POLITECNICO DI TORINO

MASTER's Degree in Electronics Engineering

MASTER's Degree Thesis

Neuromorphic Vision for autonomous flight and landing on a winged drone

Supervisors

Prof. MARCELLO CHIABERGE

Prof. DARIO FLOREANO

Phd CHARBEL TOUMIEH

Eng. SIMON JEGER

Candidate

Alessandro MARCHEI

October 2024

Abstract

Fixed-wing (FW) drones, known for their superior energy efficiency and speed, are increasingly used in applications such as environmental monitoring and long-range missions. While they offer advantages over quadrotors in terms of distance and endurance, they remain less developed compared to rotary wing drones, mainly due to the difficulty of safe testing and the lack of hovering capabilities. Precision in tasks like landing and altitude estimation relies heavily on accurate motion sensing and control, typically achieved through optical flow (OF) estimation. However, at high speeds, conventional cameras struggle with issues like motion blur and limited dynamic range, which can compromise the quality of OF data.

To overcome these limitations, this work explores the use of neuromorphic cameras (NCs), which mimic biological vision by responding to brightness changes at the pixel level. These sensors provide high temporal resolution and dynamic range, making them well-suited for high-speed applications. Nevertheless, the high velocities typical of FW drones generate an overwhelming number of events (more than 10 million per second), placing significant demands on embedded platforms and complicating real-time processing.

The core of this research evaluates various OF algorithms, including both model-based and learning-based approaches, to identify the best candidate for real-time processing in event-based vision. After a comprehensive review of state-of-the-art techniques, the sparse Lucas-Kanade method was chosen for its balance of accuracy and computational efficiency. However, the method required substantial optimization to meet the specific challenges posed by real-time event-based data processing.

The thesis details the full workflow, from processing raw event streams to generating low-level actuator commands, with emphasis on key steps such as OF computation, derotation, adaptive slicing, and altitude estimation. These processes were optimized to handle the high event rates produced by NC, ensuring that the algorithm could operate efficiently on resource-constrained systems.

The final implementation was tested in outdoor environments, under a variety of conditions including full daylight, low-light scenarios, and varying event time resolutions. The system demonstrated remarkable robustness in estimating altitude and executing precise landing maneuvers, even in challenging conditions.

The outcomes of this research suggest that the algorithm is highly generalizable and could be adapted for other drone tasks, such as obstacle avoidance and visual-inertial odometry, even on hardware with limited processing power, advancing the state-of-the-art in neuromorphic vision and robotics.

Table of Contents

List of Tables

List of Figures

Acronyms

NC
Neuromorphic Camera

DVS
Dynamic Vision Sensor

SAE
Surface of Active Events

OF
Optical Flow

FW
Fixed-wing

LK
Lucas-Kanade

TECS
Total Energy Control System

RTK
Real-Time Kinematic

MRE
Medium Relative Error

ST
Slack Time

MEPS

Million of Events Per Second

EFPS

Event Frames Per Second

UAV

Unmanned Aerial Vehicle

GPU

Graphics Processing Unit

FOV

Field of View

ROS

Robot Operating System

MAV

Micro Aerial Vehicle

TFLOPS

Tera Floating Point Operations Per Second

SNN

Spiking Neural Network

Chapter 1

Neuromorphic Cameras

Neuromorphic cameras, also known as event-based cameras, are bio-inspired vision sensors that capture changes in a scene rather than the entire scene. These cameras are designed to mimic the human eye, which only sends signals to the brain when there is a change in the visual field. In contrast to traditional frame-based cameras, NCs generate an asynchronous stream of events triggered by changes in the scene.

1.1 The Dynamic Vision Sensor (DVS)

The sensor responsible for generating event-based data is known as the Dynamic Vision Sensor (DVS). It consists of a 2D array of pixels, each of which functions independently, generating an event when the brightness changes by a certain threshold.

The first prototypes were developed in the early 1990s [1],[2], utilizing analog circuitry to independently generate voltage spikes under varying lighting conditions. Since 2010, efforts have advanced the technology further, producing more sophisticated sensors with characteristics like low resolution (128x128), high dynamic range, and low latency [3]. Today, these sensors are commercially available.

Recent advancements have been made in neuromorphic technology, with several companies (such as Samsung, iniVation, Prophesee) now offering cameras with varying resolutions and specifications.

1.1.1 Event Generation

Unlike conventional cameras, which capture images at fixed time intervals, NCs generate **events** asynchronously based on changes in brightness. Each pixel in the camera independently monitors the logarithmic intensity of its photocurrent, denoted by $L = \log(I)$, where I is the brightness. An event is triggered at pixel

(x_k, y_k) and time t_k when the change in brightness, ΔL, surpasses a predefined threshold C. This process can be mathematically expressed as:

$$|\Delta L| = |L(x_k, y_k, t_k) - L(x_k, y_k, t_k - \Delta T)| > C \tag{1.1}$$

where ΔT represents the time interval since the last event was recorded at that pixel. This mechanism allows each pixel to report an event independently and asynchronously, without the need for global synchronization, resulting in a sparse stream of data focused on significant visual changes. The threshold C is relative to what is normally called *sensitivity*, which describes the minimum change in brightness required to trigger an event. A higher sensitivity results in configuration which is more prone to generate events, but on the other hand, noise can be easily triggered.

Figure 1.1 illustrates the principle of event generation by showing how the voltage changes over time with respect to the threshold level. Each time the voltage crosses the threshold, an event is generated, indicating a change in brightness.

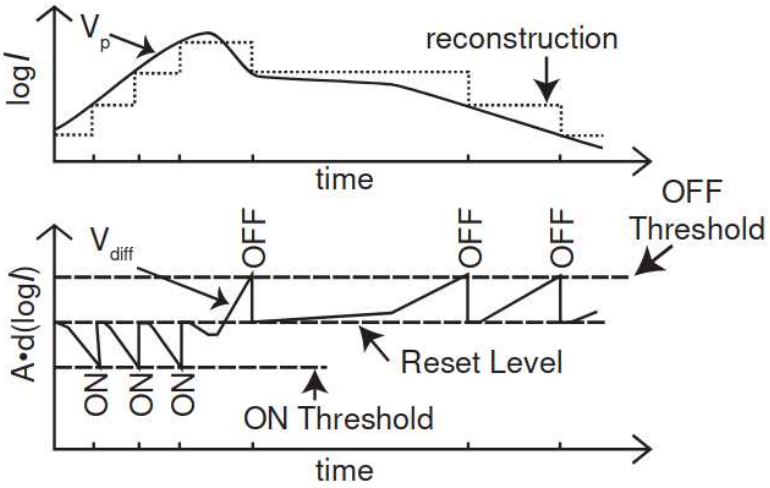

Figure 1.1: Principle of event generation. The figure on top shows the log intensity of the current over time, while the one on the bottom represents how the events are generated whenever the voltage level surpasses the thresholds. *Source:* [3]

Each triggered event is encapsulated in a tuple that encodes the information about the pixel location, time, and polarity of the brightness change. This tuple is typically represented as:

$$\boxed{e_i = (x, y, t, p)}$$

Where:

- e_i is the i-th event.
- x and y denote the pixel coordinates at which the event was triggered, with $x \in [0, W-1]$ and $y \in [0, H-1]$, where W and H are the width and height of the sensor array, respectively.
- t is the timestamp of the event, typically measured in microseconds relative to the system's epoch.
- p is the polarity of the event, where $p = +1$ indicates an increase in brightness and $p = -1$ represents a decrease.

These events, collectively known as Address-Event Representation (AER), form the output of NCs. They are generated locally by each pixel, which autonomously decides when to report a brightness change. This architecture, inspired by biological systems, uses asynchronous digital buses to transmit the events, similar to how nerve fibers relay continuous-time impulses in biological vision systems.

1.1.2 Electronic Circuitry

Figure 1.2 shows a simplified analog circuit associated with each pixel. Each pixel consists of:

- a logarithmic photoreceptor circuit,
- a differencing circuit that amplifies changes and resets voltage levels, and
- a pair of analog comparators to distinguish between positive and negative events.

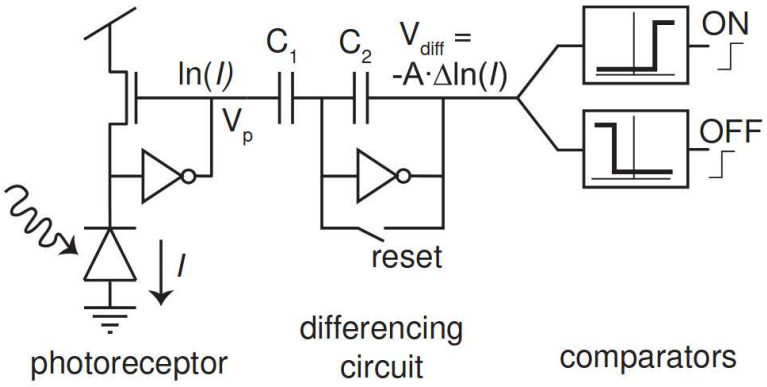

Figure 1.2: Simplified circuit for each pixel. *Source:*[3]

The photoreceptor circuit converts the incoming light intensity into a voltage signal, which is then fed into the differencing circuit. This circuit amplifies the voltage difference between the current and previous frames, generating a spike when the difference exceeds a predefined threshold. The analog comparators then determine the polarity of the event, based on whether the change in brightness is positive or negative.

This architecture is highly simplified, and the actual circuitry is more complex, incorporating additional components like gain control, noise filtering, and event generation logic. However, the basic principle remains the same: each pixel generates events independently based on changes in brightness (for more details on the pixel circuitry, refer to appendix A).

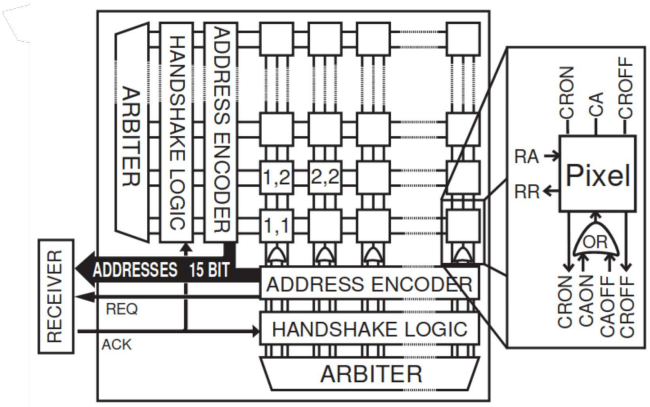

Figure 1.3: Block diagram of the architecture of the 128x128 DVS. *Source:* [3]

Each pixel in the array communicates its position and event type (ON or OFF) to the peripheral circuits asynchronously (see fig. 1.3). Each pixel has an x, y address and sends this information along with the event type. The chip outputs a 15-bit digital address, where 7 bits represent the x and y coordinates, and 1 bit indicates the event polarity (ON/OFF). This parallelism was chosen because the frame has a 128x128 resolution, and the 15-bit address is the minimum required to represent all the pixels. For a higher resolution, the address would need to be increased accordingly.

To share a common communication bus, the chip uses tristate output latches. The communication follows the Address-Event Representation (AER) protocol, ensuring that all events are transmitted without loss.

A block diagram of the architecture developed by [3] is shown in fig. 1.3.

1.2 Advantages

NCs offer significant advantages over traditional frame-based cameras. Due to their event-driven nature, they avoid redundant information when a scene is static, leading to lower bandwidth requirements. Conventional cameras capture and sample all pixels repeatedly, even when no changes occur in the scene. This results in the inefficient use of pixel bandwidth, limited dynamic range, and unnecessary data capture under controlled conditions.

In contrast, NCs emulate biological vision systems, which do not operate on frames but rather process continuous data streams in a massively parallel and data-driven manner. In biological systems, the decision to quantize visual information occurs after preprocessing steps like gain control and redundancy reduction, tasks performed by the ganglion cells. By replicating this asynchronous, event-based architecture, NCs have the potential to incorporate the strengths of biological vision into electronic devices.

Advances in integration density have made it feasible to develop vision sensors with complex pixels while maintaining acceptable pixel size and fill factors. These sensors are particularly beneficial for machine vision tasks in unsupervised environments, where the dynamic range and bandwidth limitations of traditional cameras are critical disadvantages.

1.2.1 Dynamic Range

Dynamic range is defined as the ratio of the maximum and minimum detectable light intensities. Traditional cameras have a limited dynamic range, on the range of 60-70 dB, due to the fixed voltage range of the analog-to-digital converter (ADC). In contrast, NCs have a dynamic range of at least 120 dB, which is comparable to the human eye. This is achieved by using logarithmic photoreceptors, which compress the input intensity range into a smaller voltage range, allowing for a higher dynamic range.

This allows capturing scenes with a wide range of lighting conditions, from bright sunlight to dimly lit rooms, without saturating the sensor. The high dynamic range of NCs makes them suitable for applications where the lighting conditions are unpredictable or change rapidly.

Section 1.2.1 and section 1.2.1 show a comparison between a high dynamic range scene inside a tunnel captured by a NC and a traditional camera. The NC is able to capture the entire scene without saturating the sensor, while the traditional camera fails to capture the details in the dark areas.

Figure 1.4: (Left) High dynamic range scene captured by a neuromorphic camera. (Right) High dynamic range scene captured by a traditional camera. *Source:* [4]

1.2.2 No Motion Blur

Traditional cameras capture images by integrating light over a fixed time interval, resulting in motion blur when objects move quickly. This is due to the finite exposure time of the camera, which causes moving objects to appear blurred in the image. In contrast, NCs capture events instantaneously when the brightness changes, eliminating motion blur.

