Unreal Success: Vision-Based UAV Fault Detection and Diagnosis Framework

de Alvear Cardenas, J.I.; de Visser, C.C.

DOI
10.2514/6.2024-0760

Publication date
2024

Document Version
Final published version

Published in
Proceedings of the AIAA SCITECH 2024 Forum

AIAA SciTech Forum
8-12 January 2024, Orlando, FL
AIAA SCITECH 2024 Forum

10.2514/6.2024-0760

Unreal Success: Vision-Based UAV Fault Detection and Diagnosis Framework

José Ignacio de Alvear Cárdenas*†
San Jose State University Research Foundation, Moffett Field, California, 94043, United States

Coen C. de Visser‡
Delft University of Technology, Delft, Zuid Holland, 2629HS, The Netherlands

Online fault detection and diagnosis (FDD) enables Unmanned Aerial Vehicles (UAVs) to take informed decisions upon actuator failure during flight, adapting their control strategy or deploying emergency systems. Despite the camera being a ubiquitous sensor on-board of most commercial UAVs, it has not been used within FDD systems before, mainly due to the nonexistence of UAV multi-sensor datasets that include actuator failure scenarios. This paper presents a knowledge-based FDD framework based on a lightweight LSTM network and a single layer neural network classifier that fuses camera and Inertial Measurement Unit (IMU) information. Camera data are pre-processed by first computing its optical flow with RAFT-S, a state-of-the-art deep learning model, and then extracting features with the backbone of MobileNetV3-S. Short-Time Fourier Transform is applied on the IMU data for obtaining their time-frequency information. For training and assessing the proposed framework, *UUFOSim* was developed: an Unreal Engine-based simulator built on AirSim that allows the collection of high-fidelity photo-realistic camera and sensor information, and the injection of actuator failures during flight. Data were collected in simulation for the Bebop 2 UAV with 16 failure cases. Results demonstrate the added value of the camera and the complementary nature of both sensors with failure detection and diagnosis accuracies of 99.98% and 98.86%, respectively.

I. Nomenclature

b_{cam}, b_{IMU}	=	Camera and IMU buffers
C_{UE4}	=	Occupancy grid cell size, ICF_{UE4} length units
f_{cam}, f_{IMU}	=	Camera and IMU sampling frequency, Hz
f_{FDD}	=	FDD execution frequency, Hz
f_{p}	=	AirSim physics engine thread calling frequency, Hz
f_{res}, t_{res}	=	STFT frequency, Hz, and time resolution, s
k_{UE4}	=	Conversion factor between UE4 and AirSim coordinate frames
n_{seg}	=	Number of IMU samples in b_{IMU} before FDD execution
n_x, n_y	=	Occupancy grid coordinates
n_{win}	=	STFT window size
o	=	STFT window overlap size
p_{i_x}, p_{i_y}	=	Waypoint coordinates in the occupancy grid coordinate frame
$\bar{p}_{i_x}$, $\bar{p}_{i_y}$, $\bar{p}_{i_z}$	=	Waypoint coordinates in the AirSim drone coordinate frame
$\overrightarrow{X}$	=	Vehicle state vector
x, y, z	=	Position coordinates, m
x_r, y_r, z_r	=	Reference position coordinates, m
x_{D_0}, y_{D_0}	=	Drone initial spawn coordinates in ICF_{UE4}
x_{UE4}, y_{UE4}	=	Position coordinates in ICF_{UE4}
ν	=	Measurement noise

*Project Associate, Human Systems Integration Division, San Jose State University Research Foundation, jose.dealvearcardenas@sjsu.edu
†Work performed as MSc Student, Faculty of Aerospace Engineering, Control and Simulation Division, Delft University of Technology
‡Associate Professor, Faculty of Aerospace Engineering, Control and Simulation Division, Delft University of Technology, AIAA member

ψ_r = Drone reference heading, rad
ω = Propeller rotational speed, rad/s
Ω = Vehicle angular velocity, rad/s

II. Introduction

With the advent of Smart Cities, Unmanned Air Vehicles (UAVs) have seen a surge in their number of applications, from package delivery [1, 2] to Urban Air Mobility (UAM) [3]. Most recently, as a response to the COVID-19 pandemic [4], the implementation of UAVs for medical purposes has been accelerated. Zipline, a drone start-up in California (USA), has been granted permission for transporting medical supplies in North Carolina and AVY, a start-up based in Amsterdam (The Netherlands), has received a grant from the European Commission for urgent medical transport between healthcare facilities. With Air Traffic Control programs under development for the management of drones, such as the U-Space in Europe [5], one of the main concerns of the future crowded urban airspace is safety [6].

Most of the research in this field has been focused on fault tolerant control [7], with companies such as Verity Studios successfully filing a patent in 2020 for a final product [8]. However, in order to improve the resilience of multi-rotor and hybrid drones to potential failures, work is also carried out in obstacle avoidance [9], upset recovery [10], system identification [11] or fault detection and diagnosis [12]; the latter consisting of the fault classification, as well as its location and magnitude identification. Literature in actuator fault detection and diagnosis (FDD) is very extensive but it deals with a single failure type at a time and has been limited to the manipulation of signals from the Inertial Measurement Unit (IMU), namely the accelerometers and gyroscopes, or additional external sensors such as microphones or optical flow sensors [13, 14], which add weight and complexity to the system. Cameras are nowadays ubiquitous in commercial UAVs and they have been ignored for this task, even though their information is already being processed for navigation, such as Simultaneous Localisation and Mapping (SLAM) [15], and state estimation in GPS denied urban regions, such as Visual Inertial Odometry [16]. Visual information is very rich and it could potentially identify multiple failure types at once, as well as increase the accuracy when fused with the IMU sensor.

FDD expands the envelope of the UAV's self-awareness and allows informed decisions when deploying emergency systems, such as a parachute, and switching between controllers or internal physics models to counteract a failure. Figure 1 shows a classification of FDD methods from literature which can be divided in 3 main groups: model-based, signal-based and knowledge-based. Historically, knowledge-based approaches have shown to be the most suitable for dealing with high-dimensional visual data, especially with the rise of deep learning model architectures. In contrast with model-based approaches, they do not make use of the physical properties of the system and do not build a mathematical model. However, they require greater amounts of historic data in order to create implicit models.

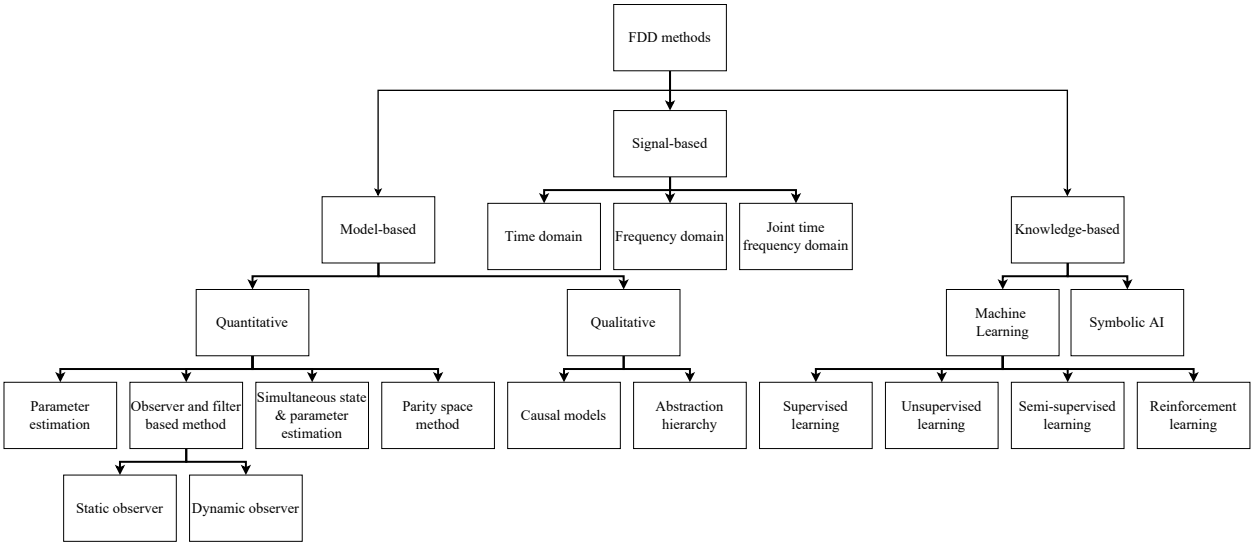

Fig. 1 Fault detection and diagnosis method taxonomy.

Machine learning methods are exploited in modern fault detection systems because they do not require a lot of computations, allowing their use online in real-time. Among them, Long Short-Term Memory (LSTM) networks are a supervised learning method that can be fed sequential data in order to extract temporal relationships to generate an output [17]. It generates predictions based on the current input and an internal state that stores information from an arbitrary number of previous inputs. It is usually combined with batch normalisation (BN) to reduce the internal covariate shift and accelerate the convergence of the algorithm. Zhao et al. [18] exploit this combination to extract dynamic information from data for online FDD within chemical processes and they demonstrate its superiority over alternative knowledge-based FDD approaches.

For the development and performance assessment of vision-based FDD algorithms, it is required to have a dataset which includes IMU and camera output in nominal flight and in failure scenarios. Unfortunately, the current available datasets do not include IMU sensor information, such as the VisDrone dataset [19] or the Indoor Navigation UAV Dataset [20], and do not have any recorded scenarios with failures, such as the UZH-FPV Drone Racing Dataset [21] or the Zurich Urban Micro Aerial Vehicle Dataset [22].

Gathering large quantities of data for knowledge-based fault detection models with an UAV is very time consuming, dangerous and expensive; data would have to be annotated, multiple failure modes would have to be induced in the vehicle and the flight environment, as well as the UAV, would have to be adapted to minimise the potential risk. Besides that, in an experimental physical setting it is very difficult to collect data from various environments and conditions. A suitable alternative is the simulation of the vehicle in a realistic environment, the storage of the sensor synthetic data for model training and the transfer learning to the real world UAV. Research has shown that the addition of large quantities of synthetic data to a smaller real dataset would lead to a performance increment [23].

In previous literature, Gazebo has been the quadrotor simulation tool of choice by the research community leading to simulators such as RotorS [24] and Hector [25]. Despite its high-fidelity physics engine, the output quality of its visual cues is far from photo-realistic. In this regard, there has been an effort in developing high-fidelity simulators for computer vision tasks in the last 10 years. Blender's Game Engine was used for the development of simulators, such as MORSE [26] from 2011 to 2016. However, Unity and Unreal Engine have become the new state-of-the-art (SOTA). Besides their photo-realism, these engines have the benefit of providing an online asset marketplace for the generation of an infinite number of simulation environments.

Examples of photo-realistic simulators are FlightGoggles [27] and Flightmare [28] developed in Unity, and UnrealCV [29], Sim4CV [30] and AirSim [31] developed in Unreal Engine. AirSim was launched in 2017 by Microsoft as an open-source simulator built on Unreal Engine 4 (UE4) for AI research. It is a modular framework that fosters the simulation of autonomous drones and ground vehicles with realistic physics and visual cues. It also includes C++ and Python APIs that allow the researcher to interact programmatically with the vehicle for the extraction of state and sensor information, as well as for providing vehicle control inputs. In contrast with the other simulators, it has an adaptable framework for the introduction of new vehicle models and it is well documented. Thanks to its modularity, later works have been built on AirSim for specialised applications, such as the AirSim Drone Racing Lab [32]. Another promising simulator is Isaac Sim* developed by NVIDIA for the development and deployment of artificial learning applied to robotics in their Omniverse simulator environment. Unfortunately, its computational requirements are beyond the specifications of most commercially available workstations.

The main contribution of this paper is an LSTM-based online FDD framework that fuses camera and IMU data. To this end, the camera information is pre-processed by a SOTA optical flow model in order to extract the magnitude and direction of the vehicle's ego-motion. The IMU data are passed through a Short-Time Fourier Transform for feature extraction. To the authors' knowledge, it is the first time that both sensor sources are combined for UAV FDD. Furthermore, this paper also presents *UUFOSim* (Unreal UAV Failure injectiOn Simulator), a data gathering pipeline built on AirSim for the collection of synthetic flight data with actuator failures in a urban environment.

The potential of UUFOSim has been demonstrated for the Parrot Bebop® 2 UAV. Its aerodynamic model is available from literature [11, 33] and it has been complemented with the blade damage model from [34]. The FDD framework was run at 10 Hz and it had to distinguish between 17 states: 16 failure states (4 levels of blade damage per propeller), and a healthy state. The results show the added value of the camera-IMU combination versus their isolated performances.

The remainder of this paper is organised as follows. Section III describes the data gathering pipeline within UUFOSim. Section IV covers the FDD framework and provides the details of the camera and IMU data pre-processing. Then, section V presents the results when both contributions are applied to the Bebop 2 platform. Finally, concluding remarks and recommendations for further work are provided in section VI.

