doppler images. They consist of matrices of dimensions (36, 64), where each element represents the detection's intensity value with a specific velocity-range value normalized between 0 and 1. Each pair of range-doppler images has been evaluated by calculating the Normalized Cross Correlation (NCC) coefficient, commonly used to measure the similarity between two images [38]. The NCC coefficient can be expressed as:

$$r = \frac{\sum_i \sum_j (x_{i,j} - \bar{x})(y_{i,j} - \bar{y})}{\sqrt{\sum_i \sum_j (x_{i,j} - \bar{x})^2 \sum_i \sum_j (y_{i,j} - \bar{y})^2}} \tag{8}$$

where $\bar{x}$ and $\bar{y}$ are the mean intensities of the simulated and real range-doppler images, respectively. The images from the real and simulated data were paired according to the distance of the human target from the sensor. The resulting correlation is shown in Table 9.

It is worth pointing out a few elements affecting the NCC coefficient results. First, the movement of the human target of the simulator can only partially match the motion of the actual person. Then, the real data images contain an undesired contribution at zero speed and range, which is absent in the simulated data. This is due to the absence of noise models in the simulation, while, in real life, noise can create false detections and limit the maximum range. Furthermore, the proposed approach is limited to an approximation of the reflection properties, taking into account the normality of the surface but not the roughness and electrical properties of the materials. Fig. 15 displays the doppler images obtained by the real and simulated measurements, respectively, in the first scenario (person walking towards the sensor). In contrast, Fig.16 represents the second scenario (person walking away from the sensor). Despite the aforementioned undesired contributions, we can observe that the simulated sensor correctly detected the targets.

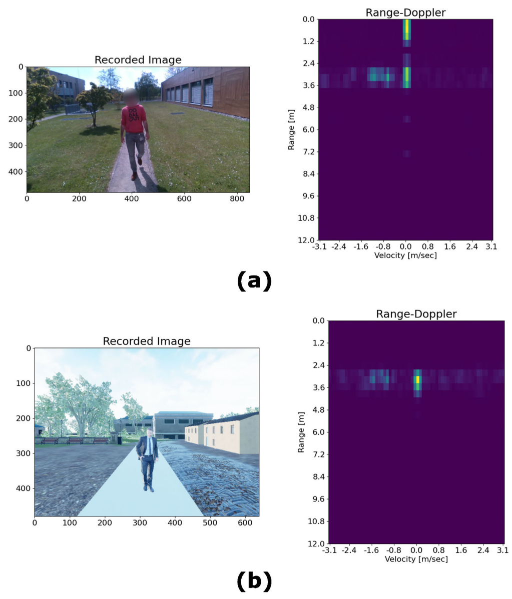


FIGURE 15. Person moving towards the RADAR. Real data and simulated data.

E. SITL AND HITL FLIGHT DEMONSTRATIONS

In this section, we describe two flight demonstrations carried out with the help of the proposed framework. First, the experiment described in Subsection V-E1 consists of a Hardware-in-the-Loop flight simulation where the drone model is commanded by an actual flight controller (Pixhawk 2) while traveling a user-defined route. Then, subsection V-E2 discusses a Software-in-the-Loop simulation of navigation in an unknown environment.

We wish to point out that no performance analysis is provided for the SUTs in this section since the performance optimization of the involved algorithms is beyond the scope of this work. Indeed, our aim is exclusively to prove the possibility to perform HITL and SITL simulations of complex navigation systems with the support of AirLoop.

1) HITL DRONE FLIGHT WITH PIXHAWK 2

Our simulation framework can be interfaced with ArduPilot, a Commercial-off-the-Shelf Flight Controller Running on a Pixhawk 2 to carry out Hardware-in-the-Loop Simulations. In more detail, the Pixhawk 2 is connected to the high-end