This is particularly useful for applications where fast-motion plays a crucial role.

1.3 Event Representation

Due to the nature of the event stream, the data generated by NCs is fundamentally different from that of traditional frame-based cameras, so there is a need to reorganize events in ways that are more suitable for processing. Depending on the application and on the accuracy required, researchers have developed different ways to represent the event stream.

To handle the event stream, there are two main approaches to represent the data:

- **Event-by-Event Processing**: Some methods handle each event individually, as it occurs, without waiting for a group of events. This approach is widely used in spiking neural networks (SNNs). These methods update their state as new events arrive, combining them with information collected from past events to produce meaningful outputs in real time.
- **Event Grouping**: Instead of processing individual events one by one, events can be grouped together in a packet, which collects all events within a certain

time OR space range. This allows the algorithm to work with a batch of events at once. Many examples that follow this approach can be implemented:

- **Edge Images or Histograms**: events are converted into a 2D image by counting the events or summing their polarity values for each pixel over a time window (see fig. 1.6). This method creates a grid-like structure where traditional image processing algorithms can be applied. However, since the timestamps of the events are averaged or ignored, this representation can lose some of the finer details of the event data. Still, it provides a simple and effective way to transform events into a form that can be handled by conventional vision algorithms [5], [6]. An enhanced version of this method is the *motion compensated* event frame, which takes into account the motion of the scene to generate a sharper image, resulting in a more accurate representation of the scene but requiring more computational resources (see fig. 1.9).
- **Surface of active events**: A time surface keeps track of when the last event occurred at each pixel. Instead of focusing on the intensity of brightness, this method emphasizes the temporal aspect, showing how recently an event happened at each location [7], [8], [9], [10]. Pixels that have seen recent changes will have higher values. It is particularly effective for smooth motion but may struggle in scenes with a lot of noise or texture, where the surface may be too cluttered to provide useful information (see fig. 1.7).
- **Voxel Grid**: In this method, events are stored in a 3D grid, where two dimensions represent the pixel position and the third dimension represents time. This approach maintains the timing of events more accurately than event frames by assigning events to specific time intervals within the grid. The voxel grid can also track whether the brightness increased or decreased, helping to preserve important details about the nature of the events (see fig. 1.8). This makes it an effective way to process events in space and time, but it requires more memory and computational resources than other methods [11].
- **Reconstructed Brightness Images**: In some cases, events are used to reconstruct a brightness image, which represents the actual scene in a more traditional form. By combining events in a way that reduces their dependence on motion, the reconstructed image provides a clearer, more stable view of the environment (see fig. 1.10). This method is useful for tasks that require a motion-invariant representation, such as object recognition.

Each of these methods offers different strengths and is suited to specific tasks.

The choice of representation depends on the problem being solved and the type of information that needs to be extracted from the event data.

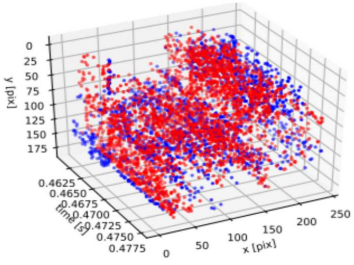

Figure 1.5: Events

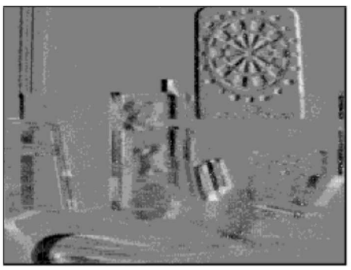

Figure 1.6: Edge Image

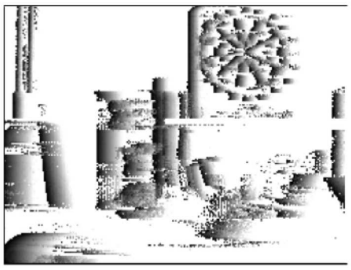

Figure 1.7: SAE

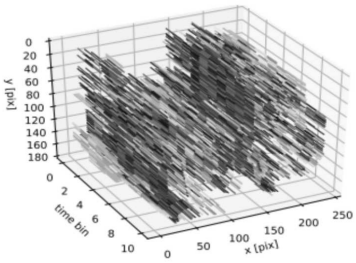

Figure 1.8: Voxel Grid

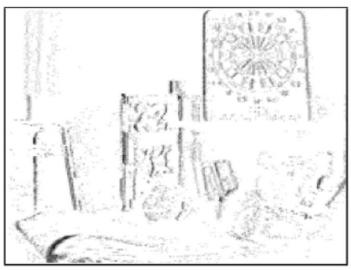

Figure 1.9: Motion Compensation

Figure 1.10: Reconstructed Image

Chapter 2

Optical Flow

Optical Flow (OF) refers to the pattern of apparent motion of objects, surfaces, and edges in a visual scene caused by the relative motion between an observer (typically a camera) and the scene. In computer vision and robotics, OF is a critical tool for estimating motion, structure, and depth in dynamic environments.

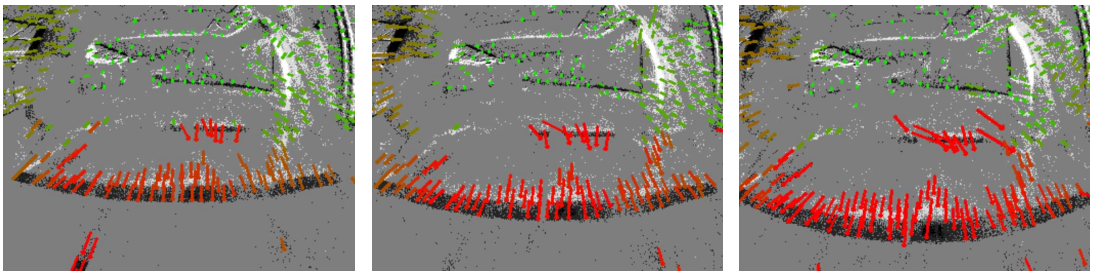

Figure 2.1: Optical Flow in a sequence of edge images.

A visual example of this vector field is shown in fig. 2.1, where the OF has been computed in a sequence of edge images. The direction of the arrow indicates the direction of motion, while the length represents the speed of motion. The scenes are captured by a camera mounted on a drone flying through a gate.

2.1 Theoretical Background

The concept of OF is grounded in the continuous tracking of pixel intensity patterns in a sequence of images. Mathematically, it is represented as a vector field that captures the motion of pixel intensities over time. Although it has a three-dimensional nature, due to the flatness of the images, we represent it as a two-dimensional vector field.

2.1.1 Optical Flow Equation

Consider a pixel at position (x, y) in an image, with intensity $I(x, y, t)$ at time t. The intensity at a specific pixel is typically represented by its grayscale value, even when the camera can reproduce RGB images. Whether due to the self-motion of the camera or the motion of the scene itself, the same area of the frame will exhibit different intensity values at a later time instant.

If the pixel moves by $(\Delta x, \Delta y)$ over a time interval Δt, the intensity at the new position $(x + \Delta x, y + \Delta y, t + \Delta t)$ will remain approximately the same due to the assumption of brightness constancy.

The brightness constancy assumption states that the brightness of an object captured by the camera remains constant over time; that is, the intensity of a pixel does not change when it moves within the image plane. This assumption can be mathematically expressed as:

$$I(x, y, t) = I(x + \Delta x, y + \Delta y, t + \Delta t) \tag{2.1}$$

eq. (2.1) indicates that under the brightness constancy assumption, the intensity of a pixel at (x, y) at time t is equal to the intensity of the same pixel at $(x + \Delta x, y + \Delta y)$ at time $t + \Delta t$.

Performing a first-order Taylor expansion of $I(x + \Delta x, y + \Delta y, t + \Delta t)$ around (x, y, t) yields:

$$I(x + \Delta x, y + \Delta y, t + \Delta t) \approx I(x, y, t) + \frac{\partial I}{\partial x}\Delta x + \frac{\partial I}{\partial y}\Delta y + \frac{\partial I}{\partial t}\Delta t$$

Using the brightness constancy assumption, we can write:

$$\frac{\partial I}{\partial x}\Delta x + \frac{\partial I}{\partial y}\Delta y + \frac{\partial I}{\partial t}\Delta t = 0$$

Dividing by Δt and defining the velocity components as $u = \frac{\Delta x}{\Delta t}$ and $v = \frac{\Delta y}{\Delta t}$, we arrive at the Optical Flow Constraint Equation:

$$\frac{\partial I}{\partial x}u + \frac{\partial I}{\partial y}v + \frac{\partial I}{\partial t} = 0 \tag{2.2}$$

Where:

- $\frac{\partial I}{\partial x}$ and $\frac{\partial I}{\partial y}$ are the spatial intensity gradients,
- $\frac{\partial I}{\partial t}$ is the temporal intensity gradient,
- u and v are the components of the OF vector at the pixel location.

2.1.2 Aperture Problem

The OF equation in eq. (2.2) alone cannot fully determine the motion vector (u, v), as it provides only one equation with two unknowns. This limitation is known as the aperture problem, which states that only the motion component in the direction of the intensity gradient can be computed [12]. To address this issue, additional assumptions or priors are introduced to regularize the solution and resolve the ambiguity. Common priors include:

- **First-Order Smoothness Prior:** Assumes that the flow field varies smoothly across the image, minimizing the spatial derivatives of the flow. This is employed in methods like Horn-Schunck [13].
- **Robust Penalty Functions:** Uses robust norms (e.g., L1 or Lorentzian) instead of L2 to better preserve motion boundaries, allowing for flow discontinuities.
- **Anisotropic Priors:** Applies smoothness regularization differently based on the image gradient direction, penalizing less along edges to better handle motion boundaries.
- **Higher-Order Priors:** Minimizes second-order derivatives of the flow to enforce smooth variations in flow curvature, capturing more complex motion patterns.
- **Spatially Weighted Priors:** Adjusts the weight of the smoothness term based on local image structure, reducing the prior's effect at edges where discontinuities are expected.

These priors help overcome the aperture problem by incorporating additional constraints, ensuring a more physically plausible OF solution.

2.1.3 Optical Flow Induced by Self-Motion

As previously discussed, OF can be induced by the self-motion of the observer (camera), even if the environment remains completely static. This phenomenon is known as ego-motion, and the OF field generated by ego-motion can provide valuable insights into the movement of the camera.

The relationship between OF and ego-motion is crucial in specific applications, as it can relate distance from the observer to speed. In fact, OF vectors are heavily dependent on scene depth at certain locations, making them suitable for estimating altitude over ground or distances from objects in space, even with monocular vision.

Formally, the Optical Flow Equation for Ego-Motion has been formulated by Koenderink and van Doorn [14]:

$$\mathbf{u}(\theta,\psi) = \frac{\mathbf{V} - (\mathbf{V}\cdot\mathbf{d}(\theta,\psi))\cdot\mathbf{d}(\theta,\psi)}{D(\theta,\psi)} - \mathbf{R}\times\mathbf{d}(\theta,\psi) \tag{2.3}$$

Where $\mathbf{u}$ is the OF vector in three-dimensional space, $\mathbf{V}$ is the translation vector, $\mathbf{R}$ is the rotation vector, $\mathbf{d}$ is the unit vector representing the viewing direction, and D is the distance to the object seen in that direction. The parameters θ and ψ are respectively the polar and azimuth angles, defining the direction of the unit vector $\mathbf{d}$ in the camera frame within a spherical coordinate system.

It is important to note that this expression is not referred to the image plane but rather to the three-dimensional space, and it is valid for each point in the scene. Since we are dealing with a flat image, the OF vectors are projected onto the image plane, and the depth information is intrinsically present and must be reprojected if the depth map is required.

As indicated in eq. (2.3), the OF vector in three dimensions results from the combination of a translational component, referred to as TransOF, and a rotational component, known as RotOF. Only the TransOF is related to distance D, while the rotational component contributes to the vector independently. This means that to compute a depth map based on D, the RotOF must be compensated and removed using an algorithm known as derotation (details in section 4.4).

Intuitively, this means that in sight directions perpendicular to the motion direction (the unit vector of TransOF), the resulting vector has the largest magnitude. Conversely, vectors generated at the focus of expansion exhibit a magnitude close to zero due to minimal or no apparent motion between the environment and the scene.

In fast motions, especially in aerial applications, self-rotations are often present, which can result in scenarios where the rotational components account for the majority of the OF vectors, leading in very inaccurate depth estimations.

2.2 Optical Flow in Nature

Optical Flow (OF) is a concept observable in nature, particularly within the visual systems of various animals. Many species, especially insects and birds, utilize OF for navigation, obstacle avoidance, and landing. Understanding how OF functions in biological systems offers valuable insights for developing robust computer vision algorithms in robotics.

2.2.1 Biological Vision and Optical Flow

Birds serve as a fascinating example of OF's application in nature, particularly in navigation and task execution. Their visual systems are primarily adapted to two

key tasks: controlling the position of their bill (or feet, in some species) during foraging and detecting predators. Notably, most bird species have a predominantly monocular field of vision, where each eye views a separate part of the environment (see fig. 2.2). This monocular configuration allows birds to achieve extensive visual coverage, enabling rapid reactions to potential threats and environmental cues.