*https://developer.nvidia.com/isaac-sim

III. UUFO Simulator

The simulator, that the authors have named Unreal UAV Failure injectiOn Simulator (UUFOSim), consists of flying a simulated drone or Undiagnosed Failing Object (UFO) in a urban environment avoiding obstacles between two random locations. UFOs fly at a uniformly sampled constant altitude and an actuator failure, within a set of modes, is injected at a random point along the trajectory. During the whole flight, including the manoeuvres after the failure, the camera and IMU data are stored to later shape the dataset for the training and testing of the FDD framework.

Figure 2 shows the three main blocks that shape the data gathering pipeline. Once the flight is concluded by achieving one of the terminating conditions, the environment and UFO are reset to their original state and the cycle is repeated. The loop continues for as many flights as it is desired for building the dataset.

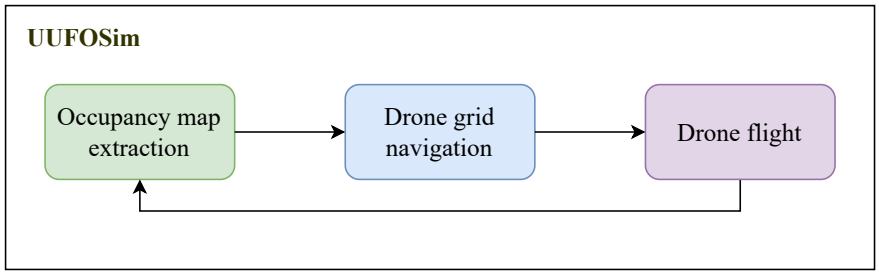

Fig. 2 Data gathering pipeline block diagram

In order to discuss in detail each of the presented blocks, this section will first present how the information from the environment is extracted offline and translated to an occupancy grid in subsection III.A. Once the environment state is known, the path that the drone should follow is computed in subsection III.B using common robot path planning algorithms. Finally, the drone flight including the sensor data collection and fault injection during flight are discussed in subsection III.C.

A. Environment and occupancy map

The environment is discretised into a matrix of inter-independent fix size cells which store whether they are free or occupied in the form of a boolean. This form of representation is called a grid occupancy map and has been exploited in the autonomous driving industry [35, 36], and more recently in the (flying) robotics sector [37, 38], to reduce the environment information to a tractable and efficient data structure.

Figure 3 describes all the steps taken to build a static 2D grid occupancy map in order to encapsulate all the information about the static obstacles found by the drone at its flying altitude prior to executing its flight.

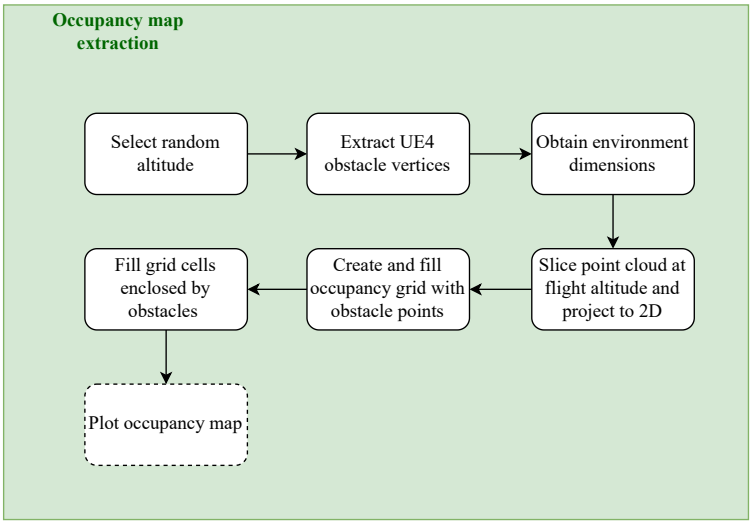

Fig. 3 Occupancy map extraction block diagram.

First, the flight altitude is randomly selected within a range to avoid overfitting of the FDD algorithm to object instances found only at certain heights (bushes, trees, windows) and horizon line locations in drone images. Next, for obstacle information extraction, AirSim's API returns 3D point clouds based on UE4 assets' triangular static meshes (see Fig. 4 for a sphere) in which each point has a label for an object in the environment.

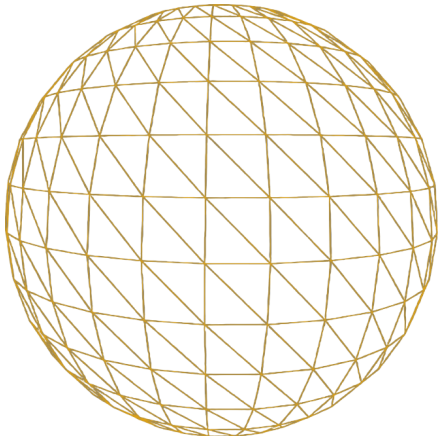

Fig. 4 Sphere mesh in Unreal Engine 4.

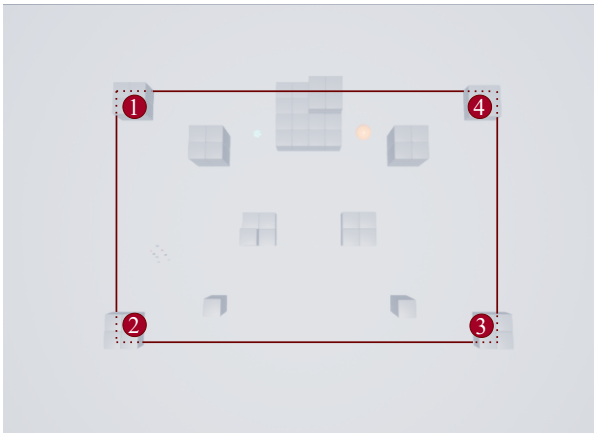

Fig. 5 Blocks environment limits for drone flight bounded by the 4 monoliths in the red rectangle corners.

Figure 5 shows the Blocks environment that will be used to showcase the following steps carried out on the grid occupancy map. It consists of grey blocks, an orange sphere, a blue cone and multiple cylinders. To ensure drone navigation around obstacles, flight boundaries are defined using the obstacles' farthest x and y coordinates. For the Blocks environment, the drone flights are confined to the red rectangle shaped by the four corner monoliths, resulting in a 3D point cloud from the obstacles' meshes of 46,248 points.

Furthermore, maintaining a constant flight altitude enables the reduction of the point cloud by slicing it within a specific altitude range and projecting 3D points onto a 2D x-y plane. Fig. 6 presents the Blocks' 2D point cloud at a drone altitude of seven meters with a three-meter range. The points that are close to each other with the same colour are part of the same object. This figure contains 8,956 2D points due to overlapping projections from various altitudes. This abundance of points can be seen when zooming to the orange blob in Fig. 7. Despite a fivefold reduction in obstacle points (and an eightfold reduction in information by discarding altitude data), many points lack meaningful information, with only the outer edges of object groups preserving vital obstacle representations.

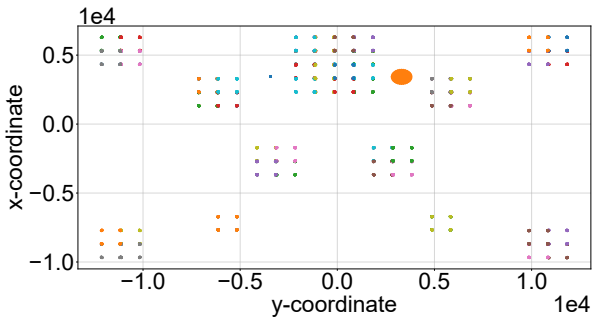

Fig. 6 2D projection of the Blocks environment object vertices within 4 and 10 metres altitude.

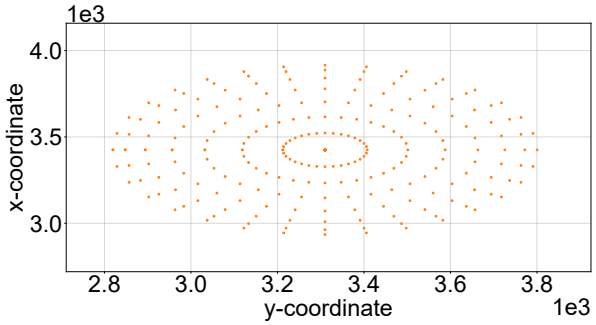

Fig. 7 Zoom-in of the 2D projected points of the sphere.

To address the desire of discarding unnecessary points, the concept of an occupancy map is introduced. The environment is divided into cells, and cells with points inside are marked as occupied (black), whereas those without any points are considered empty (white). This discretisation simplifies the data, reducing multiple points within a cell to a single data point. In the case of the Blocks example, the 2D point cloud is transformed from the world coordinate

frame to the grid coordinate frame, projecting all points onto a 2D grid and filling cells occupied by obstacle points, as shown in Fig. 8 and Fig. 9. The FDD data gathering pipeline then operates exclusively on this grid information, reducing the stored data and computational load. Instead of the initial 8,956 2D points, now a grid of 80 by 54 cells is used, totalling 4,320 cells. Since the data points (cells) are homogeneously distributed, their individual locations need not be stored, as long as the dimensions of the occupancy grid (x and y) are known. This transformation results in a 1D data stream, reducing the number of data points by a factor of two and the stored information by a factor of four. Refer to Table 1 for a summary of the evolution of the number of points and coordinates (information) through the projection and occupancy grid stages.

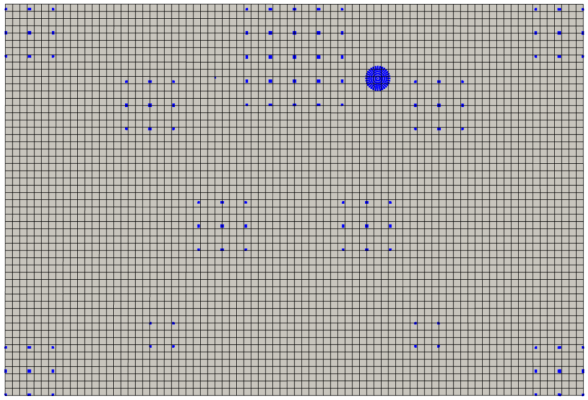

Fig. 8 2D points projected in empty occupancy grid.

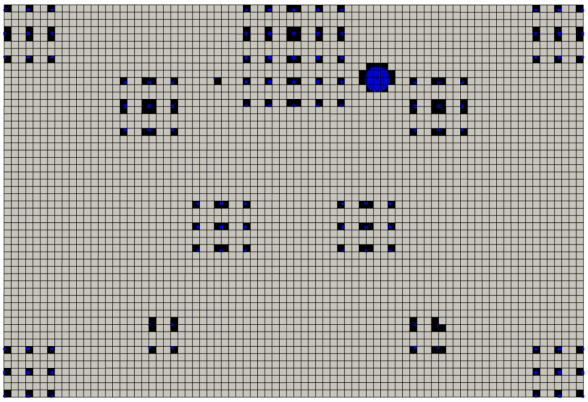

Fig. 9 Filled occupancy grid and 2D projected points.

Table 1 Evolution of the number of points and coordinates upon the occupancy map generation.

	Original	2D projection	Occupancy map
Points	46,248 (100%)	8,956 (19.36%)	4,320 (9.34%)
Coordinates	138,744 (100%)	17,912 (12.91%)	4,320 (3.11%)

Fig. 9 illustrates the limitation of merely filling grid cells occupied by obstacle vertices when creating an occupancy map. There are grid cells that lie within objects, that should not be accessible but that are not marked as occupied since there is no vertex of the static mesh on that particular cell. This problem becomes more pronounced with finer mesh resolutions. To address this, an algorithm was developed utilising Delaunay triangulation:

1) Objects in the environment are assigned coordinates of grid cells occupied by their 2D points on the occupancy map. Objects with fewer than three grid coordinates or forming linear shapes are excluded.
2) Delaunay triangulation creates a triangular mesh of each object's grid cells. Afterwards, the algorithm iterates over all the edges of each of the object's triangles, and if an edge is covered more than once, it is considered an internal edge shared by two triangles and is subsequently discarded. The remaining edges form the obstacle's outer polygon boundary.
3) All empty grid coordinates are evaluated to determine if they lie within each obstacle's polygon boundaries, marking grid cells within obstacles as occupied. See Fig. 10 for the Blocks environment. This occupancy map is then utilised by the path planning module.

B. Path planning

The steps to achieve a smooth drone flight path can be observed in Fig. 12.

1. Start and goal selection

First, random initial and goal flight coordinates are generated. These locations are subject to three design requirements:

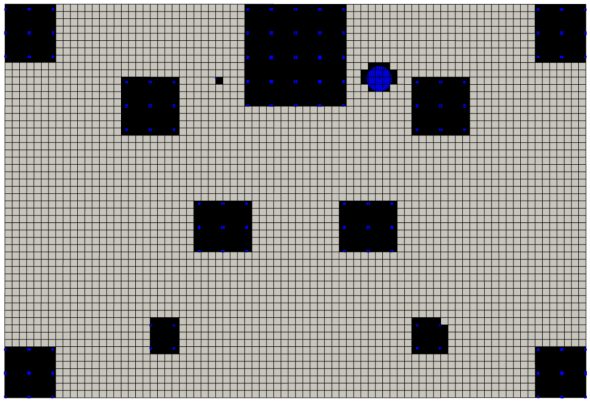

Fig. 10 Filled occupancy grid considering obstacle inner cells identified with Delaunay triangulation.