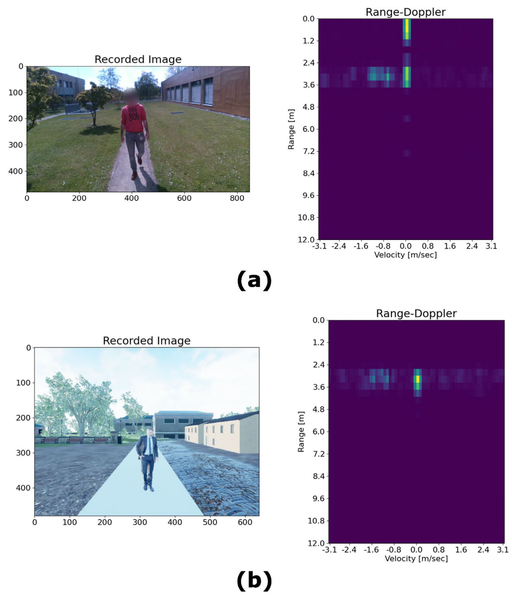


FIGURE 16. Person moving away from the RADAR. Real data and simulated data.

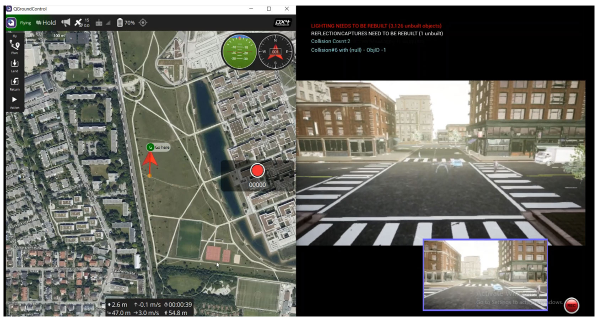


FIGURE 17. Autonomous navigation with Pixhawk 2.

laptop via a USB port and the communication between the drone model and ArduPilot takes place through the Mavlink messaging protocol. In order to command the drone through the Pixhawk 2, it is necessary to install the QC Ground Control Software. A complete Tutorial on the interfacing between AirSim-based simulators and ArduPilot is available at [39].

In this experiment, four waypoints are set as the vertices of a rectangular trajectory. The trajectory tracking is carried out via GPS. Although the navigation is based on the simulated sensors, QC Ground Control needs to be initialized with a real GPS receiver. Therefore, although the flight takes place in a simulated environment, the QC Ground Contol GUI shows a fictitious trajectory where the drone is in a real location. In this case, the experiments were carried out at the Infineon Technologies AG headquarters in Neubiberg (Germany), as shown in Fig. 17.

2) SITL AUTONOMOUS FLIGHT IN UNKNOWN ENVIRONMENT

We now describe a Software-in-the-Loop simulation of a fully autonomous flight demonstration in an unknown

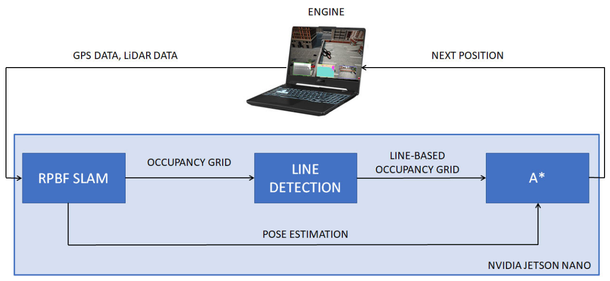


FIGURE 18. Autonomous Navigation pipeline.

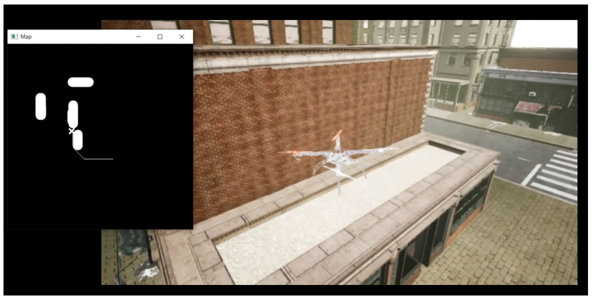


FIGURE 19. Autonomous Navigation in Unknown environment demonstration - frame.

environment. In this experiment, the drone is initially positioned in a known location and is supposed to reach a user-defined target point (expressed in map coordinates) in a completely autonomous way without any previous knowledge of the surrounding environment. Therefore, the following problems have to be solved:

- Find a suitable path to reach the target location in the shortest time possible while avoiding obstacles (path planning and obstacle avoidance)
- Create a map of the surrounding environment (mapping)
- Provide high-precision self-pose estimation (localization).