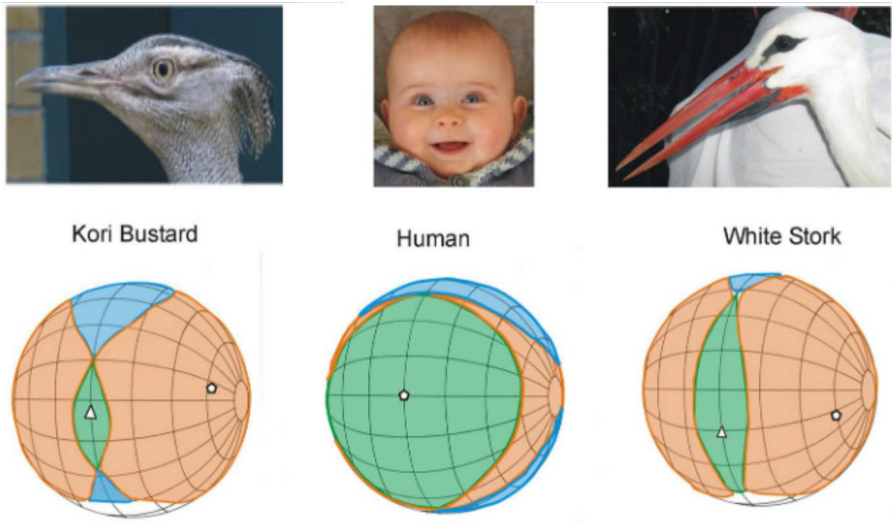

Figure 2.2: Birds' and human visual fields and binocular overlap. *Orange*, *green*, and *blue* areas represent the monocular, binocular, and blind areas, respectively [15]. *Source:* [15]

While birds possess a small binocular overlap, primarily utilized for precise tasks such as bill placement, the majority of their visual information is gathered monocularly. This raises an interesting question: *can monocular OF provide sufficient information for depth estimation?*. Evidence suggests that birds can indeed accomplish depth estimation and movement guidance through monocular vision. By relying on the rate of expansion of objects within their visual field (optic flow), birds can determine the time-to-contact with an object, which is crucial for landing and obstacle avoidance [15]. For instance, budgerigars have been observed to use optic flow to control their speed [16], while hawks rely on it to estimate their altitude and detect prey [17].

A relevant study by Serres et al.[18] demonstrates the use of optic flow for altitude estimation in seagulls. In environments with few visual cues, such as over the ocean, seagulls depend on ventral optic flow to regulate their altitude, maintaining a constant flow rate to estimate their distance from the water surface. This strategy allows them to fly at a stable altitude without requiring direct altitude measurements (see fig. 2.3).

If birds can effectively use monocular cues to navigate complex environments and

control their movements with high precision, robotic systems might similarly leverage monocular OF algorithms to achieve comparable results. This approach could prove particularly beneficial in scenarios where stereoscopic vision is impractical or unavailable, such as in lightweight aerial vehicles or drones.

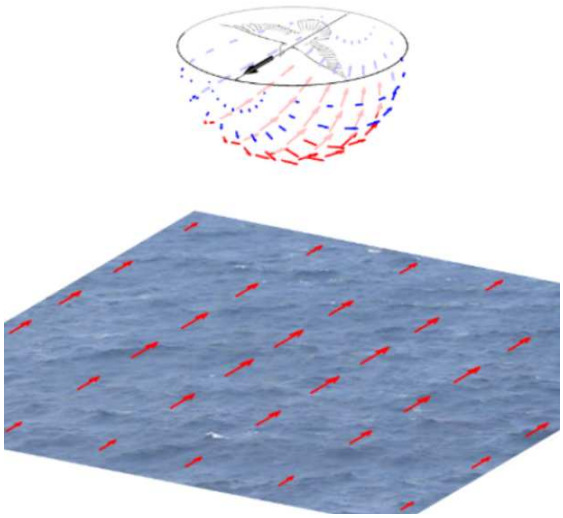

Figure 2.3: Seagulls utilize optic flow to manage their altitude over the sea. *Source:* [18].

2.3 Applications in Computer Vision

OF has a broad range of applications in computer vision, robotics, and autonomous systems due to its ability to estimate motion and structure from visual data. Some notable applications include:

- **Object Tracking:** OF is commonly employed in object tracking algorithms, enabling continuous monitoring of moving objects across video frames. By calculating the motion vectors of pixels associated with a target object, OF allows systems to follow objects even amidst complex background motion. This technique is widely utilized in surveillance systems, video editing, and augmented reality.
- **Gesture Recognition:** In human-computer interaction, OF plays a crucial role in detecting and analyzing hand and body gestures. Flow vectors corresponding to hand movements can identify various gestures, with applications in gaming, virtual reality, and assistive technologies for individuals with disabilities.

- **Motion Estimation in Video Compression:** Efficient video compression relies on motion estimation techniques, where OF aids in predicting pixel movement between consecutive frames. By representing pixel motion rather than storing full frames, compression algorithms can significantly reduce file sizes without compromising quality.

- **Autonomous Navigation:** OF is essential for facilitating autonomous navigation in drones, self-driving cars, and robots. By analyzing pixel motion in their environment, these systems can infer the relative speed and direction of objects, avoid obstacles, and plan optimal paths. A well-known application is in monocular Simultaneous Localization and Mapping (SLAM), where depth estimation and environmental mapping are performed without stereo cameras.

- **Egocentric Motion and Depth Estimation:** As discussed in section 2.1.3, OF is widely used in estimating ego-motion and depth perception, particularly in aerial vehicles and mobile robots. The ability to ascertain depth from a single camera feed makes OF invaluable in environments where stereo vision or LIDAR is impractical.

2.4 Optical Flow Algorithms for Neuromorphic Cameras

Due to the unique characteristics of Neuromorphic Cameras (NCs), which output asynchronous event streams rather than conventional frames, traditional frame-based OF algorithms are not directly applicable. This has led to the development of various specialized algorithms for event-based OF estimation, which can be broadly categorized into **model-based** and **learning-based** methods. Further subdivisions exist based on specific techniques and *sparsity* (some methods operate on a subset of detected features, while others compute flow vector-wise).

Each method is heavily influenced by the type of event representation, discussed in detail in section 1.3.

2.4.1 Model-Based Approaches

Model-based methods rely on mathematical models and geometric interpretations of event data. They typically employ an *Event-by-Event* approach, calculating flow for each incoming event from the stream.

- **Gradient-Based Methods:** These techniques attempt to adapt classical OF methods by approximating spatial and temporal gradients from sparse event data. While some success has been achieved using the brightness constancy

assumption, gradient estimation can be unreliable due to the sparsity of events [19].

- **Time-Surface Based Methods:** Time surfaces aggregate events over time and represent local activity in the x-y-t space, utilizing the *Surface of Active Events* (SAE). By fitting planes to these surfaces, OF can be estimated from the slopes, capturing the motion of edges. This method is typically employed to compute normal flow, which represents the motion component perpendicular to the edge [8, 10]. A sketch illustrating the SAE is shown in fig. 2.4.

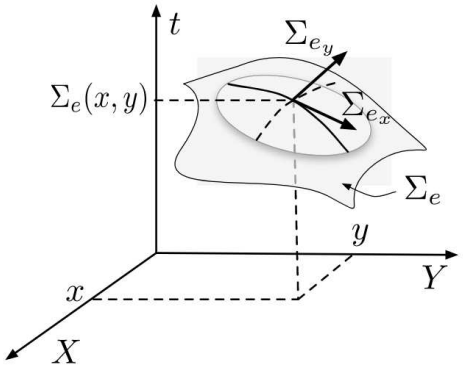

Figure 2.4: General principle of the SAE. The surface is fitted with a plane to estimate the OF. *Source:* [8].

The performance of these algorithms is evaluated in appendix B.2.

2.4.2 Learning-Based Approaches

Learning-based methods involve the use of neural networks to estimate OF directly from event streams. These methods have gained popularity due to their ability to learn complex patterns in the data without the need for manually designed models, although they require large datasets for training.

- **Artificial Neural Networks (ANNs):** Supervised and unsupervised learning techniques have been applied to train deep neural networks (typically convolutional neural networks, CNNs) on large datasets of event streams. The networks accept event frames, time surfaces, or voxel grids as input and produce dense OF estimates. Supervised methods rely on ground truth data, while unsupervised methods utilize loss functions based on photometric consistency or motion compensation to optimize flow prediction [20, 21].

- **Spiking Neural Networks (SNNs):** SNNs are particularly well-suited for processing the sparse and asynchronous nature of event data. These networks mimic the spiking behavior of biological neurons and can efficiently estimate OF by detecting motion patterns through coincidence detection [22] (as shown in fig. 2.5.c). SNNs offer low-power and bio-inspired implementations, making them ideal for neuromorphic hardware.

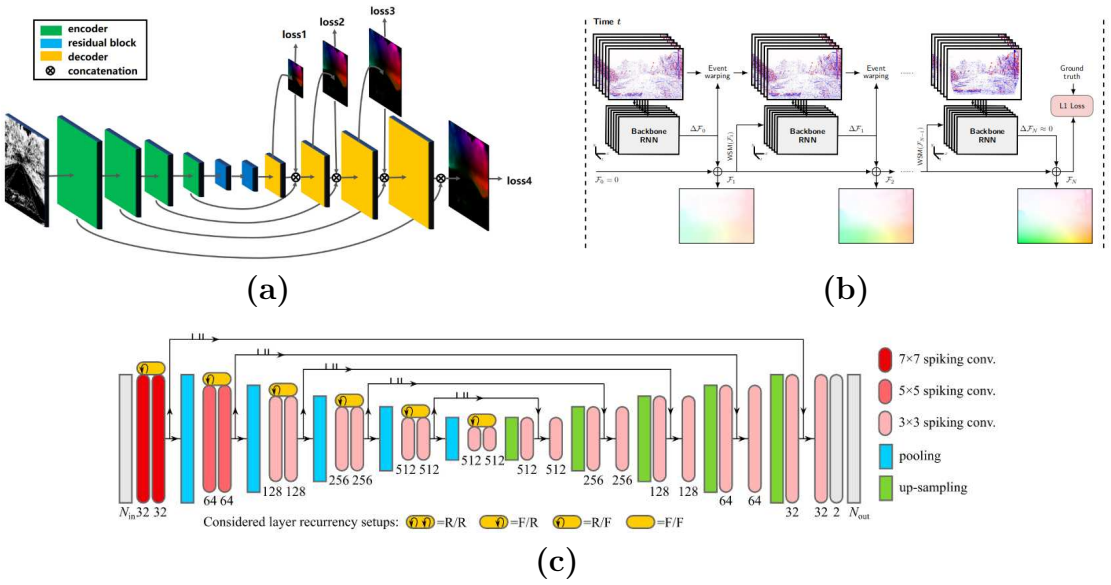

Figure 2.5: Examples of learning-based approaches for estimating optical flow. **(a)** EV-FlowNet [21], **(b)** IDNET [20], and **(c)** SNN-Timelens [22].

Overall, the diverse landscape of OF algorithms continues to evolve, drawing inspiration from both biological systems and technological advancements.

An evaluation of the performance of these algorithms is presented in appendix B.1.

2.4.3 Conventional Approaches Adapted to Neuromorphic Cameras

In addition to ad-hoc algorithms specifically designed for handling event data, several traditional Optical Flow (OF) methods developed for conventional frame-based cameras have been adapted to work with Neuromorphic Cameras (NCs). The key to these adaptations lies in representing the event stream as *edge images* (described in section 1.3), which discretize the continuous event stream into 2D histograms where the intensity of each pixel corresponds to the number of detected events in that region.

This transformation converts asynchronous data into a structured format resembling frames, enabling the application of traditional OF methods, such as:

- **Lucas-Kanade Method:** A popular algorithm for sparse OF estimation, the Lucas-Kanade method assumes small motion and utilizes a window to estimate flow by minimizing the sum of squared differences in intensities. When adapted to neuromorphic data, this method can be applied to edge images, where local gradients from the event data are used to compute flow vectors [23].
- **Horn-Schunck Method:** Another well-known algorithm for dense OF estimation, the Horn-Schunck method addresses an optimization problem that minimizes both intensity variation and the smoothness of the flow field. This method can be adapted for NCs by using edge images or voxel grids to provide the necessary input data [13].
- **Farnebäck Method:** This dense flow algorithm approximates neighborhood motion using quadratic polynomials. Similar to the Lucas-Kanade method, Farnebäck's algorithm can be adapted to edge images or time-surface representations, enabling more robust estimation on structured event data [24].

Although these methods are well-established and may not have been originally designed for event-based data, they can be effective when applied to edge images. By leveraging the spatiotemporal information in the event stream, these algorithms provide dense OF estimates compatible with traditional computer vision pipelines.

2.5 The Need for Feature Detection in Sparse Optical Flow

Sparse OF methods (e.g., Lucas-Kanade) estimate motion vectors only at specific feature points in the frame rather than computing flow for every pixel. This approach is computationally efficient and robust to noise, making it suitable for tracking distinct objects or regions of interest in a scene. However, to perform sparse OF, feature points must first be detected in the image.

Specific algorithms for event-based cameras have been developed to detect corners and edges, allowing the use of various feature detectors based on the desired latency and computational resources, including both frame-free and frame-based approaches.

The following subsections present some of the most common feature detection approaches for both frame-based and event-based cameras.

2.5.1 Frame-Based Feature Detection

Frame-based feature detection algorithms identify key points in the image that exhibit unique characteristics, such as corners, edges, or blobs. These features serve as anchor points for OF estimation, enabling the algorithm to track the motion of these points across frames.

- **Harris Corner Detector:** A widely used method for identifying corner points in an image, the Harris corner detector computes a corner response function based on local intensity gradients. It detects points with high corner scores, which are robust to noise and illumination changes, making them suitable for OF tracking [25].
- **Shi-Tomasi Corner Detector:** An extension of the Harris corner detector, the Shi-Tomasi method selects corner points based on the minimum eigenvalue of the corner response matrix. By choosing points with the highest eigenvalues, the Shi-Tomasi detector provides a more stable set of features for OF estimation [26].
- **FAST Corner Detector:** The FAST algorithm is optimized for real-time feature detection. By comparing pixel intensities around a circle of pixels, FAST can quickly identify corner points in the image. While it may be less robust than Harris or Shi-Tomasi, FAST is well-suited for applications requiring low latency [27].