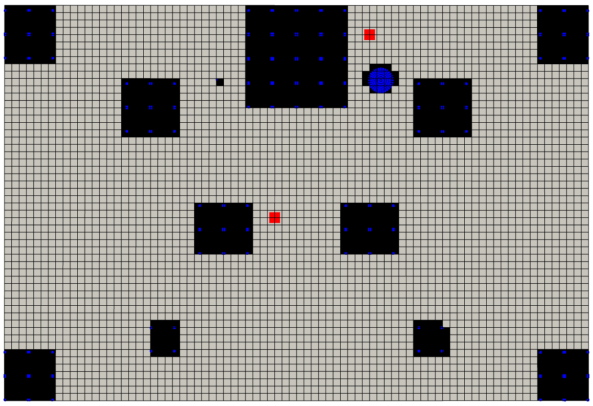

Fig. 11 Occupancy grid with start and goal locations.

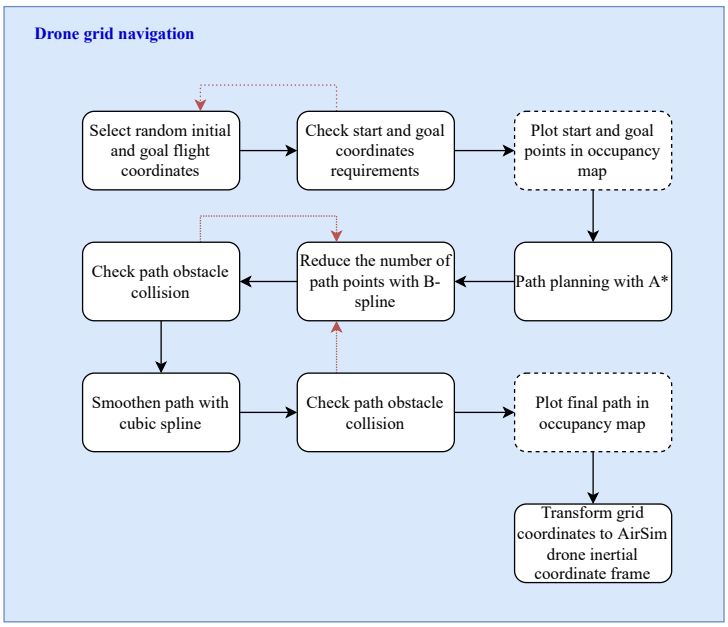

Fig. 12 Drone grid navigation block diagram.

1) The start and end locations must be separated by a distance greater than a user-defined minimum distance. This prevents extremely short paths that cannot accommodate the injection of failures.
2) The distance between the start and end locations should not exceed a user-defined maximum distance. This restriction ensures that excessively long paths, which do not significantly contribute to the training of the FDD framework and decrease the number of flights that would be executed in an allocated time, are avoided.
3) The start and goal locations must be positioned at a minimum distance from all identified obstacles in the occupancy map. This constraint guarantees the absence of unexpected collisions due to the drone's dimensions.

The random selection and requirement check are iteratively performed until the start and goal coordinates meet these established criteria. Once these conditions are satisfied, the coordinates are integrated into the occupancy map. For the Blocks example, the start and goal locations are depicted as red dots inFigure 11.

2. Path planning algorithm selection

Only two classic robot path planning methods are considered: grid (discrete) approaches and road-map methods. Figure 13 visually categorises the algorithms under investigation. Within the grid approach, both methods consist of

two steps: propagation (search) and back propagation. The primary distinction is that the Wavefront Path Planner conducts its initial search across the entire grid, whereas A* uses a function to prioritise cells for inspection based on the discovered solution space. As a result, even though each iteration in A* is more computationally expensive, fewer iterations are required because only a portion of the grid is examined.

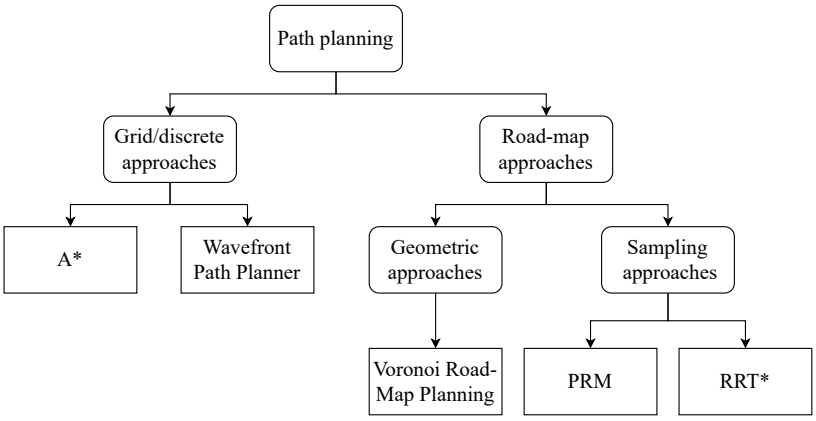

Fig. 13 Path planning algorithm classification

Regarding sampling techniques, PRM necessitates the drone to pass through randomly sampled points in the solution space, whereas RRT* only requires movement in their direction without point-to-point convergence.

All the approaches can maintain a configurable safe distance from obstacles, except for Voronoi Road-Map planning, which seeks to maximise this distance. In the Blocks environment, A* exhibits the lowest computation time, followed by Voronoi Road-Map planning. Wavefront Path Planner and PRM show similar computation times, whereas RRT* is the least efficient, requiring 17 times more time than A*. Consequently, the A* algorithm is the preferred choice. Figure 14 illustrates the A* path for the Blocks example.

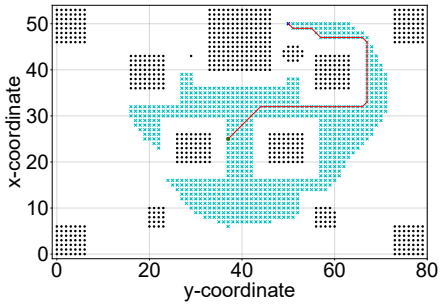

Fig. 14 A* Path Planner applied to the Blocks environment example.

3. B-spline path point number reduction

In shaping the final flight path, certain points bear no valuable information. Straight-line points can be reduced to two and some curves could be avoided if straight lines were taken. Whereas A* determines a flight path using many points spaced by an occupancy grid cell size, B-spline helps reduce it to its most indispensable points.

B-splines [39] of order k are piecewise polynomials used as the basis for spline functions, they generate smooth trajectories connecting data points and are of degree k-1. In this study, B-splines of degree 2 (order 3) are employed for optimising paths. The goal is to minimise the number of points, beginning with only 5% of the initial number. If unsuccessful, the number of points is increased by 5% in each iteration. A 100% point retention indicates no reduction is possible, and the A* generated path proceeds to the next step in path planning.

Reduced paths are checked for collisions with environmental obstacles. Vectors connecting path points are discretised, and grid cells they traverse are examined for occupancy. Additionally, this process is repeated with two parallel vectors, each shifted one cell to the right and left of the original vector, ensuring a minimum one-cell distance from obstacles. If

any of these checks detect an obstacle, it is treated as an unsuccessful point reduction, and the percentage of retained points is increased by 5%. A new B-spline is then generated, and the reduced path is re-evaluated for obstacles. In Fig. 15, even though the central vector connects the two path points through open space, an obstacle is detected because one of the points of the right parallel vector can be found within an occupied grid cell.

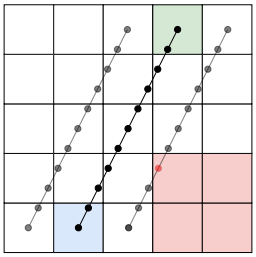

Fig. 15 B-spline reduced path obstacle detection. The blue and green cells are two path points, and the red cells are occupied by environment obstacles. The black lines are the discretised vectors for obstacle detection.

The advantages of reducing path points using B-splines are evident in the occupancy grid. Figures 16 and 17 display the path before and after point reduction. Fig.16 mirrors the path in Fig.14, comprising 61 points. In contrast, Fig. 17 achieves the same path with just 9 points

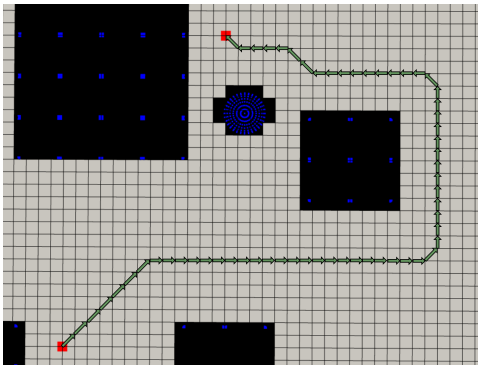

Fig. 16 Zoom-in of Blocks occupancy map with A* flight path represented by small green arrows.

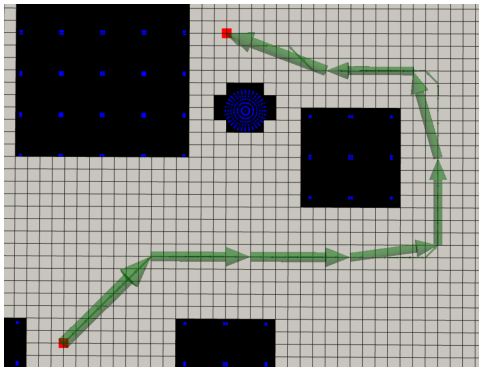

Fig. 17 Zoom-in of Blocks occupancy map with B-spline reduced flight path represented by large green arrows.

4. Cubic spline path smoothing

The B-spline method enabled the identification of pivotal points that serve as a foundation for cubic smoothing splines. Smoothing would have been proved ineffective if applied on the A* flight path due to its fine path discretisation in single occupancy grid cells.

This smoothing is essential to eliminate sharp corners. In the final smooth flight path, points are still separated by one occupancy grid cell's size, but they are no longer constrained to cell corners as in the A* and B-spline reduced paths. After smoothing, collision checks are performed using the same approach as with the B-spline.

Figure 18 displays the final flight path marked by red arrows, whereas Fig. 19 visually confirms that the cubic spline intersects the pivot points (yellow circles), with flight path points unrestricted to grid cell boundaries. Notably, a marked improvement resulting from the combination of B-splines and cubic splines (small red arrows) is evident when compared to the original A* flight path (small green arrows).

5. Flight path transformation to AirSim drone inertial coordinate frame

Finally, the grid coordinates of the smoothed spline flight path are transformed to the AirSim drone's inertial coordinate frame. Four distinct inertial coordinate frames are essential in this process:

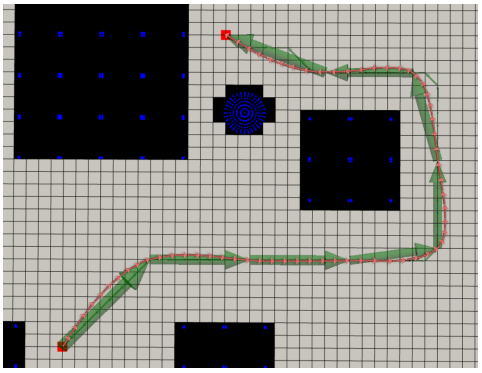

Fig. 18 Zoom-in of Blocks occupancy map with cubic spline smoothed flight path represented by small red arrows.

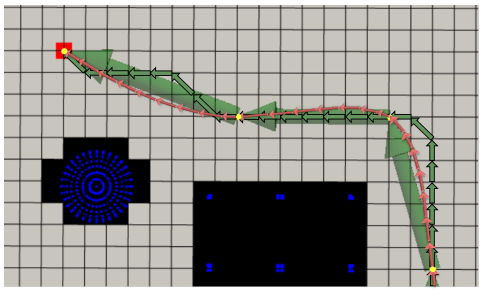

Fig. 19 Visual confirmation of the final path not being constrained to the occupancy grid and it passes through B-spline pivot points (yellow circles).

1) **Unreal Engine 4 inertial coordinate frame (ICF_{UE4})**: used to build the UE4 environment. Hence, the centre of coordinates and the direction of its axes vary from map to map. It is a drone-independent coordinate frame.
2) **Occupancy grid inertial coordinate frame (ICF_{OG})**: it has its origin at the bottom left of the occupancy map, with the y-axis pointing to the right and the x-axis pointing to the top in the 2D grid. Objects and points in the occupancy map have positive coordinates. The environment has been discretised with cells of predefined size, C_{UE4}. As can be seen in Eq. (1) and Eq. (2), in order to define the number of cells in the grid (n_x and n_y), the minimum and maximum X- and Y-coordinates among all the obstacles in UE4 environment ($x_{UE4_{min}}$, $x_{UE4_{max}}$, $y_{UE4_{min}}$, $y_{UE4_{max}}$) are required. It is a drone-independent coordinate frame.

$$n_x = (x_{UE4_{max}} - x_{UE4_{min}})/C_{UE4} \tag{1}$$

$$n_y = (y_{UE4_{max}} - y_{UE4_{min}})/C_{UE4} \tag{2}$$

3) **AirSim inertial coordinate frame (ICF_{AS})**: shares the same origin as ICF_{UE4}, with axes aligned similarly, but the units are scaled differently. In this frame, 1 unit is equivalent to 100 units in ICF_{UE4}. This factor is defined as k_{UE4}. It is a drone-independent coordinate frame.
4) **AirSim drone inertial coordinate frame (ICF_{ASD})**: identical to ICF_{AS}, but with its origin adjusted to the initial drone spawn location within the environment, expressed as x_{D_0} and y_{D_0} in ICF_{UE4}. The location of the drone within the controller is expressed using ICF_{ASD}. It is a drone-dependent coordinate frame.