The simulation environment where the flight takes place is a custom urban-like environment based on the package *Modular Military Operation Urban Training Environment - Civilian Pack*. The drone is equipped with a LiDAR and a GPS sensor. The data acquisition runs on the high-end laptop, while the sensor fusion pipeline runs on an NVIDIA Jetson Nano according to the settings described in Subsection IV-B2.

In order to plan a suitable trajectory to reach a target location, at least a partial knowledge of the surrounding environment is necessary. This means that a partial version of the map has to be available before starting to plan the path. This explains the sensor fusion pipeline schema in Fig. 18. The sensor data coming from the engine (GPS and LiDAR) are given as inputs to a Rao-Blackwellized Particle Filter (RBPF) Simultaneous Localization and Mapping (SLAM) algorithm [40]. RBPF SLAM provides a probabilistic pose estimation of the drone and a progressively updated environmental map. In more detail, the state estimation is represented

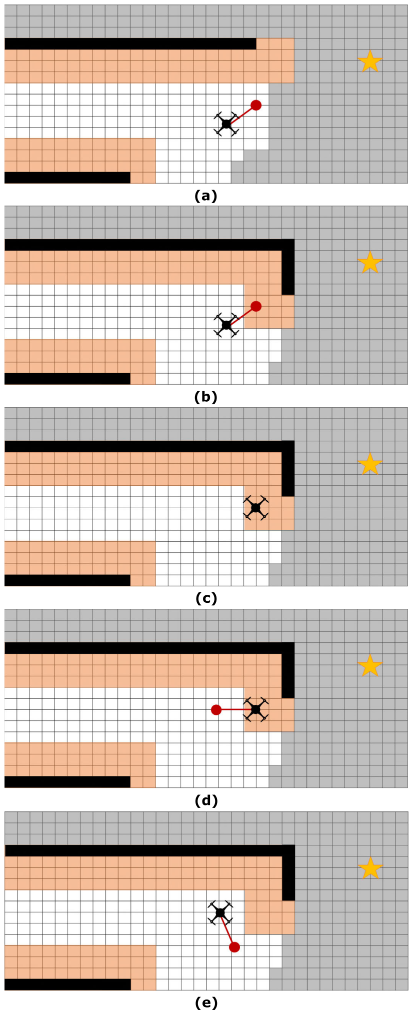


FIGURE 20. **Obstacle avoidance strategy example. The images shown in this figures are not taken from the simulator and are only aimed at providing a graphical representation of the obstacle avoidance strategy. The known occupied cells are highlighted in black, while the free space is colored in white. The unexplored cells are highlighted in grey, while the cells surrounding an obstacle are highlighted in orange. The final target location is highlighted with a yellow star, while the next planned position is marked with a red dot. A detailed explanation of the subfigures is provided in Section V-E2.**

by a probability distribution, which, in turn, is approximated by a set of weighted samples (particles). The output of RBPF SLAM consists of a pose estimation of the drone and an environmental representation in the form of an Occupancy Grid. An Occupancy Grid schematizes the world as a set of cells with uniform dimensions that can be considered free or occupied. The cells can be either bidimensional or tridimensional. For the sake of this demonstration, the environment is approximated as 2D as the drone flies at a constant altitude.

The pose estimation and the updated Occupancy Grid are the inputs for the path planning algorithm A* [41]. A* is a widely popular graph-based algorithm based on the Dijkstra algorithm. Starting from a given node, the A* algorithm selects the next nodes based on an estimate of their distance from the target location. The binary Occupancy Grid delivered by the SLAM algorithm can be schematized as a graph, where each free cell is considered a node, and the distance between two adjacent cells is an edge weight. Consequently, the SLAM algorithm provides a new pose estimation and an updated map at every iteration, and, with these inputs, A* plans a new path starting from the current location. Subsequently, the first move of the new planned path is actuated by the drone model.