These frame-based feature detectors are commonly utilized in traditional computer vision applications and can be adapted for use with NCs by processing edge images or event frames.

2.5.2 Event-Based Feature Detection

Event-based feature detection algorithms specifically identify salient points in the event stream that exhibit motion or intensity changes. A common characteristic among these algorithms is their reliance on the *Surface of Active Events* (SAE) for performing detection, often inspired by their frame-based counterparts.

- **eHarris:** A direct adaptation of the Harris corner detector, the eHarris method computes the corner response function from the SAE, detecting corners based on local event gradients [28].
- **eFAST:** Inspired by the FAST corner detector, eFAST employs the SAE to detect corner points by comparing the number of events around a circle of pixels [7] (see fig. 2.6).
- **ARC*:** An improved version of eFAST, ARC* utilizes a more efficient algorithm for detecting corners in the event stream, providing enhanced performance and accuracy [29].

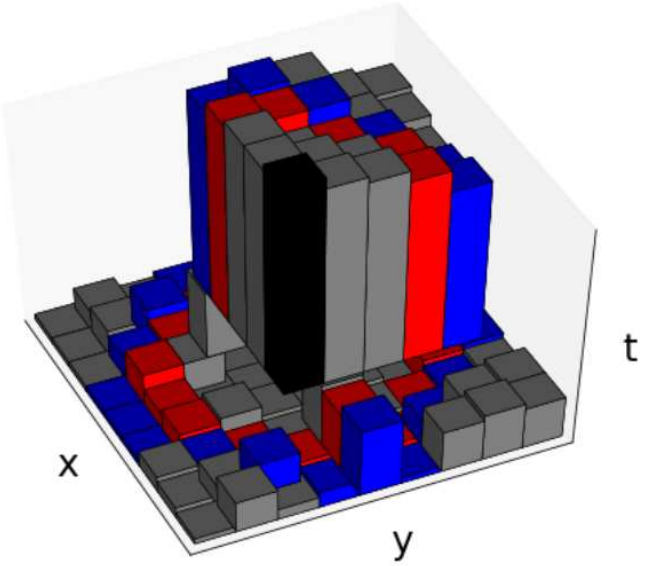

Figure 2.6: Adaptation of the FAST corner detection to the *Surface of Active Events*, used by both eFAST and ARC*. *Source:*[7].

Chapter 3

Altitude Estimation and Landing on Fixed-Wing UAVs

Vision-based control has been extensively utilized in aerial robotics, particularly in Micro Aerial Vehicles (MAVs) and fixed-wing (FW) drones, due to the wealth of environmental information that cameras can capture. Cameras provide critical data for tasks such as navigation, obstacle avoidance, and landing. However, despite their advantages, several challenges arise, including high computational demands, motion blur caused by rapid movements, and limitations in dynamic range, which complicate operations under extreme lighting conditions.

3.1 Methods for Altitude Estimation

Various methods exist for estimating altitude above ground, each with its own advantages and drawbacks:

- **GPS**: The Global Positioning System provides altitude data, but it often lacks high precision and can be unreliable in urban environments or under adverse weather conditions.
- **LIDARs**: Light Detection and Ranging offers high precision but adds significant weight and cost to the system, along with high power consumption.
- **Barometers**: Lightweight and simple, barometric altitude sensors can suffer from inaccuracies due to varying atmospheric conditions.

- **Vision-based Methods**: Cameras can estimate altitude by capturing visual cues from the ground, offering a lightweight and versatile solution without adding significant complexity to the UAV system. Additionally, using cameras allows for performing other tasks that require visual information.

3.2 Related Work

3.2.1 Landings with Fixed-Wing Drones

A variety of approaches have been proposed for FW drone landings in outdoor environments using cameras. For instance, Huh and Shim[30] introduced a system for precise landings utilizing a forward-facing camera and a hemispherical dome as the landing target, with the computations carried out entirely offboard. Similarly, Trittler et al.[31] and Wang et al.[32] relied on ground-based markers or visual patterns to guide the landing process. However, these approaches depend on specific ground infrastructure, limiting their applicability to more general scenarios.

Some efforts made by Fan et al.[33], have focused on developing and validating their algorithms in simulation environments, without real-world testing or implementation.

While substantial progress has been made in camera-based landing systems, none of these works have employed event-based cameras, which is the central focus of this thesis. Additionally, the source of OF is derived directly from optic flow mouse sensors, providing only a single vector instead of a bi-dimensional vector field [31].

3.2.2 Neuromorphic Vision

In recent years, neuromorphic vision has gained significant attention in the robotics community due to its bio-inspired nature and unique properties.

Examples of applications include:

- **Mobile Robotics**: Thanks to its properties, these cameras offer significant advantages for mobile robots (mostly wheeled or legged). The most common applications are generally related to obstacle avoidance [34], [35], and object tracking [36]. However, it should be noted that the speed of these robots was generally very low, or non-existent in the case of the last example.

- **Aerial Robotics**: These cameras have also been utilized in drones for a wider range of applications. Many examples demonstrate good obstacle avoidance capabilities, both for classical quadrotors [37], [38] (see fig. 3.1(right)), [39], [40], and bio-inspired robots [41] (see fig. 3.1(left)). Additionally, vision-based

landing has been implemented in quadrotors [42], [43], specifically for indoor applications.

- **Cars**: There has been recent interest in the automotive industry regarding these cameras. Since this technology is quite new and still in development, applications are currently limited to fast tracking and fusion with conventional cameras [44], [45], or simply to event-based datasets useful for the community [4].

Figure 3.1: Obstacle avoidance with neuromorphic cameras, on a bio-inspired drone (left) and a quadrotor (right). *Sources: [41], [37].*

In mobile and aerial robotics experiments, the motion speeds were very low (less than 3 m/s), and the cameras had low resolutions (less than 346x260). As a result, the event rates remained below 600k events per second, making the OF extraction computationally easier.

Conversely, automotive applications generate higher event rates due to their faster speeds, but they rely on powerful hardware, which can be more easily integrated into cars.

3.3 Goals

In this work, the decision to use event-based vision as a source of altitude estimation is motivated by its bio-inspired nature, especially when paired with OF algorithms.

However, computing OF in real-time remains computationally intensive, particularly in high-speed scenarios where FW drones generate a large volume of events (up to 40 million events per second (MEPS)), and where the camera has a relatively high resolution (DVXplorer Micro, 640x480). The combination of limited onboard hardware resources and the need for real-time processing adds further complexity to this challenge.

To overcome these hurdles, the main objective of this work was to design and implement a lightweight, real-time, and robust system for event-driven OF computation, optimized and tested in a landing scenario.

In addition to developing the core system, further investigations were conducted to evaluate how varying the *event time resolution* of the camera and adjusting the sensor's *sensitivity* would affect the accuracy of altitude estimation and the system's performance in low-light conditions.

- **Event Time Resolution**: This refers to the temporal precision with which an event-based camera can detect changes in the scene. Formally, it is the minimum time interval between two consecutive events. In this work, we used a DVXplorer event-based camera, which offers a configurable event time resolution ranging from 66 μs to 10 ms.
- **Sensitivity**: This is defined in section 1.1.1, described by the formula in eq. (1.1). Sensitivity is particularly important in low-light conditions, where the camera's ability to detect changes in the scene is reduced.

The DVXplorer Micro camera utilizes a *synchronous event readout scheme*, where all pixels are sampled simultaneously at fixed time intervals, similar to a *global shutter* in conventional cameras. Other cameras with lower resolutions, however, employ an *asynchronous event readout scheme*, where each pixel is read independently, and the event time resolution is the time interval between two consecutive events of the same pixel.

It is important to note that two different cases must be considered: as stated in their documentation, the camera can provide and maintain a fixed temporal resolution *without event loss* up to 1000 event frames per second (EFPS). For higher EFPS, the camera will only provide the maximum event rate, and the event time resolution will be automatically adjusted to maintain this maximum EFPS. However, in our case, for the majority of the recordings, no event loss was detected, and the camera usually maintained the desired event time resolution.

These tests were crucial in ensuring that the event-based vision system could handle varying environmental factors while maintaining reliable accuracy during landing, even under challenging lighting conditions.

The FW drone used in this work was a Bixler 3, equipped with a Pixhawk 4 flight controller and a DVXplorer Micro event-based camera, as shown in figs. 3.2 and 3.3.

The camera was mounted on the drone's nose, tilted downwards by 45°. This choice was made to avoid the conventional configuration where the camera looks straight down, which is considered trivial and limits the algorithm to landing applications only, as it does not allow the drone to see what lies ahead. By tilting the camera downwards at 45°, we achieve a better configuration for monitoring

and surveillance tasks. This setup allows the camera to capture the ground while providing some forward-looking capabilities, enabling additional functionalities such as obstacle avoidance.

The goal is to make the algorithm as general as possible while still providing cues from the ground. In fact, a further possibility is to use wider lenses, which would allow for smaller tilt angles and a broader field of view towards the front of the drone.

For more details about the physical setup, please refer to chapter 6.

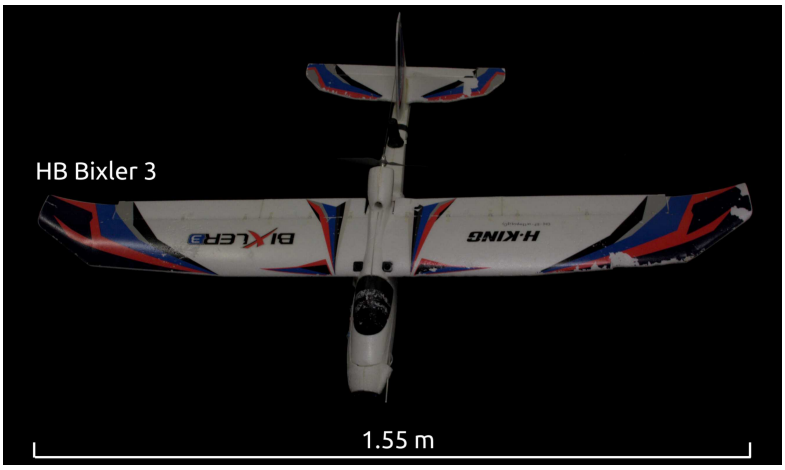

Figure 3.2: Top view of the Bixler 3 plane.

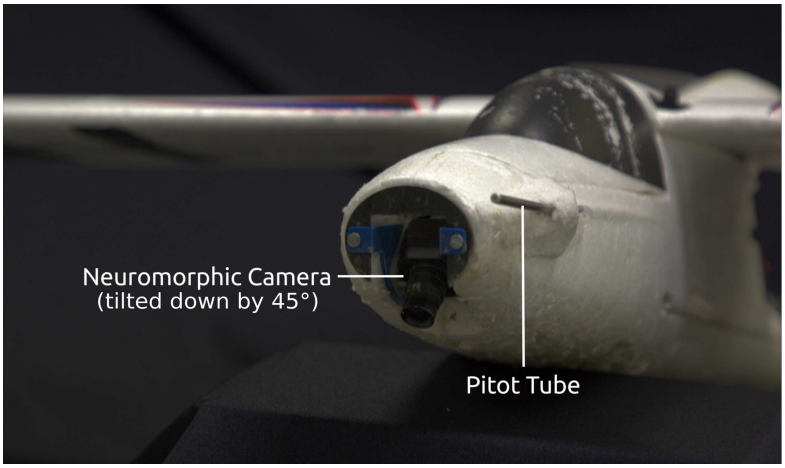

Figure 3.3: Front view of the Bixler 3 plane.

Chapter 4

The Complete Workflow : from Events to Vision-based Control

This chapter presents the complete algorithm in detail, going through each step of the pipeline, with the final goal of performed a controlled landing with a FW aircraft, the one proposed in section 5.1.

A block diagram of the complete workflow is shown in fig. 4.1, and the following sections will go through each block in detail.

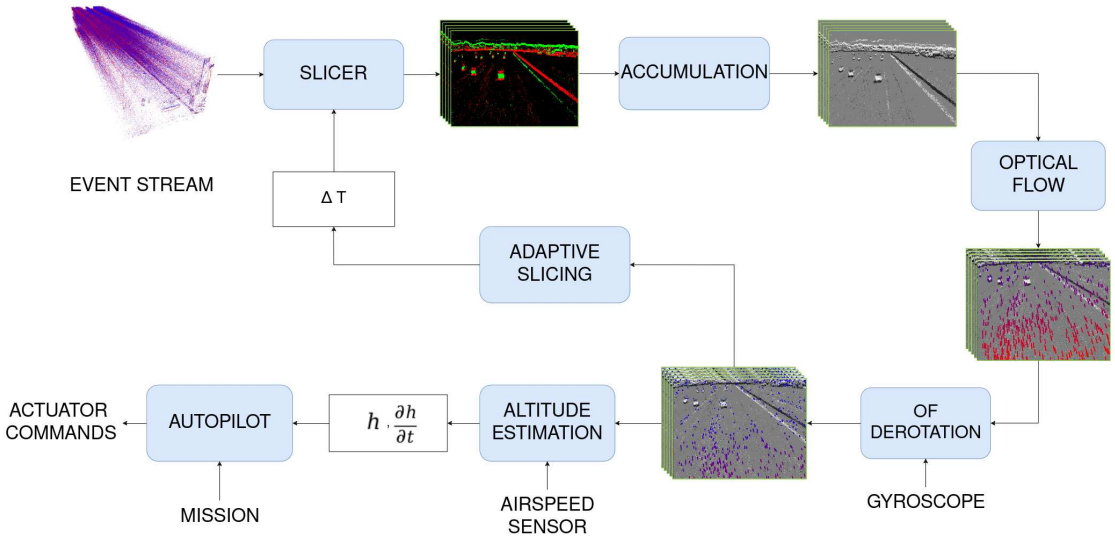

Figure 4.1: block diagram of the complete workflow

4.1 Slicer

Slicing refers to the process of dividing the *Raw Event Stream* generated by an event-based camera into finite-length vectors of events, termed *slices*. Each slice can be viewed as a batch of events.