To convert the flight path from the occupancy map to the AirSim drone inertial coordinate frame, the transformations depicted in Fig. 20 were utilised with the provided parameters.

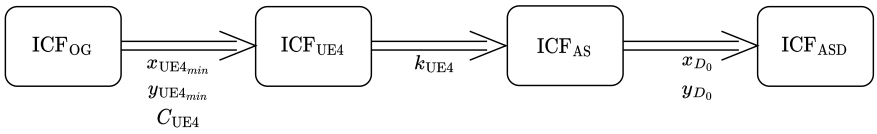

Fig. 20 Inertial coordinate frame transformations: from ICF_{OG} to ICF_{ASD}

Given a trajectory T: $[(p_{1_x}, p_{1_y}), (p_{2_x}, p_{2_y}), ..., (p_{n_x}, p_{n_y})]$ of n points in $\mathbb{R}^2$ in ICF_{OG}, the points of the flight path can be transformed to ICF_{ASD} ($\bar{p}_{i_x}$ and $\bar{p}_{i_y}$) with Eq. (3) and Eq. (4).

$$\bar{p}_{i_x} = (p_{1_x} \cdot C_{UE4} + x_{UE4_{min}})/k_{UE4} - x_{D_0} \quad i = 1, 2, ..., n \tag{3}$$

$$\bar{p}_{i_y} = (p_{1_y} \cdot C_{UE4} + y_{UE4_{min}})/k_{UE4} - y_{D_0} \quad i = 1, 2, ..., n \tag{4}$$

C. Data collection

The next step is to fly the drone within the UE4 environment, potentially induce an actuator failure and gather all the vision-based and signal data for the FDD training. Figure 21 summarises all the steps taken during the final block of the data gathering pipeline. In the following sections, each of the blocks will be briefly discussed.

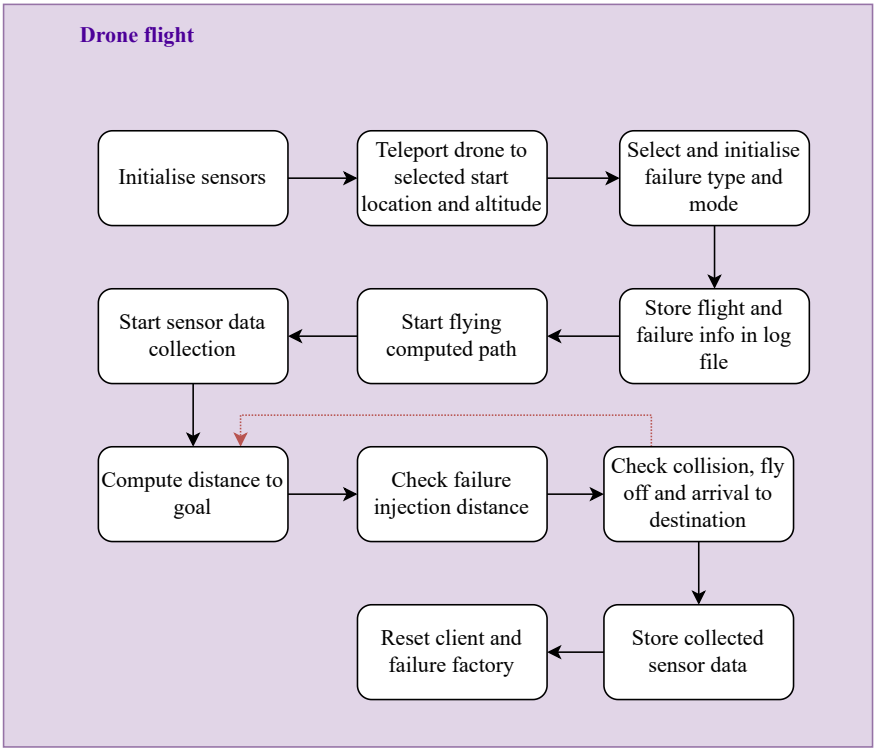

Fig. 21 Drone flight block diagram

1. Sensor initialisation and drone teleportation

In the sensor initialisation stage, data structures for IMU and camera data are created. A single drone can carry multiple cameras, each providing different information like depth, segmentation, or RGB.

After initialising the sensors, the drone is teleported to the start location with the heading set towards the first trajectory point. Users can choose whether the drone should take off from the ground or start at a specified altitude. In our research, focused on cruise phase failures, the drone is teleported to the desired altitude, as shown in Fig. 22 in the Blocks environment.

Fig. 22 Drone teleported to start location

2. Failure type & mode selection and initialisation

In this study, four actuator failure types are considered: actuator saturation, actuator lock, propeller fly-off, and propeller damage. Actuator saturation locks the propeller at its maximum rotational rate, whereas actuator lock fixes it at a percentage of this maximum. In the case of propeller fly-off, the propeller is detached completely, and for propeller damage, one or more blades are broken. The first three failure types are simulated by providing predefined locked values to the propeller's rotational rate.

To simulate propeller damage, the Blade Element Theory (BET) model presented in [34] was implemented as a plug-in to the nominal vehicle physics model. Blade damage requires a closer look at the propeller aerodynamics and centre of gravity shift, which create a loss of thrust and vibrations along the three body axis, as demonstrated in Fig. 23 and Fig. 24, which show the BET-simulated forces and moments for the Bebop 2 drone front left propeller after suffering 20% damage while rotating at 600 rad/s.

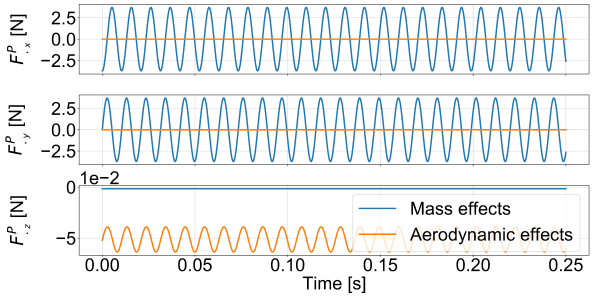

Fig. 23 BET-simulated evolution of mass and aerodynamic forces generated by lost blade sections upon 20% Bebop 2 blade damage for 0.25 s rotating at ω_0 = 600 rad/s [34].

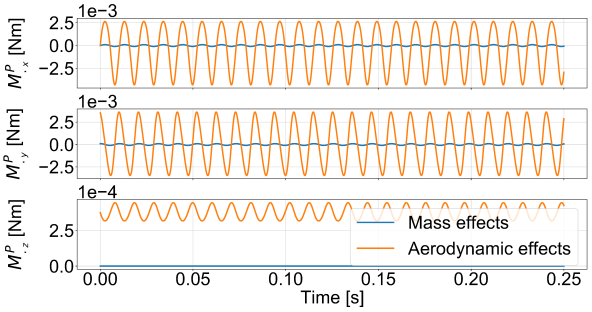

Fig. 24 BET-simulated evolution of mass and aerodynamic moments generated by lost blade sections upon 20% Bebop 2 blade damage for 0.25 s rotating at ω_0 = 600 rad/s [34].

Each failure type has a varying number of failure modes. For example, propeller fly-off has four failure modes, one for each propeller. Two hyper-parameters were introduced: discrete vs. continuous and abrupt vs. linear. Discrete values are selected from the list <0.2, 0.4, 0.6, 0.8>, whereas continuous values lie within the open interval (0,1). In the abrupt case, failure occurs instantaneously, whereas in the linear case, it transitions linearly from nominal to the fault state.

To create a balanced dataset for FDD algorithm training, users select failure types and modes. A pool of potential combinations is created and uniformly sampled before each flight. For example, if the user selects discrete and abrupt actuator saturation and actuator lock, the algorithm samples from 21 alternatives, including four actuator saturation options (one for each propeller), 16 actuator lock options (four per propeller), and one healthy option.

Before each flight, a random distance along the planned trajectory is chosen for failure injection, at least five meters from the start and goal locations, to avoid capturing transient effects at flight initialisation and completion.

3. Drone flight: guidance, control and physics model

The drone follows the computed path in a three-step loop (Fig. 25). Firstly, the guidance block generates reference position (x_r, y_r, z_r) and heading (ψ_r) for the controller based on the planned path and current vehicle states ($\vec{X}$), accounting for measurement noise (ν). Secondly, the controller translates these references into actuator rotation velocities (ω_{ic}| i=1,2,3,4). Finally, the physics model simulates the drone's response to these commands and provides the next time step's vehicle states as measured by the vehicle sensors.

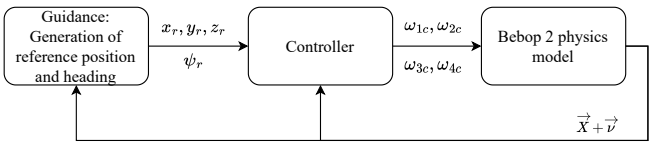

Fig. 25 Drone flight guidance, controller and physics model pipeline.

Until now, the simulation pipeline is platform agnostic. However, to showcase the FDD framework, the Bebop 2 drone is chosen for this paper. In order to simulate as close as possible to reality its behaviour, the authors integrated the Incremental Nonlinear Dynamic Inversion (INDI) controller [7] and grey-box physics model [11, 33] from Delft University of Technology using AirSim's C++ API. Additionally, the existing AirSim guidance approach was leveraged.

For numerical integration, the simulation employs the Beeman and Schofield explicit method [40] outlined in Eq. (5) and Eq. (6), where x is the position or attitude, v is the linear or angular velocity and a is the linear or angular acceleration. Additionally, Δt is the time step duration and the subscripts n+1, n and n-1 refer to the next, current and previous time steps.

$$x_{n+1} = x_n + v_n \Delta t + \frac{1}{6}\left(4a_n - a_{n-1}\right)\Delta t^2 \qquad (5)$$

$$v_{n+1} = v_n + \frac{1}{6}\left(2a_{n+1} + 5a_n - a_{n-1}\right)\Delta t \qquad (6)$$

4. Sensor data collection, failure injection and flight termination

During flight execution, sensor data could be collected through AirSim's Python API default functions. However, these functions have a maximum sampling rate of 26 Hz in the basic Blocks environment, falling short of the desired rates for IMU (500-1000 Hz) and camera (30-60 Hz). Additionally, this sampling rate fluctuates during flight due to simulator workload and Python API computations, with variations of up to a factor of 2.3.

To boost IMU sampling rates, its data collection has been coupled to the simulator's physics engine in C++. When the flight ends, the data are retrieved by the Python API and the C++ vectors are cleared for the next flight. This approach is also applied to other signal-based sensors like the barometer, magnetometer, and GPS. Users can select the sensors to activate and their sampling rates.

An advantage of tying IMU data collection to the physics engine is immunity to simulation slowdowns. However, the sampling frequencies are limited to integer factors of the physics engine thread calling rate. With the nominal physics engine thread calling period of 0.003 seconds (f_p = 333.33 Hz), the sampling frequencies available for IMU data gathering are discrete and limited to values of f_p divided by integer values.

For video sampling rates, it is not possible to couple the image data storage to the physics engine because the AirSim image retrieval functions are part of another simulator thread. Therefore, the simulation clock speed is modified to achieve a higher number of frames per second (fps). This, in turn, affects the physics engine and the IMU sampling rates. For instance, using a clockspeed factor of 0.5 would allow an IMU sampling rate of up to 666.66 Hz. Therefore, a trade-off must be made between IMU and camera sampling rates by tuning the clockspeed factor.

Concurrently, as data are collected, the drone's proximity to the goal location is monitored. When the designated failure point along the trajectory is reached, the Python API triggers the chosen failure. For actuator-related failures, the damage coefficient is adjusted, whereas for blade damage, forces and moments from lost blade sections are added to the healthy aerodynamic model. The timing of these changes depends on whether the failure is abrupt or continuous.

Once the failure has been injected, the simulation pipeline starts to check whether any of the following flight termination conditions has been reached: collision with the ground, collision with an obstacle, drone flies above a predefined altitude or timeout.

5. Flight & failure metadata logging and sensor data storage

Once the flight has been concluded, flight and failure metadata, such as the type, location and magnitude of the failure, are stored for the posterior labelling of the gathered data for the training and testing of FDD algorithms. The data recorded by the IMU at every time step of the flight are stored in a .csv file. All the camera frames are stored in the same directory as the IMU, each image with the name of the timestamp at which it was taken in order to preserve the temporal sequence information.

Finally, the client, sensors and failure factory are reset to their original values in order to repeat the complete data gathering pipeline shown in Fig. 2 for the next flight.

IV. Fault detection and diagnosis framework

This section introduces an FDD framework designed to detect drone actuator failures, point to the failed actuator and quantify the damage. The architecture proposed here fuses data from the Inertial Measurement Unit (IMU) and the on-board camera. Thanks to the simulator developed in section III, it is possible to use knowledge-based approaches previously impossible due to the lack of data.