The creation of the Occupancy Grid can be affected by several factors, such as sensor noise or the momentary presence of moving targets, which could result in some cells being wrongly considered occupied. In addition, the specific conformation of our environment and the constant flight altitude enables approximating the environmental features as a set of lines. Therefore, a line detection algorithm is applied to the Occupancy Grid delivered by the SLAM algorithm to eliminate false detections. Moreover, since the map is continuously updating, A* can cause the drone to move toward an obstacle. In addition, the actuators can cause overshooting in the drone motion, resulting in the drone getting closer to the obstacles than planned. To prevent this scenario from occurring, the strategy described below is actuated.

- Based on the current knowledge of the environment, the cells in the Occupancy Grid are assessed as either free or occupied, as shown in Fig. 20 (a). In more detail, the occupied cells are highlighted in black, the free cells in white and the unexplored cells are colored in grey. However, to prevent the scenarios mentioned above from occurring, a larger set of cells are fictitiously considered occupied (highlighted in orange) to prevent A* from directing the drone too close to the known obstacles. The width of the *orange area* is chosen considering the drone's dimensions and speed.
- Once the map is updated, the next pose may fall within the *orange area* (Fig. 20 (b) and 20 (c)).
- When this happens, the next move is planned to take distance from the obstacle. Therefore, the drone will move away from the obstacles perpendicularly to the obstacle surface (Fig. 20 (d)).
- Then, the successive moves are regularly planned according to A* until a similar scenario occurs (Fig. 20 (e)).

For the sake of this demonstration, the drone is given a reference speed of 3 m/s and a reference height of 10 meters. The sampling frequency of the onboard sensors is set to 10 FPS. The UAV starts at a known position identified as the map's origin (0,0), and the final target location is (70m, 60m). A frame showing the drone moving within the environment and the updating map is shown in Fig. 19.

VI. CONCLUSION

In this paper, we presented AirLoop: a UAV simulation environment for the testing and validating Sensor Fusion algorithms for autonomous navigation. The proposed simulator is based on AirSim and offers a variety of sensors, including a novel simulation model of an FMCW RADAR. A key novelty in the proposed RADAR simulator is the possibility of obtaining synthetic raw data. The simulated RADAR has been evaluated based on the Infineon. Technologies BGT60TR13C RADAR.

Furthermore, the simulation engine enables the execution of HITL and SITL simulations. Communication with NVIDIA Jetson Nano through LAN has been evaluated in different modalities and exploited to carry out several demonstrations. Experiments such as the acquisition of synthetic data, object detection, and autonomous navigation in an unknown environment have been performed in more detail.

In addition, the possibility of interfacing the proposed simulation engine with a COTS flight controller has been demonstrated with a GPS-aided path planning simulation.

LIST OF ABBREVIATIONS

The following abbreviations are used in the manuscript:

ADC	Analog-to-Digital Converter
AI	Artificial Intelligence
ANN	Artificial Neural Network
API	Application Programming Interface
ASL	Autonomous System Lab
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
CUT	Client Under Test
CW	Continuous Wave
FLOPS	Floating Point Operations Per Second
FMCW	Frequency Modulated Continuous Wave
FOV	Field Of View
FPS	Frames Per Second
GPS	Global Positioning System
GPU	Graphics Processing Unit
GUI	Graphical User Interface
HDD	Hard Drive Disk
HITL	Hardware In The Loop
IF	Intermediate Frequency
IMU	Inertial Measurement Unit
IP	Internet Protocol
LAN	Local Area Network
LiDAR	Laser imaging Detection and Ranging
MAV	Micro Aerial Vehicle
NCC	Normalized Cross Correlation
OS	Operating System
RADAR	Radio detection and ranging
RAM	Random Access Memory
RBPF	Rao-Blackwellized Particle Filter
RCP	Remote Procedure Call
RCS	RADAR Cross-Section
RL	Reinforcement Learning
ROS	Robot Operating System
RX	Receive
SDK	Software Development Kit
SITL	Software in the Loop
SLAM	Simultaneous Localization and Mapping
SSD	Solid State Drive
SSD	Single-Shot multibox Detection
SUT	System Under Test
TCP	Transmission Control Protocol
TensorRT	Tensor RunTime
TL	Transfer Learning
ToF	Time-of-Flight
TX	Transmit
UAV	Unmanned Aerial Vehicle
ViRa	Virtual RADAR
VOC	Visual Object Classes

REFERENCES

[1] M. Hassanalian and A. Abdelkefi, “Classifications, applications, and design challenges of drones: A review,” *Progr. Aerosp. Sci.*, vol. 91, pp. 99–131, May 2017.