This process is crucial for any event-based algorithm, whether it involves a neural network or a model-based approach. The event stream is asynchronous and can theoretically be infinite in length, making it impractical to process events as they are generated; the algorithm would never complete. Therefore, slicing is necessary to create manageable, finite batches of events, even if their lengths vary.

The slicing process follows specific criteria, which generally include:

- **Time-based Slicing**: The simplest method involves taking a fixed time window and collecting all events that occur within that period. This is the most common approach and the one used in this work. The time window can be roughly equivalent to the *frames per second (fps)* of a traditional camera. A simplified illustration of this slicing method is shown in fig. 4.2 (top).
- **Number of Events Slicing**: This approach slices the event stream by collecting a fixed number of events, regardless of when they were generated (see fig. 4.2 (bottom)). Although less common, it can be useful in certain scenarios.
- **Area-Number of Events Slicing**: This more complex method subdivides the entire 2D pixel array into configurable blocks (e.g., 20×20). Each block contains a counter that tracks the number of events within it. As the event stream is generated, the algorithm checks if a block's event count exceeds a predetermined threshold, triggering a slice at that point. A simplified view of this method is illustrated in fig. 4.3.

The latter method features configurable parameters that can be tuned for specific applications. For instance, one can choose the size of each block and the trigger threshold, leading to various outcomes.

As depicted in fig. 4.3, the entire frame (640 x 480 pixels) is divided into 20×20 blocks, resulting in a 32×24 grid. Each block has a counter that tracks the number of events that fall within it. In this example, the trigger is set to 1000; as indicated by the arrow, block (16,5) generated over 1000 events, prompting the slicing process.

4.1.1 Choice of Slicing Method

The choice of slicing method is highly dependent on the application's nature; however, most applications typically utilize time-based slicing. This method is

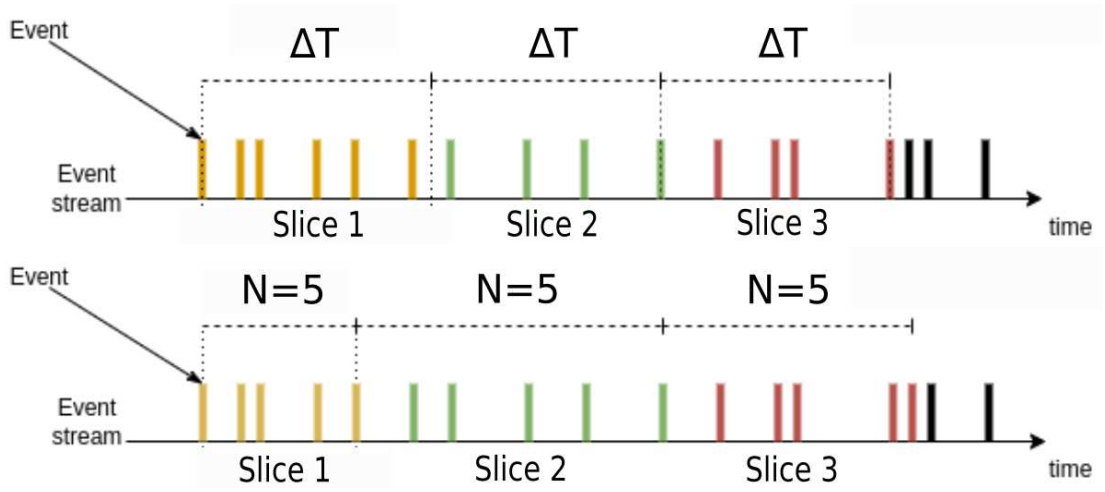

Figure 4.2: Basic principles of *slicing by time* (top) and *slicing by number of events* (bottom).

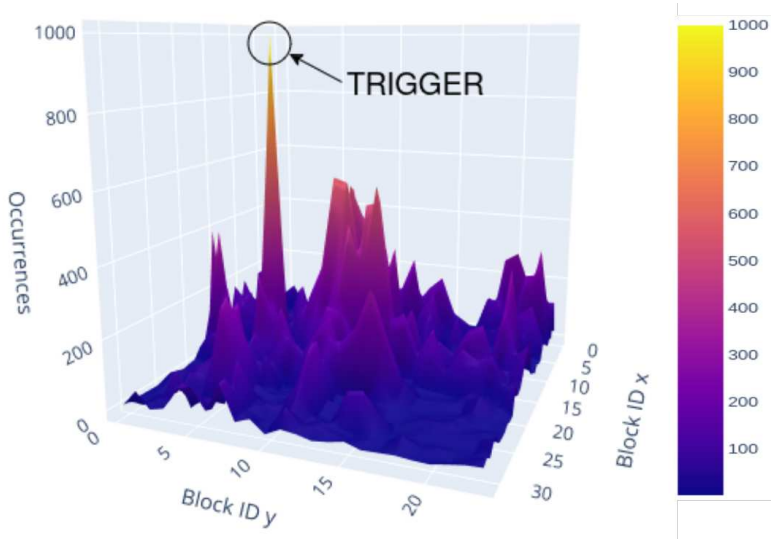

Figure 4.3: Basic principle of *area-number of events* slicing, with the trigger highlighted. The surface indicates occurrences for each block, aided by a color scale.

straightforward and intuitive, closely mirroring traditional camera frame-based approaches.

Slicing by number of events offers a simple adaptation to the density of information within a scene, allowing for slicing based on the number of events generated rather than the elapsed time.

Research has demonstrated [5] that *area-number of events slicing* can be more efficient in certain cases. This method adapts based on the density of events in the scene, acting as an *auto-focus* mechanism where rapid, dense areas trigger the slicing process, allowing for effective tracking of those regions. This approach is best suited for static cameras observing dynamic scenes. However, its implementation and tuning can be more complex, as it requires maintaining a 2D array of counters that must be updated with each event, potentially incurring high computational costs during periods of high event rates, as seen in this work.

Conversely, time-based slicing may prove inefficient in some scenarios. A very small time window may yield slices with too few events, resulting in unreliable information. On the other hand, a excessively large time window could encompass too many events, leading to higher computational costs (see fig. 4.4).

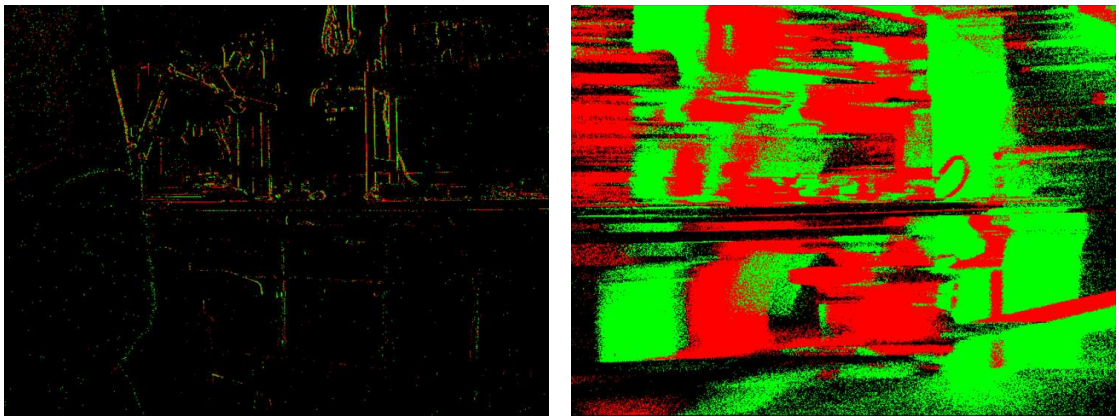

Figure 4.4: (Left) Scene captured by a small time window (1 ms). (Right) Scene captured by a large time window (33 ms). Both scenarios are not optimal, since they do not produce reliable information.

In this work, all these slicing methods were implemented and evaluated, ultimately selecting the *time-based slicing* method paired with a more sophisticated mechanism called **Adaptive Time Slicing**. This approach is likely more effective than the alternatives in terms of adaptability and computational efficiency.

Given the need for OF data, the Adaptive Time Slicing mechanism will be explained in detail in section 4.7.

For our experiments, the time window interval was chosen to be between **10 and 25 milliseconds**, as this range has demonstrated optimal effectiveness regarding efficiency and information content.

4.2 Event Representation: Edge Image

As discussed in section 1.3, it is common practice to transform the vector of events generated by the slicer into a different format, depending on the specific algorithm to be applied. This is because, in computer vision tasks, working with matrices is often more effective and efficient than dealing with vectors of potentially undefined lengths.

4.2.1 Choice of Representation

After evaluating state-of-the-art algorithms for optical flow OF estimation from events, as detailed in appendix B, we decided to utilize an adapted version of the *Sparse Lucas-Kanade* algorithm, specifically tailored for NCs.

Consequently, the choice of event representation is inherently linked to the OF algorithm selected. In this case, since the algorithm was initially developed for conventional frame-based cameras, it is essential to adapt it to event-based data. Thus, we employ the **Edge Image Representation**.

4.2.2 Edge Image

Let the accumulated event image be represented by a 2D matrix $I \in \mathbb{R}^{H \times W}$, where H is the height and W is the width of the image. For each time window T_w, grouped by the slicer, the intensity $I(x, y)$ at pixel location (x, y) is computed as follows:

$$I(x, y) = b + \sum_{e_i \in T_w} w \cdot p_i \cdot \delta(x - x_i, y - y_i)$$

where:

- $\mathbf{I} \in \mathbb{R}^{H \times W}$ represents the grayscale intensity of the *edge image*.
- b is a bias added to each pixel (e.g., $b = 128$).
- w denotes the weight (contribution) of each event.
- $p_i \in \{-1, +1\}$ is the polarity of event e_i.
- $\delta(x - x_i, y - y_i)$ is the Dirac delta function, ensuring that only events at the specific location (x_i, y_i) contribute to the intensity at pixel (x, y).
- H and W represent the height and width of the image, respectively.
- T_w is the time window of events, defined by the slicer.

In simpler terms, the intensity of each pixel is the sum of the polarities of the events that occurred at that pixel location, weighted by the coefficient w (known as *Event Contribution*), and offset by the bias b (referred to as *Neutral Potential*). Therefore, the generated matrix can also be viewed as a 2D histogram of the events occurring within the time window T_w, where each pixel essentially acts as a counter for the events at that coordinate.

If no events occur at certain locations, the resulting grayscale intensity at that pixel will equal the neutral potential. Typically, the event contribution is set to values like 70 or 100, while the neutral potential is set to either 0 or 128.

A significant aspect to consider is whether to **Ignore Event Polarity**. In this case, the value p would always assume positive values, treating negative events as positive. Since events are intrinsically linked to edges, polarity plays a minor role in the final result [46]. Robust algorithms can still perform well even without considering polarity, which reduces the complexity of the algorithm.

Figure 4.5 illustrates the difference between the two approaches. The top image shows the edge image representation obtained by *ignoring event polarity*, while the bottom image displays the representation obtained by *considering event polarity*.

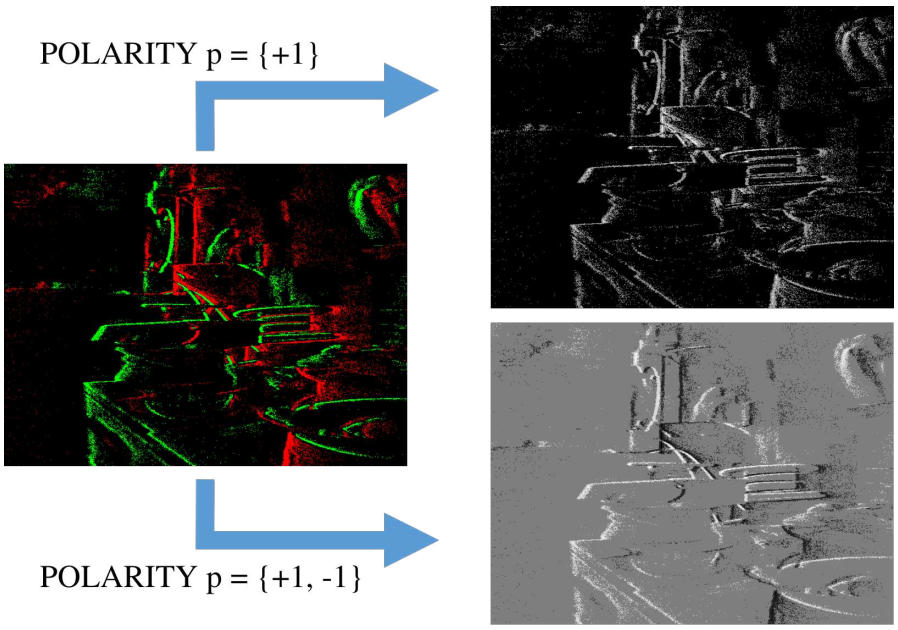

Figure 4.5: Comparison between the edge image representation with and without considering event polarity.