The complete FDD architecture can be observed in Fig. 26. It comprises two independent parallel data processing paths for the camera and IMU, followed by feature fusion using a Long Short-Term Memory (LSTM) network, architecture with feedback connections which allows the ingestion of sequential data. Subsequently, a dense Neural Network is employed for data classification, with the number of output neurons aligning with distinct classes. For instance, in the context of failure detection, there are two classes: 'healthy' and 'failure,' resulting in two output neurons.

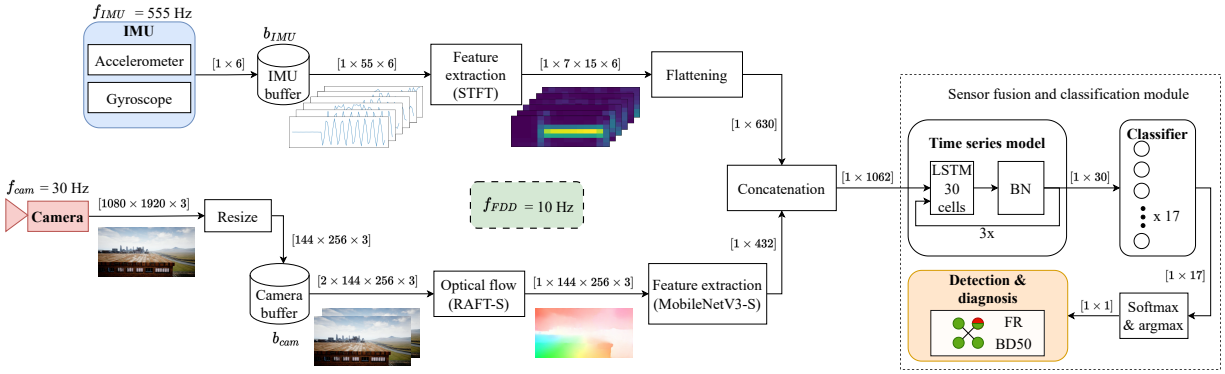

Fig. 26 The FDD pipeline consists of (i) an IMU time-frequency feature extractor in the form of a Short-Time Fourier Transform, (ii) the MobileNetV3-S as feature extractor from the camera optical flow computed with RAFT-S and (iii) a Long Short-Term Memory network followed by a single layer Neural Network as sensor fusion and classification module. The FDD framework is run at 10 Hz and the sampling rate of the IMU and camera are 555 Hz and 30 Hz, respectively.

Synchronising data from sources with varying sampling rates without discarding precious data can present a challenge for FDD architectures. On drones, the IMU often samples data at a much higher rate than the camera. Rather than naively synchronising data at the camera's frequency and discarding the IMU data between camera shots, the presented FDD architecture accommodates flexible operation without data loss at a commanded frequency, provided it does not exceed the slowest sensor's rate. This adaptability accommodates diverse computation constraints.

In the following sections, a detailed exposition of each architecture component is provided, specifically tailored to the Bebop 2 drone's characteristics (frequencies and tensor sizes) used in our research. First, subsection IV.A delves into image feature extraction. Then, subsection IV.B details the IMU data processing. Finally, subsection IV.C shows the fusion of sensor features for detection and diagnosis prediction, treating the tasks as classification problems.

A. Camera data processing

The inspiration for the introduction of the camera into the FDD pipeline stems from the observation that human beings are able to detect that they are falling thanks to their "natural time differentiated accelerometer" or vestibular system, an apparatus within the inner ear that provides information about changes in acceleration, as well as from their visual sensory system. When the vestibular system is saturated (e.g. rapidly rotating on an office chair) or the changes in acceleration are imperceptible (e.g. accumulating slow changes in aircraft attitude), the visual sensory system is still able to detect the subject's ego motion thanks to the relative movement of elements of the environment in its visual field. For instance, if a human subject sees a block moving to the right in a static environment, the subject understands that it is moving then to the left.

The two main factors affecting judgement of self-motion are the gradients and the pattern of optical flow which provide information about the relative velocity (amount) and direction of relative motion, respectively. Hence, the authors believe that knowledge about the magnitude and direction of the optical flow could enhance the diagnosis component of the FDD framework by implicitly quantifying the failure magnitude and identifying the failed actuator. For instance, if the front right clockwise rotating (from top view) propeller is lost, then it is expected that the drone will lose lift, tilt forward and rotate clockwise. In optical flow, this should translate to a vector field with an up-left direction. The stronger the gradient, the greater the failure magnitude.

There are two ways in which optical flow can be represented, namely sparse and dense optical flow [41]. The main difference is that the first computes the optical flow for a predetermined number of features of interest whereas the second computes it for the complete frame. Even though the sparse optical flow is less computationally expensive, it has

two main problems. First, those features of interest may disappear or become hidden after a few frames, forcing the optical flow approach to select new features. Second, the algorithm may choose different features between frames as some become more salient than others throughout time. As a result, it is difficult to infer a potential actuator failure from a specific optical flow change pattern as it could be attributed to the tracking of different features over consecutive frames. Hence, dense optical flow was chosen.

In literature there are two main classes of dense optical flow approaches, namely traditional or classical energy-based and deep-learning based. In recent years, deep learning based approaches have been able to surpass the traditional counterparts in accuracy and lower inference times, allowing them to run in real time and becoming the de facto choice for computationally constrained devices and platforms [42, 43].

Within this deep-learning based approaches there are three architectures that stand out from literature for their high accuracy and low inference time, while providing their code and trained model weights:

- CNNs for Optical Flow using Pyramid, Warping, and Cost Volume (PWC-NET[†]) [44]. It was published in June 2018, one of the fastest methods in literature and the fastest from the selection; it is considered a milestone algorithm in the field [45].
- Recurrent All-Pairs Field Transforms for Optical Flow (RAFT[‡]) [46]. It was published in November 2020 and it shows the highest performance of the three considered approaches in the MPI-Sintel dataset with the highest reported inference time [46].
- Displacement-Invariant Matching Cost Learning for Accurate Optical Flow Estimation (DICL-Flow[§]) [47]. It was published in December 2020 and it shows a reported runtime and performance between the PWC-NET and RAFT approaches.

Next to them, two more approaches were taken into consideration: a classical approach for comparison, namely Gunnar Farneback's algorithm [48] developed in 2003, and a small pre-trained RAFT model (RAFT-S) implemented within the Torchvision library. In contrast with the original RAFT model, it contains five times less parameters but it maintains superior performance when compared to the PWC-NET and, in some instances, DICL-Flow models.

Unfortunately, it is not clear whether the inference times reported in literature were obtained from systems with similar compute specifications. To compare these approaches, they were executed on three datasets collected with UUFOSim at different image resolutions, resulting in the inference times shown in Table 2. Each time value is the average that each algorithm took to predict the optical flow for 250 frames on a laptop with a 6 core Intel Core i7-9750H CPU, 16 GB of RAM DDR4 and an NVIDIA Quadro P2000 with 5 GB of GDDR5 memory. As can be seen, even though DICL-Flow is the intermediate option from literature, it presents the worst inference time for all resolutions.

Table 2 Inference time of dense optical flow approaches on the UUFOSim dataset at different resolutions.

Methods	**256×144**	**512×288**	**1024×576**
	(s)	(s)	(s)
PWC-NET	0.073	0.143	0.423
RAFT	0.17	0.17	0.36
DICL-Flow	0.274	0.296	0.617
RAFT-S	0.06	0.10	0.35
Farneback	0.008	0.042	0.177

Figure 27 allows for a visual comparison of the approaches' optical flow quality with a frame from the 1024×576 dataset. As can be seen in Fig. 27b, even though PWC-NET has the lowest run time among the deep learning approaches, its optical flow prediction is very noisy without any recognisable features, indicating a poor cross-dataset generalisation. Furthermore, from Fig. 27f it can be seen that Farneback does not perceive slight movements. Most of the pixels are black, leading to the loss of potential features (pixels) that could serve as rich sources of information further down the FDD pipeline. Besides that, a strong flickering behaviour has been observed in Farneback's optical flow over multiple frames, which hints to unreliable predictions.

[†]https://github.com/philferriere/tfoptflow
[‡]https://github.com/princeton-vl/RAFT
[§]https://github.com/jytime/DICL-Flow

(a) Original (b) PWC-NET (c) RAFT (d) DICL-Flow (e) RAFT-S (f) Farneback

Fig. 27 Dense optical flow visual quality comparison.

Given the high inference time of DICL-Flow with the collected dataset and the low visual quality of PWC-NET and Farneback, the two remaining options for optical flow computation are RAFT and RAFT-S. As both show similar visual quality and RAFT-S has a run time three times lower than its larger version for the lowest resolution, RAFT-S is chosen as the optical flow module of the FDD pipeline.

Returning the attention to Fig. 26, the bottom information path shows the camera data processing. In the case of the Bebop 2 drone, the camera captures images at 1080p, meaning frames of 1080 pixels in height and 1920 in width with three RGB channels [1080×1920×3], and they are resized to a tensor of dimensions [144×256×3] before being stored in the camera buffer (b_{cam}). Then, at every time step at which the FDD framework is executed, the b_{cam} contains $f_{\text{cam}}/f_{\text{FDD}} + 1$ samples, and the first and last entry of the buffer are passed on to the optical flow model. Here, f_{cam} stands for the fps rate at which the drone collects image data and f_{FDD} is the frequency at which the FDD pipeline is executed on board of the drone. Next, the buffer is emptied except for the last stored image, which remains in memory for the next FDD time step. This ensures the temporal coherence of the optical flow over multiple FDD calls.

Once RAFT-S computes the optical flow, the output tensor is fed to a feature extractor. For this part of the pipeline, the authors opted for transfer learning. The model of choice was the backbone of MobileNetV3-Small [49] with frozen weights pre-trained on the ImageNet dataset [50] because it has the lowest inference time among all keras pre-trained models[¶] at the time of writing. A depth multiplier (alpha) of 0.75 was set in order to proportionally decrease the number of filters in each layer, achieving a reduction in the number of parameters from 2.9 to 2.4 million (3 ms of inference time). Finally, the last layer of MobileNetV3-Small is set to be a global average pooling layer which collapses the width and height of the output tensor to a single feature, resulting in a 1D tensor of 432 features.

B. IMU data processing

From the IMU, the FDD algorithm receives six 1D data streams, namely the linear acceleration and the angular velocity in the x, y and z directions. Two key signal features that contribute to the detection and classification of these failures are the evolution of their bias through time and the amplitude of their oscillations; the latter especially in the case of blade damage, as highlighted in [34]. Information about both features can be encapsulated in their Short-Time Fourier Transform (STFT), creating compact time-frequency maps or spectrograms and removing potential sensor noise. To illustrate this, Fig. 28 shows the accelerometer signal in the x-direction and its spectogram for a random flight within the dataset which experienced a blade damage failure of 0.8, 6.83 seconds after the start (as highlighted by the red dashed vertical line). As can be seen, failure can easily be detected by the sudden appearance of signal content at high frequencies, in this case between 173 and 186 Hz.

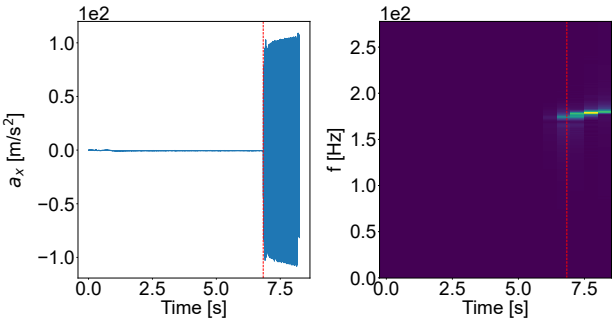

Fig. 28 UAV x-axis acceleration and its spectrogram. The dashed red vertical line denotes the time of failure.

[¶]https://keras.io/api/applications/

From the IMU information path shown in the upper half of Fig. 26, the incoming data from the accelerometer and the gyroscope is stored in a buffer (b_{IMU}). Once the FDD module is called, the buffer is emptied and its data are used for computing the STFT. This form of frequency analysis is a windowed approach which divides the time signal into small equally sized segments and applies an independent Fourier transform to each one of them. Hence, there is a trade-off between the time and frequency resolutions; the wider the window the higher the frequency resolution at the expense of the time resolution. Since the STFT is applied to small sample sizes of $n_{\mathrm{seg}}=\lfloor f_{\mathrm{IMU}}/f_{\mathrm{FDD}} \rfloor$ at a time, a window size of $n_{\mathrm{win}}=\lfloor n_{\mathrm{seg}}/4 \rfloor$ is chosen with $o = \lfloor 3/4 n_{\mathrm{win}} \rfloor$ samples of overlap between windows, i.e. a stride of $\mathrm{s}=\lceil 1/4 n_{\mathrm{win}} \rceil$. The sample vector is padded such that the time resolution or the number of steps in which the time axis of the spectrogram is divided is $t_{\mathrm{re}}=\lceil n_{\mathrm{s}}/(n_{\mathrm{win}}\text{-}o) \rceil+1$. As can be seen, as n_{win} increases, t_{res} decreases. The opposite is observed in the frequency resolution $f_{\mathrm{res}}=\lfloor n_{\mathrm{win}}/2 \rfloor + 1$.