[2] F. Samadzadegan and G. Abdi, “Autonomous navigation of unmanned aerial vehicles based on multi-sensor data fusion,” in *Proc. 20th Iranian Conf. Electr. Eng. (ICEE)*, May 2012, pp. 868–873.

[3] A. Bittar, H. V. Figuereido, P. A. Guimaraes, and A. C. Mendes, “Guidance software-in-the-loop simulation using X-plane and simulink for UAVs,” in *Proc. Int. Conf. Unmanned Aircr. Syst. (ICUAS)*, May 2014, pp. 993–1002.

[4] P. S. Andrews, S. Stepney, and J. Timmis, “Simulation as a scientific instrument,” in *Proc. Workshop Complex Syst. Modelling Simulation*, Orleans, France. Citeseer, 2012, pp. 1–10.

[5] A. Mairaj, A. I. Baba, and A. Y. Javaid, “Application specific drone simulators: Recent advances and challenges,” *Simul. Model. Pract. Theory*, vol. 94, pp. 100–117, Jul. 2019.

[6] J. Glossner, S. Murphy, and D. Iancu, “An overview of the drone open-source ecosystem,” 2021, *arXiv:2110.02260*.

[7] S. Shah, D. Dey, C. Lovett, and A. Kapoor, “AirSim: High-fidelity visual and physical simulation for autonomous vehicles,” in *Field and Service Robotics*. Cham, Switzerland: Springer, 2018, pp. 621–635.

[8] N. Valcasara, *Unreal Engine Game Development Blueprints*. Birmingham, U.K.: Packt, 2015.

[9] S. Krishnan, B. Boroujerdian, W. Fu, A. Faust, and V. J. Reddi, “Air learning: A deep reinforcement learning gym for autonomous aerial robot visual navigation,” *Mach. Learn.*, vol. 110, no. 9, pp. 2501–2540, Sep. 2021.

[10] Y. Song, S. Naji, E. Kaufmann, A. Loquercio, and D. Scaramuzza, “Flightmare: A flexible quadrotor simulator,” 2020, *arXiv:2009.00563*.

[11] J. K. Haas, “A history of the unity game engine,” *Diss. WORCESTER Polytech. Inst.*, vol. 483, p. 484, Mar. 2014.

[12] UZH-RPG. *ROS Tutorials · Issue #37 · UZH-RPG/Flightmare*. Accessed: Dec. 12, 2022. [Online]. Available: https://github.com/uzh-rpg/flightmare/issues/37

[13] C. E. Agüero, N. Koenig, I. Chen, H. Boyer, S. Peters, J. Hsu, and B. Gerkey, “Inside the virtual robotics challenge: Simulating real-time robotic disaster response,” *IEEE Trans. Automat. Sci. Eng.*, vol. 12, no. 2, pp. 494–506, Apr. 2015.

[14] N. Koenig and A. Howard, “Design and use paradigms for Gazebo, an open-source multi-robot simulator,” in *Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst.*, vol. 3, Oct. 2004, pp. 2149–2154.

[15] F. Furrer, M. Burri, M. Achtelik, and R. Siegwart, “Rotors—A modular gazebo MAV simulator framework,” in *Robot Operating System (ROS)*. Cham, Switzerland: Springer, 2016, pp. 595–625.

[16] K. Zheng, “ROS navigation tuning guide,” in *Robot Operating System (ROS)*. Cham, Switzerland: Springer, 2021, pp. 197–226.

[17] R. Salem. *Rohitsalem/Projection: Tools for Auto Generation of Image Datasets With Annotated Bounding Boxes*. Accessed: Dec. 12, 2022. [Online]. Available: https://github.com/rohitsalem/projection

[18] *BGT60TR13C Technical Datasheet*, Infineon Technologies, Neubiberg, Germany, 2021.