It is evident that while the information content remains unchanged (since the event batch is identical), the edge image considering polarity presents redundant data. In the latter case, every pixel location that did not receive any events will have

a value of 128, which is often treated as a normal value by algorithms. In contrast, the alternative representation encodes background pixels, where no events occurred, with a value of 0. As discussed in section 4.3, this *sparsity* can significantly enhance computational efficiency.

For these reasons, we opted to use the *ignoring event polarity* approach, as it is more efficient without compromising the final result of the OF algorithm. Consequently, the formula simplifies to:

$$I(x, y) = \sum_{e_i \in T_w} w \cdot \delta(x - x_i, y - y_i) \tag{4.1}$$

Equation (4.1) omits the bias term (since it is set to zero) and removes the polarity term, as it is always positive.

Another potential enhancement that some research works have explored is **Motion Compensation** applied to the edge image. This technique aims to further reduce motion blur and improve the quality of OF estimation [47, 38]. The core idea is that events are generated by the motion of objects within the scene. By compensating for the relative motion of these objects, the events can be better aligned with the edges, leading to more accurate OF estimation.

In this process, each event is warped to its position as if the object were stationary, utilizing gyroscope data and executing various operations on each event. However, despite its linear complexity of *O(n)*, the potentially high event rates from NCs, coupled with rapid motion, could impose significant computational demands. Therefore, this technique has not been implemented in the current work.

4.2.3 Comparison with Other Event Representations

The most commonly utilized event representations include the *Edge Image*, *SAE*, and *Voxel Grid*, with the choice depending on the specific application. Given the variety of Voxel Grid implementations, it has been excluded from consideration in this work due to its complexity and higher computational cost. The SAE initially appeared promising, but we ultimately decided against it after selecting our algorithm.

Both edge image transformation and SAE exhibit linear complexities. However, the latter tends to be slightly more time-consuming (typically aroun 1.3x slower) [48], as shown in fig. 4.6. Furthermore, algorithms utilizing this input type often overlook the conversion's impact on latency.

Regardless of the representation chosen, the process of transforming event batches into alternative formats suffers from high event rates. This leads to latencies ranging from 2-3 ms, with peaks reaching 15-20 ms, which can pose challenges for real-time event processing.

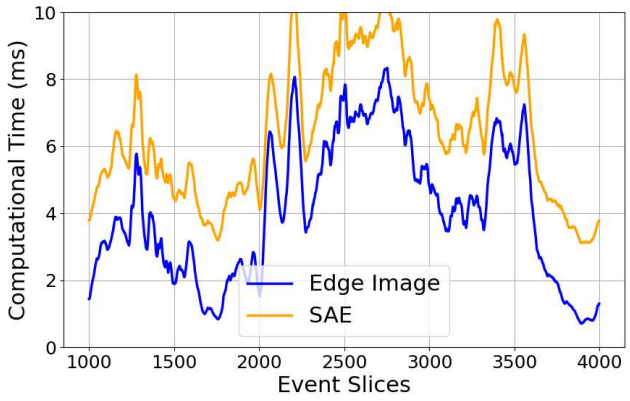

Figure 4.6: Comparison between accumulation into *edge image* and *SAE* representations in terms of computational time (in ms). The SAE is slower than accumulation.

4.3 Sparse Optical Flow

State of the art algorithms for OF computation for event-based cameras were deeply analyzed during the initial stages of this work, see appendix B, with a careful attention towards the feasibility of running in real time on embedded devices. The outcome of the evaluation favoured classical conventional methods for frame-based cameras, adapted to the asynchronous nature of event-based cameras, and the most promising candidates were the *Sparse Lucas Kanade* (LK) OF paired with either the *FAST feature detector* or the *ARC* feature detector*, and the *Dense Inverse Search* OF algorithm, which instead produces a dense flow field.

Among these three approaches, the **FAST + LK** (Sparse Lucas Kanade with FAST feature detector) algorithm was chosen as the most suitable for the task, due to its robustness against noise and and for its real-time performance (a visual example of the sparse flow is shown in fig. 4.7).

The following sections will describe in details how these two algorithms work, and then how they have been adapted and optimized to work with event-based data.

4.3.1 Sparse Lucas-Kanade Method

The *LK* method is an iterative algorithm used to estimate OF at sparse points in an image, based on the OF constraint equation. The method assumes that the OF remains constant within a small neighborhood around each pixel of interest.

The OF constraint equation, derived in the previous chapter (see eq. (2.2)), is:

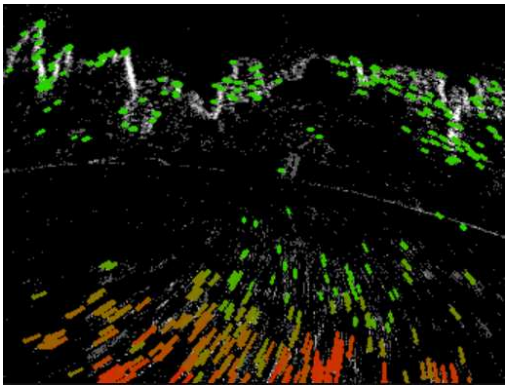

Figure 4.7: FAST + LK algorithm on a recording from a drone. The flow vectors are shown with direction and color. Green indicates small displacements, while red is used for larger ones. The frame is an *Edge Image* from the event-based camera.

$$\frac{\partial I}{\partial x}u + \frac{\partial I}{\partial y}v + \frac{\partial I}{\partial t} = 0$$

where $\frac{\partial I}{\partial x}$ and $\frac{\partial I}{\partial y}$ are the spatial gradients of the image intensity, and $\frac{\partial I}{\partial t}$ is the temporal gradient, representing intensity changes over time. u and v are the horizontal and vertical components of the OF.

This equation alone is underdetermined because there are two unknowns (u and v) but only one equation per pixel. The key assumption of the LK method is that OF is constant within a small window of pixels, typically 5x5, 7x7 or NxN in general. By considering multiple pixels within this window, we can formulate an overdetermined system that can be solved using least-squares minimization.

Mathematical Derivation

For each pixel i in a small neighborhood, we can write the OF constraint equation:

$$\frac{\partial I_i}{\partial x}u + \frac{\partial I_i}{\partial y}v = -\frac{\partial I_i}{\partial t}$$

This results in a system of equations for all pixels in the window. These can be compactly written as:

$$\mathbf{Av} = \mathbf{b}$$

where the matrix $\mathbf{A}$ contains the spatial image gradients, and $\mathbf{b}$ represents the temporal gradients:

$$\mathbf{A} = \begin{bmatrix} \sum_i \left(\frac{\partial I_i}{\partial x}\right)^2 & \sum_i \frac{\partial I_i}{\partial x}\frac{\partial I_i}{\partial y} \\ \sum_i \frac{\partial I_i}{\partial x}\frac{\partial I_i}{\partial y} & \sum_i \left(\frac{\partial I_i}{\partial y}\right)^2 \end{bmatrix} \tag{4.2}$$

$$\mathbf{b} = \begin{bmatrix} -\sum_i \frac{\partial I_i}{\partial x}\frac{\partial I_i}{\partial t} \\ -\sum_i \frac{\partial I_i}{\partial y}\frac{\partial I_i}{\partial t} \end{bmatrix} \tag{4.3}$$

The least-squares solution for the OF vector $\mathbf{v} = \begin{bmatrix} u \\ v \end{bmatrix}$ is obtained by solving:

$$\mathbf{v} = \mathbf{A^{-1}b}$$

To ensure the system can be reliably solved, the matrix $\mathbf{A}$ must be invertible, which requires sufficient variation in the image gradients. Mathematically, this means that the matrix $\mathbf{A}$ should have a non-zero determinant, and the eigenvalues of A should be well-conditioned, which is typical of edges, features and corners in the image (reason why it is usually paired with a corner detector algorithm).

Construction of the Matrix A and Vector b

Consider a small neighborhood around a pixel, hypothetically targeted by a feature detector. For each pixel i within this window, we calculate the spatial gradients $\frac{\partial I_i}{\partial x}$ and $\frac{\partial I_i}{\partial y}$, and the temporal gradient $\frac{\partial I_i}{\partial t}$. These gradients are used to form the matrix $\mathbf{A}$ and vector $\mathbf{b}$.

Spatial Gradients The spatial gradients measure the intensity changes of a pixel in both horizontal (x) and vertical (y) directions. This is key to understanding how the image brightness changes across neighboring pixels. For each pixel i in the search window, we compute:

- Gradient in the x-direction $\frac{\partial I_i}{\partial x}$:

  This gradient represents how the brightness of the pixel changes horizontally, i.e., across neighboring pixels in the x-axis. We approximate it by calculating the difference in intensity between the pixel directly to the right of i and the pixel directly to the left of i:

  $$\frac{\partial I_i}{\partial x} \approx I(x+1, y) - I(x-1, y)$$

  Here, $I(x, y)$ represents the intensity at the pixel located at coordinates (x, y), and $x+1$ and $x-1$ represent the neighboring pixels along the horizontal axis (to the right and left of the central pixel, respectively).

- Gradient in the y-direction $\frac{\partial I_i}{\partial y}$

  This gradient represents how the brightness of the pixel changes vertically, i.e., across neighboring pixels in the y-axis. We approximate it by computing the difference between the pixel directly above i and the pixel directly below i:

$$\frac{\partial I_i}{\partial y} \approx I(x, y+1) - I(x, y-1)$$

  Here, $y+1$ and $y-1$ represent the neighboring pixels along the vertical axis (above and below the central pixel, respectively).

These spatial gradients are computed for every pixel within the window, providing information about how the image intensity changes in both directions across space. Once all the gradients are computed, they are used to build the matrix $\mathbf{A}$, which contains the sum of these gradients across all pixels in the window, as shown in eq. (4.2).

Temporal Gradient The temporal gradient $\frac{\partial I_i}{\partial t}$ describes how the intensity of a pixel changes over time, specifically between two consecutive *Edge Images* in a sequence. This gradient is essential for capturing the motion between them and is computed as the difference in intensity for each pixel. Mathematically, for each pixel i:

$$\frac{\partial I_i}{\partial t} \approx I(x, y, t) - I(x, y, t-1)$$

Again, $I(x, y, t)$ represents the intensity of pixel (x, y) in the current slice at time t, and $I(x, y, t-1)$ represents the intensity of the same pixel in the previous slice. This difference gives us the rate of change in brightness at each pixel location.

The temporal gradients for all pixels in the neighborhood are used to form the vector $\mathbf{b}$, defined as in eq. (4.3).

The matrix $\mathbf{A}$ and vector $\mathbf{b}$ are computed by summing these values over the window. Then, by solving the system $\mathbf{Av} = \mathbf{b}$, we obtain the flow vector $\mathbf{v} = \begin{bmatrix} u \\ v \end{bmatrix}$, representing the movement in the x and y directions.

The least-squares minimization is applied to solve the system $\mathbf{Av} = \mathbf{b}$, but in many cases, it is preferable to give more weight to pixels that are closer to the central pixel of the window. This is achieved using a weighted least-squares solution, where the goal is to give different importance to each pixel based on its proximity to the central pixel.

In this weighted version, the least-squares equation becomes:

$$\mathbf{A}^T\mathbf{WAv} = \mathbf{A}^T\mathbf{Wb}$$

or, equivalently:

$$\mathbf{v} = (\mathbf{A}^T\mathbf{W}\mathbf{A})^{-1}\mathbf{A}^T\mathbf{W}\mathbf{b}$$

Here, $\mathbf{W}$ is an $n \times n$ diagonal matrix that contains the weights $W_{ii} = w_i$ for each pixel q_i in the neighborhood. The weight w_i is typically chosen as a Gaussian function of the distance between the pixel q_i and the central pixel p, so that pixels closer to the central pixel contribute more to the OF estimation.

Thus, the flow vector $\mathbf{v} = \begin{bmatrix} u \\ v \end{bmatrix}$ can now be computed using the weighted sums of the spatial and temporal gradients as follows:

$$\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} \sum_i w_i \left(\frac{\partial I_i}{\partial x}\right)^2 & \sum_i w_i \frac{\partial I_i}{\partial x}\frac{\partial I_i}{\partial y} \\ \sum_i w_i \frac{\partial I_i}{\partial x}\frac{\partial I_i}{\partial y} & \sum_i w_i \left(\frac{\partial I_i}{\partial y}\right)^2 \end{bmatrix}^{-1} \begin{bmatrix} -\sum_i w_i \frac{\partial I_i}{\partial x}\frac{\partial I_i}{\partial t} \\ -\sum_i w_i \frac{\partial I_i}{\partial y}\frac{\partial I_i}{\partial t} \end{bmatrix}$$

In this formulation, the weights w_i ensure that pixels nearer to the central pixel have a stronger influence on the computed OF, making the estimation more robust, especially when there are outliers or less reliable pixels in the neighborhood.

Pyramidal implementation of Lucas-Kanade

While the basic LK method works well for small displacements, it struggles with larger motions between frames. To address this, the *Pyramidal LK* method operates on multiple scales of the image using a Gaussian pyramid. This allows the algorithm to handle large displacements by first estimating the flow at lower resolutions (where displacements appear smaller) and then refining the solution at progressively higher resolutions.

The pyramid is built by successively downsampling the image to create a series of lower-resolution images. At each level of the pyramid, the LKod is applied, and the flow estimate from the coarser level is used as an initial guess for the finer level. The process continues until the original resolution is reached. The multi-scale approach allows for the following steps:

- Downsample the image to create a pyramid of lower resolutions.
- Start by computing the OF at the coarsest level (smallest image).
- Use the flow from the coarsest level to initialize the next finer level.
- Refine the flow estimates as you ascend through the pyramid, moving from coarse to fine resolution.