For the present research, f_{IMU} of the collected dataset and f_{FDD} approximately equal 555 Hz and 10 Hz, respectively. Hence, 55 samples are fed to the STFT at every FDD time step, which outputs a tensor of dimensions [7×15×6]. This means a frequency resolution of seven and a temporal resolution of 15. Figure 29 and Fig. 30 show the IMU signals and their STFTs for the same flight as in Fig. 28, using a time segment of 0.1 s (f_{FDD}=10 Hz) starting at 6.78 s in order to include the transition from a healthy to a failure state. Again, it can still be clearly observed when the blade damage has taken place for failure detection. Finally, the STFT output tensor is flattened to a single dimensional tensor of 630 features that will be fused with those coming from the camera data processing path of the pipeline.

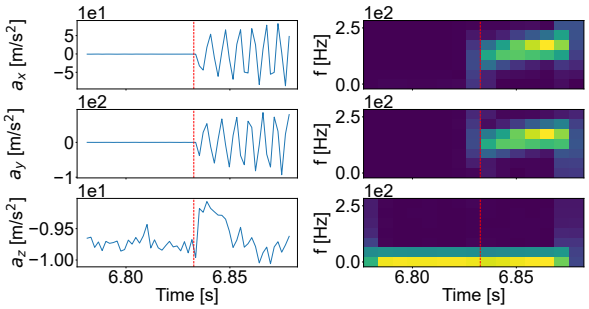

Fig. 29 Sample flight accelerometer signals and spectrograms for a 0.1 s time interval starting at 6.78 s.

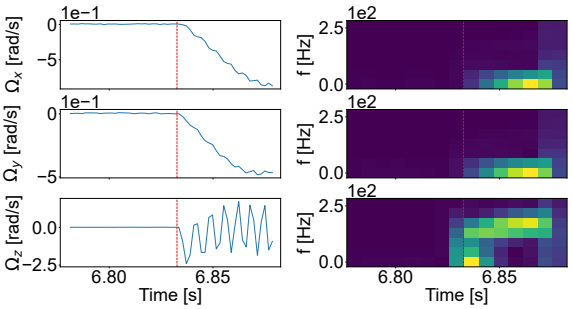

Fig. 30 Sample flight gyroscope signals and spectrograms for a 0.1 s time interval starting at 6.78 s.

C. Sensor fusion and classification module

As can be seen in Fig. 26, the features from the camera and the IMU are concatenated into a single vector of 1062 features and fed to a sequence-to-sequence LSTM model, which allows the FDD framework to take decisions based on current and previous data at every time step. LSTM cells have an internal state that stores information from an arbitrary number of previous inputs which, in conjunction with the current input, is used to extract sequential relationships to generate an output. For the present research, the time series model consists of a simple stack of three LSTM layers of 30 cells, each followed by a Batch Normalisation (BN) layer; transformation that maintains the mean and standard deviation of its input batch close to zero and one, respectively. At every FDD time step, an input vector of 1062 features is fed into the network which outputs a tensor of 30 features.

The last stage of the FDD pipeline is the classifier that will simultaneously perform the tasks of failure detection, failure magnitude quantification and failed propeller identification. The problem is simplified by considering each potential drone state, namely each failure mode and the healthy state, as a class. As an example, if abrupt actuator saturation and abrupt propeller fly-off are considered as the only modes of failure, then the classification layer would have to discern among nine classes, namely two failure classes per propeller and one for the healthy state. To perform this classification task, a single layer dense neural network layer (NN) is used with the number of neurons equal to the number of classes, followed by the softmax activation function in order to generate a multinomial probability distribution; the model outputs the probability it believes the input belongs to each class. The goal is that the highest probability is attributed to the correct failure or healthy drone state at each time during the flight.

Both the LSTM model and the classifier are the only two trainable components of the FDD pipeline, as the RAFT-S and MobileNetV3-S weights are frozen. For their training, the sparse categorical cross-entropy loss function and adam optimiser are used, both extensively exploited in literature for multi-class classification.

V. Results

A. UUFOSim dataset

The right clockspeed is a function of the image resolution; the larger the image, the lower the camera sampling rate at the same clockspeed. Therefore, a trade-off needs to be performed between sampling rate accuracy and simulation speed for the chosen image resolution of 256×144 (width×height). Figures 31 and 32 show 20 simulations at different clockspeeds and their camera and IMU sampling rates. It can be observed that a clockspeed of 0.6 has a large spread of camera sampling rates between flights and the clockspeeds of 0.4 and 0.5 have IMU sampling rates far below the desired 512 Hz. Since the remaining clockspeeds show similar performance, 0.3 was selected for being the fastest.

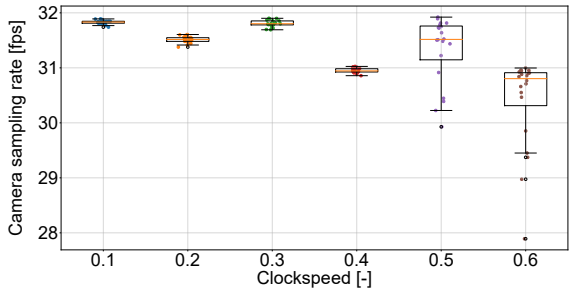

Fig. 31 Boxplot of the camera sampling rate for different clockspeeds with an image resolution of 256×144 pixels (width×height).

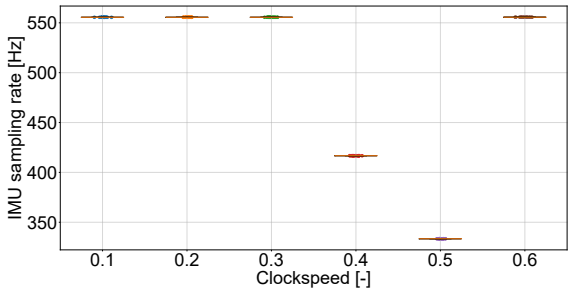

Fig. 32 Boxplot of the IMU sampling rate for different clockspeeds with an image resolution of 256×144 pixels (width×height).

The IMU sampling rate samples are almost constant at the same clockspeed because the data gathering of this sensor has been coupled with the simulator's physics model, as discussed in subsection III.C. In contrast, the camera sample rates are much more dispersed, especially the higher the clockspeed. For the same simulation time and slower clockspeed, the simulation checks the thread that receives the calls from the Python API more frequently. Hence, the frequency at which camera images can be called is higher, reducing the impact of simulation slow downs and, hence, the camera sampling rate dispersion.

5,000 flights were flown with a clockspeed of 0.3 and image resolution of 256×144. To verify that the camera and IMU sampling rate predictions estimated with 20 flights were accurate, the same box plot was created with the flown 5,000 flights. The results are shown in Fig. 33 and Fig. 34: the camera runs at 31.81 fps and the IMU has a sampling rate of 555.59 Hz.

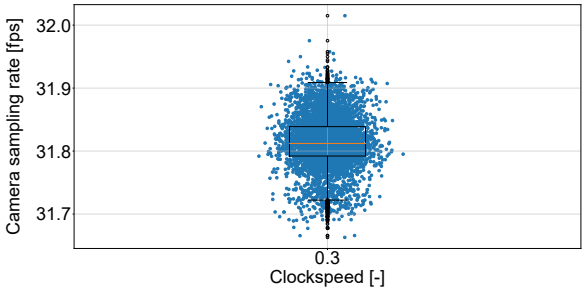

Fig. 33 Boxplot of the camera sampling rate for 5,000 flights with a clockspeed of 0.3 and an image resolution of 256×144 pixels (width×height).

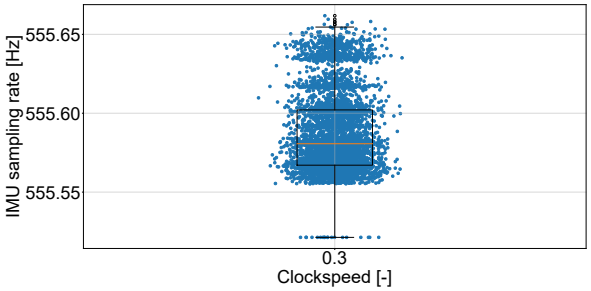

Fig. 34 Boxplot of the IMU sampling rate for 5,000 flights with a clockspeed of 0.3 and an image resolution of 256×144 pixels (width×height).

The simulation pipeline discussed in section III was run in a Windows OS PC with a 20 core Intel Xeon W-2255 CPU, 32 GB of RAM DDR4 and an NVIDIA RTX A4000 GPU with 16 GB of GDDR6 memory. The 5,000-flight dataset was collected in 61.67 hours and has a memory footprint of 239 GB. Only blade damage failures of 20%, 40%, 60% and 80% were simulated since those are the failure modes that will be used to train and test the FDD pipeline. A sample of frames from a single flight separated by 35 frames from each other can be observed in Fig. 35.

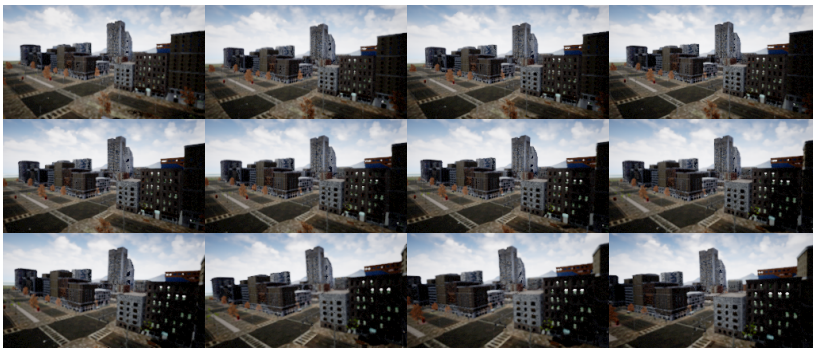

Fig. 35 Camera captured frames during single flight read from top to bottom and from left to right (only shown one every 35 frames).

B. Fault detection and diagnosis framework

To demonstrate the potential of the proposed FDD framework, only four modes of discrete failure were considered per propeller, namely 20%, 40%, 60% and 80% single abrupt blade damage. As a result, there are 17 classes among which the FDD pipeline should discern. For this purpose, the dataset was split into 70% training, 20% validation and 10% testing. Each flight of this dataset has a variable duration between 6 and 16.9 seconds with an average length of 11.6 seconds. The first second of every flight is ignored in order to avoid the acceleration transient after the flight has started. From the remaining flight time, single 5.5-second data snippets are used per flight in order to batch train and evaluate the pipeline with equal length data sequences without padding. Flights of length shorter than 6.5 seconds only constitute 0.72% of the total dataset and were eliminated. Besides that, flights that were not properly recorded in UE4 — the drone does not take off or the sensor data are not recorded at the correct rate — were also removed. At the end, the training dataset consisted of 3,468 5.5-second flights.

Table 3 shows the results for the pipeline presented in section IV in terms of inference time and test accuracy. The runtime was obtained from the same compute setup that was used to generate Table 2. Furthermore, three different types of test accuracy are considered, namely general, detection and diagnosis. The first refers to the accuracy outputted by the model. The second is obtained by lumping the failure classes 2 to 17 into a single class and computing the resulting accuracy. This means that a prediction of a data point whose ground truth is a failure class is deemed correct as long as any class from 2 to 17 is chosen, independently of whether the right class is predicted. The third is estimated by ignoring the data points whose ground truth is class 1 (the healthy state) and recomputing the accuracy of correctly classifying the failure among the remaining classes.

Table 3 FDD accuracy and inference time results. With a total of 17 classes, four discrete and abrupt failure modes were simulated for the Bebop 2 UAV per propeller, namely 20%, 40%, 60% and 80% single blade damage.

Data processing	Data fusion model	Inference time (ms)	General accuracy (%)	Detection accuracy (%)	Diagnosis accuracy (%)
IMU	LSTM (l3-c30)+BN	**88.20**	80.70	**99.98**	50.52
CAM	LSTM (l3-c30)+BN	240.01	95.93	98.53	89.94
CAM+IMU	LSTM (l3-c30)+BN	250.47	**99.55**	**99.98**	**98.86**
	Dense (l3-c128)+BN	241.77	93.56	**99.98**	83.49

Additionally, the same metrics of modified versions of the pipeline are also presented in order to demonstrate the added value of each of its components. The "Data processing" column stands for the active branches of the network, where CAM and IMU are networks with only the camera or the IMU paths active. LSTM (l3-c30)+BN is the data fusion architecture explained in subsection IV.C, whereas Dense(l3-c128)+BN is an alternative approach where the temporal relationships of the data are ignored by substituting the LSTM network with a three-layer dense NN with 128 neurons per layer.

Even though the IMU-only network feeds the sequential model with 46% more features than the camera-only network, as can be seen in Fig. 26, the latter shows an overwhelming superiority in the diagnosis of the failures with a 39.42% difference in accuracy. The reason behind that difference can be observed in Fig. 36; its confusion matrix of the predicted and true failure modes. The IMU-only network systematically confuses the front right (FR) and front left propellers (FL), as well as the back right (BR) and back left (BL). However, despite being unable to identify the failed propeller, it is able to infer the correct degree of damage. This is shown by the parallel diagonals three cells apart.