[19] M. Ciarambino, Y.-Y. Chen, and N. Peinecke, “A game engine-based millimeter wave radar simulation,” *Proc. SPIE*, vol. 11759, pp. 21–29, Apr. 2021.

[20] *Cosmosim: Radar Sensor Simulation Framework*. Accessed: Dec. 12, 2022. [Online]. Available: https://www.otsl.jp/en/product/cosmosim/

[21] *Radar Signal Simulation and Processing for Automated Driving*. Accessed: Dec. 12, 2022. [Online]. Available: https://de.mathworks.com/help/radar/ug/radar-signal-simulation-and-processing-for-automated-driving.html

[22] M. Burri, J. Nikolic, P. Gohl, T. Schneider, J. Rehder, S. Omari, M. W. Achtelik, and R. Siegwart, "The EuRoC micro aerial vehicle datasets," *Int. J. Robot. Res.*, vol. 35, no. 10, pp. 1157–1163, 2016.
[23] M. Everingham, A. Zisserman, and C. K. I. Williams, "The 2005 PASCAL visual object classes challenge," in *Proc. Mach. Learn. Challenges Workshop*. Berlin, Germany: Springer, 2005, pp. 117–176.
[24] K. C. Leung, V. O. K. Li, and D. Yang, "An overview of packet reordering in transmission control protocol (TCP): Problems, solutions, and challenges," *IEEE Trans. Parallel Distrib. Syst.*, vol. 18, no. 4, pp. 522–535, Apr. 2007.
[25] A. D. Birrell, "An assessment of the remote procedure call mechanism," in *Proc. 5th Workshop ACM SIGOPS Eur. Workshop Models Paradigms Distrib. Syst. Structuring (EW)*, 1992, pp. 1–3.
[26] M. Chmurski, M. Zubert, K. Bierzynski, and A. Santra, "Analysis of edge-optimized deep learning classifiers for radar-based gesture recognition," *IEEE Access*, vol. 9, pp. 74406–74421, 2021.
[27] C. Schoffmann, B. Ubezio, C. Bohm, S. Muhlbacher-Karrer, and H. Zangl, "Virtual radar: Real-time millimeter-wave radar sensor simulation for perception-driven robotics," *IEEE Robot. Autom. Lett.*, vol. 6, no. 3, pp. 4704–4711, Jul. 2021.
[28] S. K. Lam, A. Pitrou, and S. Seibert, "Numba: A LLVM-based Python JIT compiler," in *Proc. 2nd Workshop LLVM Compiler Infrastruct. (HPC)*, 2015, pp. 1–6.
[29] J. Lin Jr., Y.-P. Li, W.-C. Hsu, and T.-S. Lee, "Design of an FMCW radar baseband signal processing system for automotive application," *SpringerPlus*, vol. 5, no. 1, pp. 1–16, 2016.
[30] G. M. Brooker, "Understanding millimetre wave FMCW radars," in *Proc. 1st Int. Conf. Sens. Technol.*, vol. 1, 2005, pp. 1–6.
[31] C. Iovescu and S. Rao, "The fundamentals of millimeter wave sensors," Texas Instruments, Dallas, TX, USA, 2017, pp. 1–8.
[32] *Build AirSim on Windows*. Accessed: Dec. 12, 2022. [Online]. Available: https://microsoft.github.io/AirSim/build_windows/
[33] *Development Workflow—AirSim*. Accessed: Dec. 12, 2022. [Online]. Available: https://microsoft.github.io/AirSim/dev_workflow/
[34] Y.-C. Chiu, C.-Y. Tsai, M.-D. Ruan, G.-Y. Shen, and T.-T. Lee, "MobileNet-SSDv2: An improved object detection model for embedded systems," in *Proc. Int. Conf. Syst. Sci. Eng. (ICSSE)*, Aug. 2020, pp. 1–5.
[35] J. Mendez, M. Molina, N. Rodriguez, M. P. Cuellar, and D. P. Morales, "Camera-LiDAR multi-level sensor fusion for target detection at the network edge," *Sensors*, vol. 21, no. 12, p. 3992, 2021.
[36] S. J. Pan and Q. Yang, "A survey on transfer learning," *IEEE Trans. Knowl. Data Eng.*, vol. 22, no. 10, pp. 1345–1359, Jan. 2021.
[37] Dusty-NV. *Dusty-NV/Jetson-Inference: Hello AI World Guide to Deploying Deep-Learning Inference Networks and Deep Vision Primitives With Tensorrt and NVIDIA Jetson*. Accessed: Dec. 12, 2022. [Online]. Available: https://github.com/dusty-nv/jetson-inference
[38] J.-C. Yoo and T. H. Han, "Fast normalized cross-correlation," *Circuits, Syst. Signal Process.*, vol. 28, pp. 819–843, Dec. 2009.
[39] *Px4 Setup for AirSim*. Accessed: Dec. 12, 2022. [Online]. Available: https://microsoft.github.io/AirSim/px4_setup/
[40] K. P. Murphy, "Bayesian map learning in dynamic environments," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 12, 1999, pp. 1–7.
[41] H.-Y. Zhang, W.-M. Lin, and A.-X. Chen, "Path planning for the mobile robot: A review," *Symmetry*, vol. 10, no. 10, p. 450, 2018.