Mathematically, this involves solving the OF problem at each level l of the pyramid, using the solution from the previous level $l+1$ as an initial guess. At each level, the system $\mathbf{Av} = \mathbf{b}$ is solved as before, but now the flow estimates are propagated across scales, improving both accuracy and robustness to large displacements (see fig. 4.8).

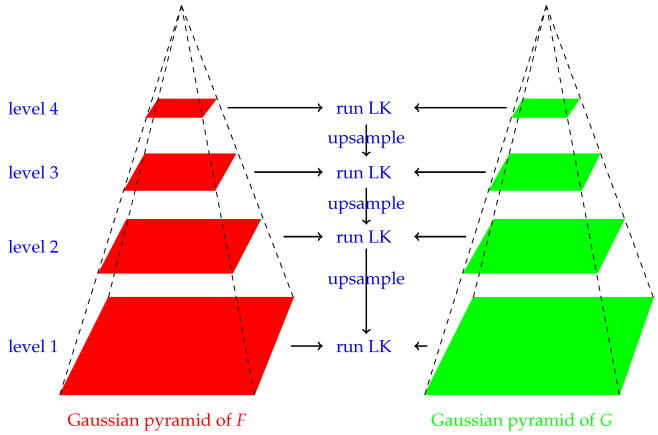

Figure 4.8: Pyramidal implementation of the LK algorithm. The OF is computed at multiple scales of the image pyramid, starting from the coarsest level and refining the estimates as the resolution increases. The flow from the previous level is used as an initial guess for the current level, allowing the algorithm to handle large displacements. In this case four levels are shown.

This pyramidal approach significantly enhances the ability of the LK to track features with large inter-frame motion, making it suitable for applications involving fast-moving objects or large scene changes, like the ones we expect in our application.

Tunable Parameters

In our approach, we have used the OpenCV implementation of the LK algorithm, called `cv::calcOpticalFlowPyrLK()`. It is a robust and efficient implementation that provides several parameters to control the behavior of the algorithm. The key parameters we can tune are:

- **Window Size**: The size of the window used for the LK. A larger window size allows the algorithm to capture larger displacements but may be less accurate in regions with small features. A smaller window size is more accurate but may struggle with large displacements. In our case 50x50 has been chosen.
- **Max Pyramidal Levels**: The maximum number of pyramid levels to use in the algorithm. More levels allow the method to handle larger displacements

but increase computational cost. In our case, 5 levels have been chosen.

- **Termination Criteria**: The criteria for stopping the iterative optimization process. This includes the maximum number of iterations and the threshold for the change in flow between iterations. In our case, 40 iterations and a threshold of 0.01 have been chosen.

Regarding accuracy, no actual evaluation has been performed compared to other state of the arts methods, because the main goal was to have a real-time algorithm that could run on embedded devices.

4.3.2 FAST feature detector

The *FAST* (Features from Accelerated Segment Test) feature detector is a corner detection algorithm that is computationally efficient and well-suited for real-time applications on embedded platforms. It was developed by Edward Rosten and Tom Drummond in 2006 [27]. It is designed to detect feature points that can later be tracked using algorithms like LK, making it an ideal pairing to reduce computational time while ensuring that the matrix $\mathbf{A}$ in LK is well-conditioned, especially when selecting good features to track.

Compared to other feature detectors such as the *Shi-Tomasi*, *Harris detector*, the FAST algorithm stands out for its superior computational speed, making it a strong candidate for real-time processing tasks. With the use of machine learning techniques, the performance of FAST can be further improved in terms of both speed and accuracy.

Segment Test Detector

The FAST detector works by evaluating a circle of 16 pixels (a Bresenham circle of radius 3) around a candidate pixel p. Each pixel in this circle is labeled from 1 to 16 in a clockwise fashion (see fig. 4.9). The detector classifies p as a corner if there exists a contiguous set of N pixels in the circle that are either all brighter than the intensity of p plus a threshold t, or all darker than the intensity of p minus the threshold t.

Mathematically, the corner conditions are defined as follows:

- Condition 1: A set of N contiguous pixels S satisfy $I(q_i) > I(p) + t$ for all $q_i \in S$ (i.e., p is a dark corner on a bright background).

- Condition 2: A set of N contiguous pixels S satisfy $I(q_i) < I(p) - t$ for all $q_i \in S$ (i.e., p is a bright corner on a dark background).

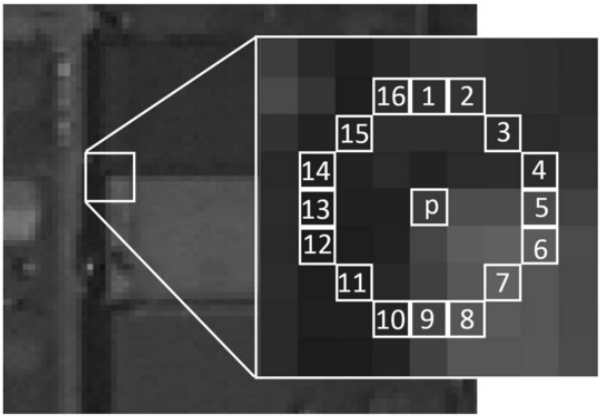

Figure 4.9: Principle of the FAST detector. The circle of 16 pixels around the candidate pixel p is evaluated to determine if p is a corner. The pixels are labeled from 1 to 16 in a clockwise fashion.

Where $I(q_i)$ is the intensity of the pixel q_i in the circle, $I(p)$ is the intensity of the candidate pixel p, and t is a threshold to determine whether the pixel is sufficiently different from the candidate pixel to be classified as a corner. Typically, N is set to 12 to balance between detecting sufficient corners and maintaining computational efficiency.

High-Speed Test for Non-Corner Rejection

The FAST algorithm includes a high-speed test to quickly reject non-corner points. Instead of checking all 16 pixels in the circle, the algorithm first examines 4 key pixels (typically at positions 1, 9, 5, and 13 in the circle). Since a valid corner requires at least 12 contiguous pixels that are all brighter or darker than $I(p)$, at least 3 of these 4 key pixels must satisfy the corner condition.

The high-speed test proceeds as follows:

- Examine the pixels at positions 1 and 9 in the circle. If both $I(1)$ and $I(9)$ fall within the range $[I(p) - t, I(p) + t]$, then p is not a corner, and no further tests are needed.
- If $I(1)$ and $I(9)$ do not satisfy the above condition, then the pixels at positions 5 and 13 are examined. If at least 3 of the 4 test pixels (1, 5, 9, 13) are either all brighter than $I(p) + t$ or all darker than $I(p) - t$, the remaining pixels in the circle are tested to confirm the corner classification.

This high-speed test significantly reduces the computational load, as it allows many non-corner pixels to be discarded after examining just a few pixels, without the need to evaluate the entire circle.

Improvements with Machine Learning

To further improve the efficiency and accuracy of the FAST detector, machine learning techniques can be applied. This enhancement involves training a decision tree on labeled corner data. Each pixel in the circular pattern around a candidate point p can be classified into one of three states:

- d: $I(q_i) \leq I(p) - t$ (darker)
- s: $I(p) - t \leq I(q_i) \leq I(p) + t$ (similar)
- b: $I(q_i) \geq I(p) + t$ (brighter)

Using these classifications, a decision tree is built using the ID3 algorithm, maximizing information gain at each step. This process ensures that the pixels most likely to indicate a corner are tested first, optimizing both accuracy and speed.

The result of this machine learning-based optimization is a highly efficient detector that requires fewer pixel comparisons on average, making it suitable for real-time applications where speed is critical. The decision tree model is then used to classify pixels in future images, providing a fast and reliable corner detection method.

Advantages of FAST with Neuromorphic Cameras and Edge Images

The *FAST* feature detector shows significant advantages when used in conjunction with NCs and edge images, particularly due to the sparse nature of these inputs. As described in section 4.2.2, accumulation of events into *Edge Images* **Ignoring Polarities**, produces frames under the form of **2D Sparse Matrixes**, where only the pixels that have received events are set to a value different from 0.

This means that the FAST detector will use the edge image as input and as a **Mask**, and will only process the pixels that have received events, skipping the others. This is a significant advantage, as it allows the algorithm to focus only on the relevant parts of the image, reducing computational load and improving efficiency.

A test comparing the performance of the FAST detector algorithm by using the normal edge image with a neutral value equal to 128 and the edge image with the neutral value set to 0, so produced by ignoring the event polarities, has been performed. The results are shown in fig. 4.10. The same algorithm has been used in both cases, with the only difference being the encoding of the events into the edge image.

Also in this case, the OpenCV implementation of the FAST detector has been used, with the function `fast->detect(edgeImage, features, edgeImage)`. The

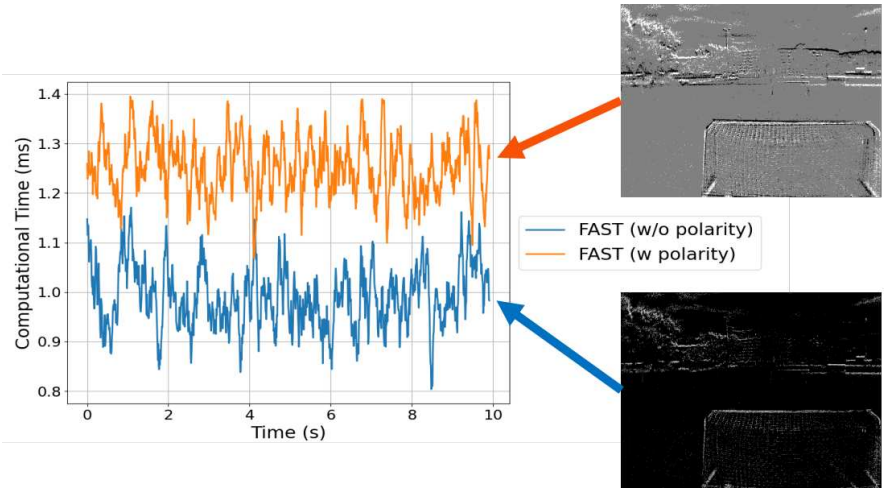

Figure 4.10: Comparison of the FAST detector performance using the accumulation considering the polarities (top) and ignoring the polarities (bottom). Although dependent on the number of events, the second case is generally faster by at least 20%. Tests were performed on a Raspberry Pi 5, on a real flight scenario.

first argument is the input image, the second is the output destination on which to save the detected features, and the third is the mask, usually a 2D matrix to skip the search on values that are different from 0. As shown, the *Edge Image* is used both as input and for the mask, so the algorithm will only process the pixels that have received events.

In terms of detected features, the number of keypoints detected is basically identical in both cases.

4.3.3 Putting all together : an improved version

Since OF computation is one of the most computationally expensive tasks in the pipeline, a significant effort has been spent to optimize the algorithm and make it as efficient as possible.

Below, the OF pipeline is shown (see fig. 4.11), and the following paragraphs will motivate and describe each choice.

Gradient Scoring

Since the FAST detector has been developed for frames captured by conventional cameras, it could suffer in detecting a constant number of features over time, as the edge images are somehow artificial and do not contain the same amount of information as a normal frame.

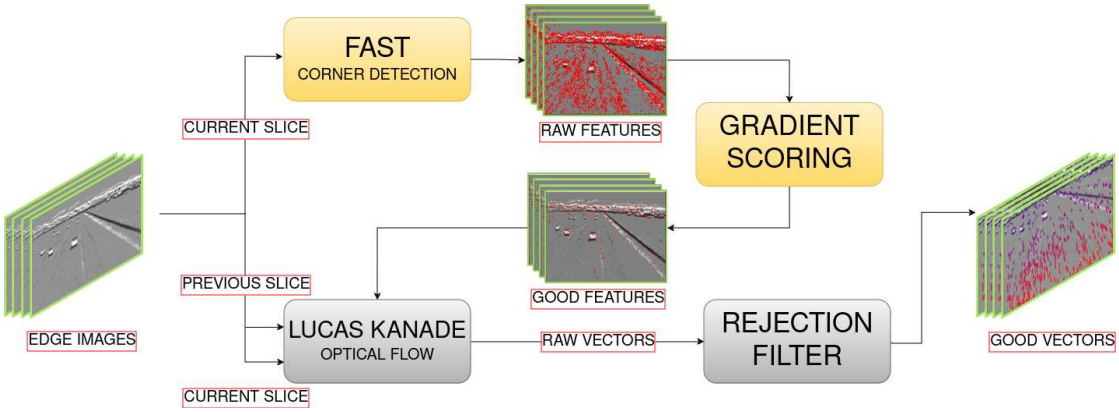

Figure 4.11: OF pipeline, including a *parallel version* of the FAST feature detector and the LK algorithm.

The OpenCV implementation of the detector does not allow to choose a specific number of features to detect (as in the case of the Shi-Tomasi detector), but only a **threshold** to consider a pixel as a corner. This could become a problem, because it is highly desirable to rely on a constant number of OF vectors, in order to have a stable and reliable estimation of the plane's motion.

Moreover, the *threshold* itself is a parameter that is difficult to tune in case of edge images, as the intensity of the pixels really presents steep changes, not gradual as in the case of normal frames. The result is that there is no clear threshold that can be set to detect the corners, and the number of features detected can vary significantly from slice to slice. An example is shown in section 4.3.3.