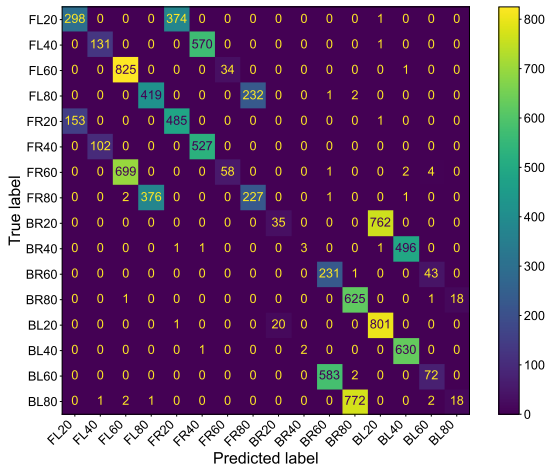

Fig. 36 IMU-only LSTM model confusion matrix of the failure modes.

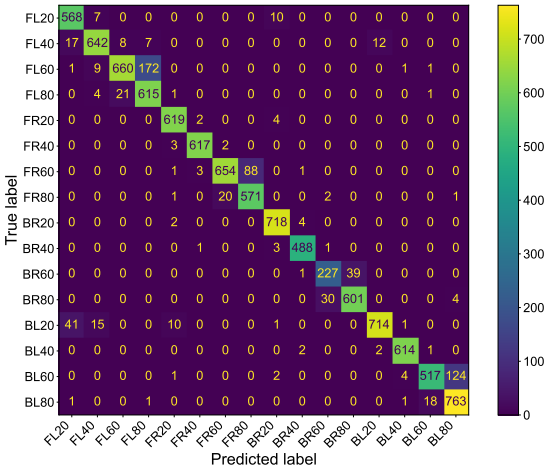

Fig. 37 Camera-only LSTM model confusion matrix of the failure modes.

In contrast, Fig. 37 shows that the camera-only network is able to correctly identify the failed actuator but fails to always accurately quantify the damage. Most of the incorrectly labelled predictions are one degree of damage higher or lower than the true label, but within the same actuator.

Both observations demonstrate the complementary nature of the camera and IMU sensors, which combined lead to the highest measured diagnosis accuracy of 98.86%. Figure 38 shows the IMU+CAM network confusion matrix with the main diagonal filled with -1's in order to visually highlight error patterns. From the multiple coloured parallel lines to the main diagonal, it can be inferred that the largest source of error originates from failing to correctly identify the damaged actuator. However, it is not constrained to the front and back propeller combinations, as it was the case for the IMU-only model.

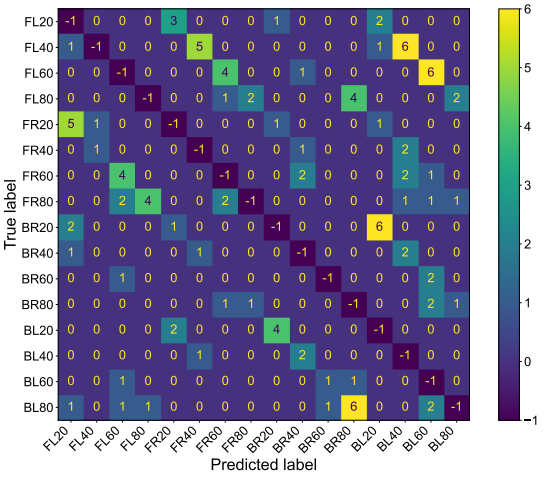

Fig. 38 IMU+CAM LSTM model confusion matrix of the failure modes with -1's in main diagonal.

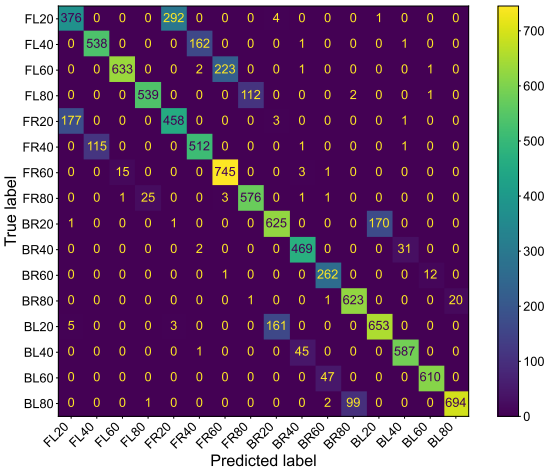

Fig. 39 IMU+CAM Dense model confusion matrix of the failure modes.

Furthermore, the difference in diagnosis accuracy between the CAM+IMU LSTM and Dense models highlights the importance of including the data temporal relationships in the FDD framework. However, it can also be seen from the detection accuracy that this information does not play a role when detecting the presence of a failure.

From the confusion matrix of the CAM+IMU Dense model shown in Fig. 39, the misinterpretation among the failures in the front and back propeller groups can again be seen. From this, it can be deduced that it is not the optical flow but its change that allows their decoupling. If the optical flow and the LSTM can each be considered a first derivative in time, then it is the second derivative of the camera's visual information that carries the differentiation factor between left and right actuators.

Finally, despite the success of the combined sensor approach, it has an inference time 2.84 times higher than the IMU-only approach: 8.03 ms (3.20%) for STFT, 72.30 ms (28.77%) for RAFT-S, 90.53 ms (36.03%) for MobileNetV3-S, and 80.41 ms (32.00%) for the LSTM+BN and classifier model. Further work has to be done in reducing the compute required by the camera path of the model by, for instance, developing tailored optical flow and feature extraction models. Additionally, an ablation study has to be performed on the hyper-parameters of the LSTM network.

VI. Conclusion

This paper proposes a novel UAV actuator FDD framework that fuses for the first time camera and IMU data online with an LSTM network. The framework pre-processes the camera information by first computing its optical flow with the RAFT-S model and then extracting features with the backbone of the MobileNetV3-S model. Both are off the shelf pre-trained efficient SOTA deep neural networks. STFT is applied on the IMU signals in order to obtain time-frequency features in the form of flattened spectrograms.

Additionally, a high-fidelity photo-realistic UAV simulator built in Unreal Engine 4 on AirSim, called UUFOSim, was presented. It is the first simulator that allows the collection of multi-sensor UAV flight data with mid-flight actuator failures injected programmatically. To the authors knowledge, UUFOSim generated the first synthetic dataset in literature for the training and testing of UAV actuator FDD approaches. Such data are of superior quality when compared to alternative simulation environments, e.g. Gazebo, allowing the development of applications with a reduced reality gap.

To demonstrate the potential of the FDD framework, UUFOSim was used to generate a dataset of 5,000 flights flown in a urban environment by a Bebop 2 platform with four options of blade damage per propeller injected during flight. The drone platform was simulated using a grey-box aerodynamic model [11] complemented with a Blade Element Theory blade damage model [34]; both obtained from literature.

The IMU-only model has shown to fail to perform damage actuator identification by systematically confusing the left and right propellers, whereas the camera-only model errors are from failure magnitude quantification. When combined, they fill the gap left by each other's weaknesses. Results show the complementary nature of the IMU and camera for FDD, achieving an accuracy of 99.98% for detection and 98.86% for diagnosis on the test dataset.

The need for a model which considers the temporal relationships in sequential data was demonstrated by substituting the LSTM layers with dense NN that do not share information about previous inputs. This modified FDD model led to a decrease in diagnosis accuracy by 15.37 percentage points without any gain in inference time.

Despite the proposed vision-based FDD framework's high accuracy, its inference time at 250 ms exceeds the IMU-only alternative model by a factor of 2.84. To address this, the authors suggest optimising the camera data processing by developing custom optical flow and feature extraction models, which currently contribute over 64% of the inference time. Optical flow ground truth images for training and testing can be retrieved from UUFOSim, and MobileNetV3-S could be further reduced in size by progressively removing the last layers and fine-tuning its weights. Alternatively, it should be investigated whether the current camera pipeline could be substituted by a sparse optical flow approach (e.g. Lucas-Kanade [51]) followed by histogram analysis for magnitude and direction of sparse optical flow vectors. Since this work has shown that the main contribution of the camera is the identification of the failed actuator, it may be the case that only the vector direction histogram would be necessary. Moreover, an ablation study on the hyper-parameters of the LSTM network could lead to a reduction in layers and/or cells. The authors also expect the rise of compute power available by the time UAVs and UAM concepts are introduced in urban environments.

Future work includes the study of a probabilistic classifier, such as a Bayesian NN, in order to provide a degree of confidence besides a prediction, as well as improving the explainability of the black-box model. The potential of other architectures that ingest sequential (image) data, such as Convolutional LSTMs and lightweight attention-based machine learning approaches, should also be considered. Another alley of investigation is the substitution of MobileNetV3-S by an image Fourier Transform as a more efficient feature extractor. Furthermore, atmospheric turbulence models should be implemented within the simulator in order to assess the robustness of the FDD approach to external disturbances;

they could induce a similar initial UAV motion as an actuator failure. Additionally, a hybrid dataset could be built which combines large quantities of synthetic UUFOSim data with a smaller real world dataset in order to reduce the reality gap. Data from multiple drones could be collected in order to make the FDD framework platform agnostic. Finally, the proposed framework should be implemented on a real Bebop 2 platform to validate the results.

To conclude, the proposed framework demonstrates the potential of including the UAV on-board camera for online failure detection and diagnosis. The authors hope that UUFOSim will help the research community to build benchmarks that will assist in the tracking of the future progress of UAV FDD, as well as other tasks that aim at making future drones more resilient to failures.

Acknowledgments

The work presented in this paper was conducted when José Ignacio de Alvear Cárdenas was a graduate student at the Faculty of Aerospace Engineering at Delft University of Technology within the Control and Simulation Division. Hence, the authors would like to thank this academic institution for providing the funding and the resources that enabled the completion of this research.

References

[1] Aurambout, J.-P., Gkoumas, K., and Ciuffo, B., "Last Mile Delivery by Drones: An Estimation of Viable Market Potential and Access to Citizens Across European Cities," *European Transport Research Review*, Vol. 11, 2019. https://doi.org/10.1186/s12544-019-0368-2.

[2] Choudhury, S., Solovey, K., Kochenderfer, M. J., and Pavone, M., "Efficient Large-Scale Multi-Drone Delivery using Transit Networks," *2020 IEEE International Conference on Robotics and Automation (ICRA)*, 2020, pp. 4543–4550. https://doi.org/10.1613/jair.1.12450.

[3] Thipphavong, D. P., Apaza, R., Barmore, B., Battiste, V., Burian, B., Dao, Q., Feary, M., Go, S., Goodrich, K. H., Homola, J., Idris, H. R., Kopardekar, P. H., Lachter, J. B., Neogi, N. A., Ng, H. K., Oseguera-Lohr, R. M., Patterson, M. D., and Verma, S. A., "Urban Air Mobility Airspace Integration Concepts and Considerations," *2018 Aviation Technology, Integration, and Operations Conference*, Atlanta, GA, 2018. https://doi.org/10.2514/6.2018-3676.

[4] Khan, H., Kushwah, K. K., Singh, S., Urkude, H., Maurya, M. R., and Sadasivuni, K. K., "Smart Technologies Driven Approaches to Tackle COVID-19 Pandemic: A Review," *3 Biotech*, Vol. 11, No. 2, 2021. https://doi.org/10.1007/s13205-020-02581-y.

[5] Lappas, V., Zoumponos, G., Kostopoulos, V., Shin, H., Tsourdos, A., Tantarini, M., Shmoko, D., Munoz, J., Amoratis, N., Maragkakis, A., Machairas, T., and Trifas, A., "EuroDRONE, a European UTM Testbed for U-Space," *2020 International Conference on Unmanned Aircraft Systems (ICUAS)*, 2020, pp. 1766–1774. https://doi.org/10.1109/icuas48674.2020.9214020.

[6] Mohammed, F., Idries, A., Mohamed, N., Al-Jaroodi, J., and Jawhar, I., "UAVs for Smart Cities: Opportunities and Challenges," *2014 International Conference on Unmanned Aircraft Systems (ICUAS)*, 2014, pp. 267–273. https://doi.org/10.1109/icuas.2014.6842265.

[7] Sun, S., Wang, X., Chu, Q., and de Visser, C., "Incremental Nonlinear Fault-Tolerant Control of a Quadrotor With Complete Loss of Two Opposing Rotors," *IEEE Transactions on Robotics*, Vol. PP, 2020, pp. 1–15. https://doi.org/10.1109/tro.2020.3010626.

[8] Mueller, M. W., Lupashin, S., D'andrea, R., and Waibel, M., "Controlled Flight of a Multicopter Experiencing a Failure Affecting an Effector," , 08 2020. URL https://patents.google.com/patent/EP3007973A1.

[9] Heng, L., Meier, L., Tanskanen, P., Fraundorfer, F., and Pollefeys, M., "Autonomous Obstacle Avoidance and Manoeuvring on a Vision-Guided MAV Using On-Board Processing," *2011 IEEE International Conference on Robotics and Automation*, Shanghai, 2011, pp. 2472–2477. https://doi.org/10.1109/icra.2011.5980095.