JESSICA GIOVAGNOLA received the B.S. and M.S. degrees in automation and control engineering from Politecnico di Milano, Italy, in 2018 and 2020, respectively. She is currently pursuing the Ph.D. degree with Infineon Technologies AG, Germany, in collaboration with the University of Granada, Spain, with a focus on sensor fusion applications for drone technologies.

She joined Infineon Technologies AG, in March 2021. Her current research interests include simultaneous localization and mapping (SLAM) techniques in GPS-denied environments, edge computing, and sensor fusion for autonomous navigation.

JUAN B. MORO MEGÍAS received the B.Sc. degree in industrial electronic engineering and the M.S. degree in industrial electronics from the University of Granada, Spain, in 2021 and 2022, respectively.

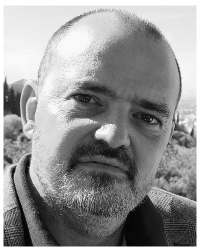
From 2021 to 2022, he was with Infineon Technologies AG, Germany, as a working student, where he worked on real data acquisition and simulation techniques for machine-learning applications for unmanned aerial vehicles. In 2022, he joined MAHLE Electronics, Spain, where he is currently working as a Hardware Engineer on an electric vehicle charger project.

MIGUEL MOLINA FERNÁNDEZ received the B.Sc. degree in electronics engineering from the University of Granada, in 2019, and the M.Sc. degree in electronic systems engineering from the Polytechnic University of Madrid, in 2020. He is currently pursuing the Ph.D. degree with the University of Granada.

He joined Infineon Technologies AG, in August 2020. He is currently working on applications of sensor fusion for drones and into hardware implementations of artificial neural networks, especially for spiking neural networks (SNNs). His main research interests include field-programmable gate arrays (FPGAs), edge computing, artificial neural networks, and industry 4.0.

MANUEL PEGALÁJAR CUÉLLAR received the bachelor's and master's degrees in computer science from the Department of Computer Science and Artificial Intelligence, University of Granada, Spain, in 2003 and 2006, respectively.

Since 2012, he has been a tenured Professor with the University of Granada, where he is currently with the Department of Computer Science and Artificial Intelligence. He has been working in time series forecasting and neural networks, since 2004. His main research interests include neural networks, evolutionary computation and metaheuristics, ambient intelligence, reinforcement learning, and sensor data gathering and modeling for AI applications. His current research interests include quantum machine learning and quantum neural networks and their application to different machine learning fields.



DIEGO P. MORALES SANTOS received the M.Eng. and Ph.D. degrees in electronics engineering from the University of Granada, in 2001 and 2011, respectively.

Since 2001, he has been an Assistant Professor with the Department of Computer Architecture and Electronics, University of Almeria. He joined the Department of Electronics and Computer Technology, University of Granada, in 2006, where he is currently a tenured Professor. He is the Co-Founder of the Biochemistry and Electronics as Sensing Technologies (BEST) Research Group, University of Granada. He has coauthored more than 80 scientific contributions. His current research interests include low-power energy conversion, energy harvesting for wearable sensing systems, and new materials for electronics and sensors.