[10] Sun, S., Baert, M., Schijndel, B., and de Visser, C. C., "Upset Recovery Control for Quadrotors Subjected to a Complete Rotor Failure from Large Initial Disturbances," *2020 IEEE International Conference on Robotics and Automation (ICRA)*, Paris, France, 2020, pp. 4273–4279. https://doi.org/10.1109/icra40945.2020.9197239.

[11] Sun, S., and de Visser, C., "Aerodynamic Model Identification of a Quadrotor Subjected to Rotor Failures in the High-Speed Flight Regime," *IEEE Robotics and Automation Letters*, Vol. 4, No. 4, 2019, pp. 3868–3875. https://doi.org/10.1109/lra.2019.2928758.

[12] Jiang, Y., Zhiyao, Z., Haoxiang, L., and Quan, Q., "Fault Detection and Identification for Quadrotor Based on Airframe Vibration Signals: A Data-Driven Method," *2015 34th Chinese Control Conference (CCC)*, 2015, pp. 6356–6361. https://doi.org/10.1109/chicc.2015.7260639.

[13] Chen, Z., Chen, W., Liu, X., and Song, C., "Fault-Tolerant Optical Flow Sensor/SINS Integrated Navigation Scheme for MAV in a GPS-Denied Environment," *J. Sensors*, Vol. 2018, 2018, pp. 1–17. https://doi.org/10.1155/2018/9678505.

[14] Iannace, G., Ciaburro, G., and Trematerra, A., "Fault Diagnosis for UAV Blades Using Artificial Neural Network," *Robotics*, Vol. 8, 2019, p. 59. https://doi.org/10.3390/robotics8030059.

[15] García, S., López, M. E., Barea, R., Bergasa, L. M., Gómez, A., and Molinos, E. J., "Indoor SLAM for Micro Aerial Vehicles Control Using Monocular Camera and Sensor Fusion," *2016 International Conference on Autonomous Robot Systems and Competitions (ICARSC)*, 2016, pp. 205–210. https://doi.org/10.1109/icarsc.2016.46.

[16] Scaramuzza, D., and Zhang, Z., "Visual-Inertial Odometry of Aerial Robots," *Encyclopedia of Robotics*, 2020.

[17] Chen, K., "Recurrent Neural Networks for Fault Detection : An Exploratory Study on a Dataset about Air Compressor Failures of Heavy Duty Trucks," Master's thesis, Halmstad University, School of Information Technology, 2018.

[18] Zhao, H., Sun, S., and Jin, B., "Sequential Fault Diagnosis Based on LSTM Neural Network," *IEEE Access*, Vol. 6, 2018, pp. 12929–12939. https://doi.org/10.1109/access.2018.2794765.

[19] Zhu, P., Wen, L., Du, D., Bian, X., Fan, H., Hu, Q., and Ling, H., "Detection and Tracking Meet Drones Challenge," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2021, pp. 1–1. https://doi.org/10.1109/tpami.2021.3119563.

[20] Kouris, A., and Bouganis, C., "Learning to Fly by MySelf: A Self-Supervised CNN-based Approach for Autonomous Navigation," *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2018, pp. 5216–5223. https://doi.org/10.1109/iros.2018.8594204.

[21] Delmerico, J., Cieslewski, T., Rebecq, H., Faessler, M., and Scaramuzza, D., "Are We Ready for Autonomous Drone Racing? The UZH-FPV Drone Racing Dataset," *IEEE Int. Conf. Robot. Autom. (ICRA)*, 2019, pp. 6713–6719. https://doi.org/10.1109/icra.2019.8793887.

[22] Majdik, A., Till, C., and Scaramuzza, D., "The Zurich Urban Micro Aerial Vehicle Dataset," *The International Journal of Robotics Research*, Vol. 36, 2017, p. 027836491770223. https://doi.org/10.1177/0278364917702237.

[23] Bousmalis, K., Irpan, A., Wohlhart, P., Bai, Y., Kelcey, M., Kalakrishnan, M., Downs, L., Ibarz, J., Pastor, P., Konolige, K., Levine, S., and Vanhoucke, V., "Using Simulation and Domain Adaptation to Improve Efficiency of Deep Robotic Grasping," *2018 IEEE International Conference on Robotics and Automation (ICRA)*, Brisbane, 2018, pp. 4243–4250. https://doi.org/10.1109/icra.2018.8460875.

[24] Furrer, F., Burri, M., Achtelik, M., and Siegwart, R., *RotorS – A Modular Gazebo MAV Simulator Framework*, Springer International Publishing, 2016, Chap. 7, pp. 595–625. https://doi.org/10.1007/978-3-319-26054-9_23.

[25] Kohlbrecher, S., Meyer, J., Graber, T., Petersen, K., Klingauf, U., and von Stryk, O., "Hector Open Source Modules for Autonomous Mapping and Navigation with Rescue Robots," *RoboCup 2013: Robot World Cup XVII*, edited by S. Behnke, M. Veloso, A. Visser, and R. Xiong, Springer Berlin Heidelberg, Berlin, Heidelberg, 2014, pp. 624–631. https://doi.org/10.1007/978-3-662-44468-9_58.

[26] Echeverria, G., Lassabe, N., Degroote, A., and Lemaignan, S., "Modular Open Robots Simulation Engine: MORSE," *2011 IEEE International Conference on Robotics and Automation*, Shanghai, 2011, pp. 46 – 51. https://doi.org/10.1109/icra.2011.5980252.

[27] Guerra, W., Tal, E., Murali, V., Ryou, G., and Karaman, S., "FlightGoggles: Photorealistic Sensor Simulation for Perception-Driven Robotics Using Photogrammetry and Virtual Reality," *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2019, pp. 6941–6948. https://doi.org/10.1109/iros40897.2019.8968116.

[28] Song, Y., Naji, S., Kaufmann, E., Loquercio, A., and Scaramuzza, D., "Flightmare: A Flexible Quadrotor Simulator," *Conference on Robot Learning*, PMLR, 2020, pp. 1147–1157. https://doi.org/10.5167/uzh-193792.

[29] Qiu, W., Zhong, F., Zhang, Y., Qiao, S., Xiao, Z., Soo Kim, T., Wang, Y., and Yuille, A., "UnrealCV: Virtual Worlds for Computer Vision," *ACM Multimedia Open Source Software Competition*, 2017, p. 1221–1224. https://doi.org/10.1145/3123266.3129396.

[30] Müller, M., Casser, V., Lahoud, J., Smith, N., and Ghanem, B., "Sim4CV: A Photo-Realistic Simulator for Computer Vision Applications," *International Journal of Computer Vision*, Vol. 126, No. 9, 2018, p. 902–919. https://doi.org/10.1007/s11263-018-1073-7.

[31] Shah, S., Dey, D., Lovett, C., and Kapoor, A., "AirSim: High-Fidelity Visual and Physical Simulation for Autonomous Vehicles," *Field and Service Robotics*, Springer International Publishing, Zürich, Switzerland, 2018, pp. 621–635. https://doi.org/10.1007/978-3-319-67361-5_40.

[32] Madaan, R., Gyde, N., Vemprala, S., Brown, M., Nagami, K., Taubner, T., Cristofalo, E., Scaramuzza, D., Schwager, M., and Kapoor, A., "AirSim Drone Racing Lab," *Proceedings of the NeurIPS 2019 Competition and Demonstration Track*, Proceedings of Machine Learning Research, Vol. 123, PMLR, Vancouver, Canada, 2020, pp. 177–191.

[33] Sun, S., de Visser, C. C., and Chu, Q., "Quadrotor Gray-Box Model Identification from High-Speed Flight Data," *Journal of Aircraft*, Vol. 56, No. 2, 2019, pp. 645–661. https://doi.org/10.2514/1.c035135.

[34] de Alvear Cárdenas, J. I., and de Visser, C. C., *Blade Element Theory Model for UAV Blade Damage Simulation*, 2023. Manuscript submitted for publication.

[35] Badue, C., Guidolini, R., Carneiro, R. V., Azevedo, P., Cardoso, V. B., Forechi, A., Jesus, L., Berriel, R., Paixão, T. M., Mutz, F., de Paula Veronese, L., Oliveira-Santos, T., and De Souza, A. F., "Self-driving cars: A survey," *Expert Systems with Applications*, Vol. 165, 2021, p. 113816. https://doi.org/10.1016/j.eswa.2020.113816.

[36] Godoy, J., Jiménez, V., Artuñedo, A., and Villagra, J., "A Grid-Based Framework for Collective Perception in Autonomous Vehicles," *Sensors (Basel, Switzerland)*, Vol. 21, No. 3, 2021. https://doi.org/10.3390/s21030744.

[37] Carloni, R., Lippiello, V., D'Auria, M., Fumagalli, M., Mersha, A., Stramigioli, S., and Siciliano, B., "Robot Vision: Obstacle-Avoidance Techniques for Unmanned Aerial Vehicles," *Robotics & Automation Magazine, IEEE*, Vol. 20, 2013, pp. 22–31. https://doi.org/10.1109/mra.2013.2283632.

[38] Krämer, M. S., and Kuhnert, K.-D., "Multi-Sensor Fusion for UAV Collision Avoidance," *Proceedings of the 2018 2nd International Conference on Mechatronics Systems and Control Engineering*, Association for Computing Machinery, New York, NY, USA, 2018, p. 5–12. https://doi.org/10.1145/3185066.3185081.

[39] Wang, K., "B-Splines Joint Trajectory Planning," *Computers in Industry*, Vol. 10, No. 2, 1988, pp. 113–122. https://doi.org/https://doi.org/10.1016/0166-3615(88)90016-4.

[40] Schofield, P., "Computer Simulation Studies of the Liquid State," *Computer Physics Communications*, Vol. 5, No. 1, 1973, pp. 17–23. https://doi.org/10.1016/0010-4655(73)90004-0.

[41] Fortun, D., Bouthemy, P., and Kervrann, C., "Optical Flow Modeling and Computation: A Survey," *Computer Vision and Image Understanding*, Vol. 134, 2015, pp. 1–21. https://doi.org/10.1016/j.cviu.2015.02.008.

[42] Hur, J., and Roth, S., "Optical Flow Estimation in the Deep Learning Age," *Modelling Human Motion: From Human Perception to Robot Design*, Springer International Publishing, Cham, 2020, pp. 119–140. https://doi.org/10.1007/978-3-030-46732-6_7.

[43] Shah, S. T. H., and Xuezhi, X., "Traditional and Modern Strategies for Optical Flow: An Investigation," *SN Applied Sciences*, Vol. 3, No. 3, 2021, p. 289. https://doi.org/10.1007/s42452-021-04227-x.

[44] Sun, D., Yang, X., Liu, M.-Y., and Kautz, J., "PWC-Net: CNNs for Optical Flow Using Pyramid, Warping, and Cost Volume," *CVPR*, 2018, pp. 8934–8943. https://doi.org/10.1109/CVPR.2018.00931.

[45] Zhai, M., Xiang, X., Lv, N., and Kong, X., "Optical Flow and Scene Flow Estimation: A Survey," *Pattern Recognition*, Vol. 114, 2021, p. 107861. https://doi.org/10.1016/j.patcog.2021.107861.

[46] Teed, Z., and Deng, J., "RAFT: Recurrent All-Pairs Field Transforms for Optical Flow," *Computer Vision – ECCV 2020*, edited by A. Vedaldi, H. Bischof, T. Brox, and J.-M. Frahm, Springer International Publishing, Cham, 2020, pp. 402–419. https://doi.org/10.1007/978-3-030-58536-5_24.

[47] Wang, J., Zhong, Y., Dai, Y., Zhang, K., Ji, P., and Li, H., "Displacement-Invariant Matching Cost Learning for Accurate Optical Flow Estimation," *Proceedings of the 34th International Conference on Neural Information Processing Systems*, Curran Associates Inc., Red Hook, NY, USA, 2020, p. 15220–15231. https://doi.org/10.5555/3495724.3497000.

[48] Farnebäck, G., "Two-Frame Motion Estimation Based on Polynomial Expansion," *Image Analysis*, edited by J. Bigun and T. Gustavsson, Springer Berlin Heidelberg, Berlin, Heidelberg, 2003, pp. 363–370. https://doi.org/10.1007/3-540-45103-x_50.

[49] Howard, A., Pang, R., Adam, H., Le, Q., Sandler, M., Chen, B., Wang, W., Chen, L.-C., Tan, M., Chu, G., Vasudevan, V., and Zhu, Y., "Searching for MobileNetV3," *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*, 2019, pp. 1314–1324. https://doi.org/10.1109/iccv.2019.00140.

[50] Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., and Fei-Fei, L., "ImageNet: A Large-Scale Hierarchical Image Database," *2009 IEEE Conference on Computer Vision and Pattern Recognition*, 2009, pp. 248–255. https://doi.org/10.1109/cvpr.2009.5206848.

[51] Lucas, B. D., and Kanade, T., "An Iterative Image Registration Technique with an Application to Stereo Vision," *Proceedings of the 7th International Joint Conference on Artificial Intelligence - Volume 2*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1981, p. 674–679. https://doi.org/10.5555/1623264.1623280.