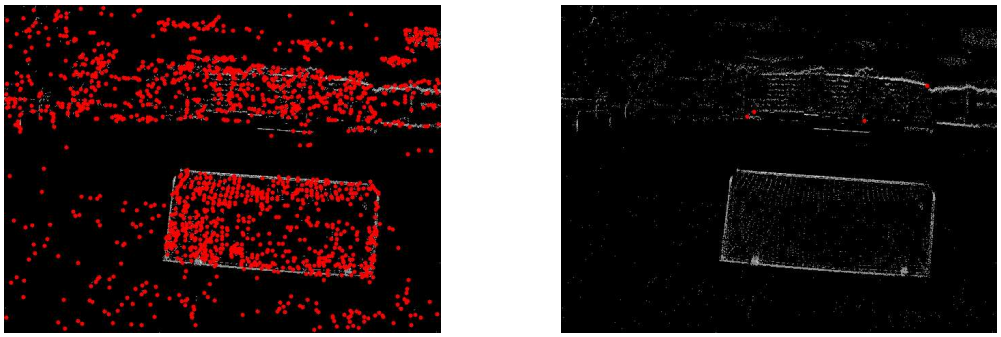

Figure 4.12: Comparison between detected thresholds by FAST in case of different thresholds. (Left) shows the result of the detector with a threshold of 142, while (Right) shows the result with a threshold of 143. Only a small change in the threshold produces a significant difference in the number of detected features, due to the nature of the images.

The detector only produce either too few (4) or too many (1682) features, and this is not acceptable for the OF algorithm, as it would produce either a very noisy or a very sparse flow field.

The solution could be to use a more robust feature detector (e.g. *Shi-Tomasi*), but that would highly affect computational time, and the goal is to have a real-time algorithm.

The approach we used instead is called **Gradient Scoring**, and it based on the idea of using a 1^{st} order *Sobel Operator* to compute the gradient of the image, and then use the gradient itself as a scoring function for the FAST detector. The idea is that the gradient of the image is a measure of the intensity changes, and hence of the edges, and the FAST detector should detect the corners where the edges are more pronounced.

Sobel Operator

The Sobel operator is used to approximate the gradient of an image by detecting intensity changes. It applies two 3x3 convolution kernels: G_x for horizontal changes and G_y for vertical changes:

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}, \quad G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

For an image I, the gradients $\frac{\partial I}{\partial x}$ and $\frac{\partial I}{\partial y}$ are computed as:

$$\frac{\partial I}{\partial x} = G_x * I, \quad \frac{\partial I}{\partial y} = G_y * I$$

The gradient magnitude, which highlights edges, is given by:

$$\text{Magnitude} = \sqrt{\left(\frac{\partial I}{\partial x}\right)^2 + \left(\frac{\partial I}{\partial y}\right)^2}$$

Each corner is then associated with a score equal to the magnitude of the gradient at that point. The higher the gradient magnitude, the higher the *quality* of the pixel to be identified as a corner. This approach ensures that the FAST detector focuses on the most significant edges in the image, improving the robustness of the feature detection process.

Other scoring methods exists, like the `cv::cornerMinEigenVal` function, which is employed by the Shi-Tomasi detector. However, this method requires more computation time and is therefore not preferred for real-time applications. In fact, computing eigenvalues is a more complex operation than computing the gradient,

and it is not necessary to have a more accurate scoring function, as the FAST detector is already very efficient.

In our method, we have used a **Threshold = 90**, in a way that many features were detected, then the gradient scoring was applied to *rank* the features, and only the **Top 150** were chosen. This way, we have a constant number of features, and the most significant ones are selected, while paying for only a small overhead that is usually around 2-3 ms.

The quality of the features are really similar to the ones proposed by the *Shi-Tomasi* detector (see section 4.3.3), but the computational time is significantly lower, by a factor of 4.5x. Each iteration of the GoodFeaturesToTrack algorithm takes around 20 ms, while the FAST detector with the gradient scoring takes around 4 ms.

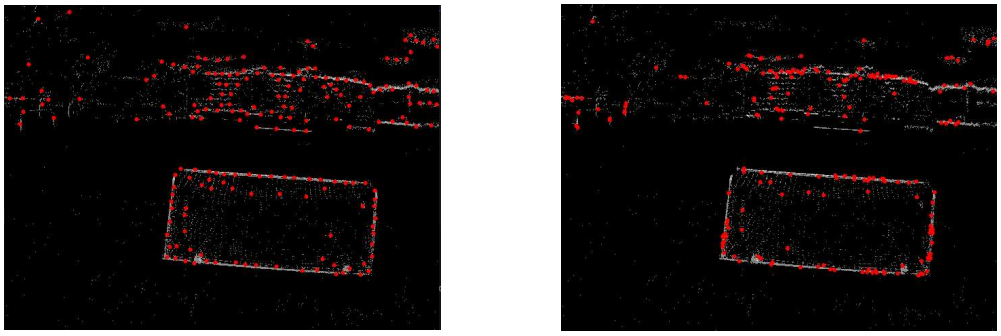

Figure 4.13: Comparison between the features detected by the Shi-Tomasi detector (Left) and the FAST detector with gradient scoring (Right). The number of features is set to 150 in both cases.

Parallel implementation

The LK algorithm is dependent on the features detected by the FAST detector, and the two algorithms are typically run sequentially. In such cases, the OF algorithm must wait for the new features to be detected before it can proceed. To overcome this limitation, it is possible to parallelize the execution of the two algorithms, improving the overall performance.

The key idea is to allow the FAST detector to work on the current frame at time t_i, while the LK algorithm processes the features and frames from the previous time step t_{i-1}. In this manner, the features detected by FAST in frame i will be used in the subsequent iteration of LK, at time t_{i+1}. Thus, at any given time t_i, the LK is operating on the slices $i-1$ and i for the OF computation, while the FAST detector is concurrently detecting features on the slice i.

This approach allows for the simultaneous processing of feature detection and

OF computation, avoiding the bottleneck caused by sequential execution. For instance, at time t_i, the LK uses the features detected by FAST at time t_{i-1} and processes slices $i-1$ and i to compute the OF. Meanwhile, the FAST detector is detecting features on slice i, which will be used by the LK in the next iteration at time t_{i+1}.

This parallelization is made possible by modern multi-core architectures such as the Raspberry Pi 5, which is equipped with a Quad-Core Cortex-A76 processor, allowing up to four threads to run in parallel. By allocating separate threads to the FAST detector and the LK OF algorithm, both processes can execute concurrently. While running the two algorithms in parallel may introduce some overhead due to context switching and resource sharing, the overall computational time for processing each frame is reduced compared to running the processes sequentially (see fig. 4.14).

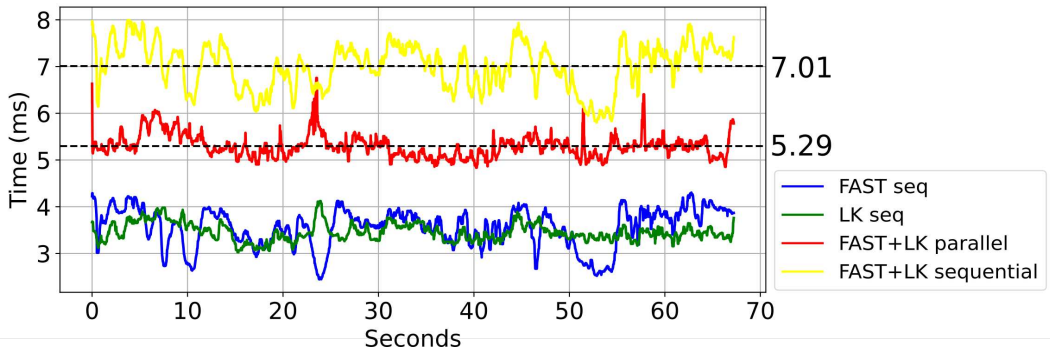

Figure 4.14: Test comparing the execution time of the FAST detector and the LK when run sequentially and in parallel. The performance improvement is around 25%. The data was collected on a Raspberry Pi 5, during a real flight scenario.

4.3.4 Filtering of the OF vectors

For every iteration, arond 150 features are detected by our algorithm, and the LK computes the OF vectors for each of them. However, not all of them are reliable, and some may be outliers or noisy (as a result of the OF algorithm). To ensure the accuracy of the estimation, it is essential to filter out the unreliable vectors and retain only the most robust ones.

There are many types of filters available, but in our case a 2 stage filter is applied:

1. **Magnitude Threshold**: The first filter is based on the magnitude of the OF vectors. The idea is to discard the ones that have a magnitude above a

certain threshold, as they are likely to be unreliable. This value is set to 60 pixels (as displacement), which is a reasonable value for the expected motion of the drone.

2. **Interquartile Range (IQR) Filter**: The second filter is based on the interquartile range (IQR) of the OF vectors magnitudes. The IQR is a measure of statistical dispersion that is robust to outliers. The filter works by computing the IQR of the magnitudes of the OF vectors and removing the vectors that fall outside a certain range. This range is defined as $Q1 - 1.5 \times IQR$ to $Q3 + 1.5 \times IQR$, where $Q1$ and $Q3$ are the first and third quartiles, respectively. This filter helps to remove outliers and retain only the most reliable OF vectors.

4.4 Vector Derotation

OF vectors describe the relative motion between the observer and the environment (see section 2.1). Since we assume that the environment is static, the OF field purely describes the camera motion, given by a combination of translation and rotation.

However, since the goal of the project is to estimate depths from the OF field, we need to remove the rotational components of the vectors, since they are not related to the depth (hence the name **Derotation**). As a recall, the *Optical Flow Equation for ego-motion* in eq. (2.3) is the following:

$$\mathbf{P}(\theta, \psi) = \frac{\mathbf{V} - (\mathbf{V} \cdot \mathbf{d}(\theta, \psi)) \cdot \mathbf{d}(\theta, \psi)}{D(\theta, \psi)} - \mathbf{R} \times \mathbf{d}(\theta, \psi) \tag{4.4}$$

It can also be expressed as the sum of two components, the translational and the rotational components:

$$\mathbf{P}(\theta, \psi) = \mathbf{P}_{\text{trans}}(\theta, \psi) + \mathbf{P}_{\text{rot}}(\theta, \psi)$$

The translational component $\mathbf{P}_{\text{trans}}(\theta, \psi)$ is given by:

$$\mathbf{P}_{\text{trans}}(\theta, \psi) = \frac{\mathbf{V} - (\mathbf{V} \cdot \mathbf{d}(\theta, \psi)) \cdot \mathbf{d}(\theta, \psi)}{D(\theta, \psi)}$$

where $\mathbf{V}$ is the translational velocity, $\mathbf{d}(\theta, \psi)$ is the unit direction vector, and $D(\theta, \psi)$ is the depth.

The rotational component $\mathbf{P}_{\text{rot}}(\theta, \psi)$ is expressed as:

$$\mathbf{P}_{\text{rot}}(\theta, \psi) = -\mathbf{R} \times \mathbf{d}(\theta, \psi)$$

where $\mathbf{R}$ is the rotational velocity vector, and $\mathbf{d}(\theta, \psi)$ is the same unit direction vector. Thus, the total OF $\mathbf{P}(\theta, \psi)$ can be seen as the sum of the translational and rotational effects.

It has to be noted that $\mathbf{P}$ represents the real OF vector, which is a 3D vector, while the OF vectors obtained from the camera are 2D vectors, projected on the image plane, often denoted as $\mathbf{u}$.

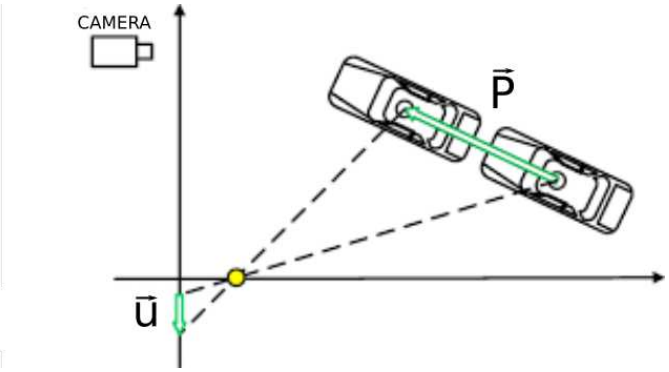

Figure 4.15: The $\mathbf{P}$ vector in 3D space and its projection $\mathbf{u}$ in a 2D frame.

The main source of data able to compensate for the rotational motion is the IMU, which provides the angular velocity of the camera. This way, the $\mathbf{R}$ vector is directly available from the gyroscope.

In literature, several methods have been developed to perform the derotation of the OF vectors. [49] and [50] performed this process by simply subtracting the gyro samples to the OF vector since their flow was pointwise due to the OF mouse sensors they have used, which were positioned on the horizontal plane of the drones, resulting in a simple 2D rotation.

In case of 2D cameras, the derotation process becomes more complex beucase 3D components are involved. [51] proposed a method using each IMU sample (which is at a higher frequency than the camera samples) to reconstruct the original position of a feature as if no rotation occurred during the camera sampling window. This method has been tested in this work, and results are explained and discussed in section 4.4.3. [52] implemented a Kalman Filter to estimate the rotation of the camera and then remove it from the OF vectors. It is a more complex method, but the accuracy of the estimated $\mathbf{u}_{\text{trans}}$ is hardly comparable to the other methods, as it is not directly available.

In this work, three different *derotation systems* have been implemented and tested, and in the following sections methods and results are explained and discussed.

4.4.1 Notations

Before delving into the process of vector derotation, it is essential to formally define the two key vectors that will be used throughout the following sections.

For a given pixel position $\mathbf{p} = (x, y)$ on the image plane, we define the following:

1. **Vector a**: The 3D vector **a** represents the coordinate of a feature in space with respect to the camera's optical axis. It is mathematically expressed as:

$$\mathbf{a} = \begin{bmatrix} x - c_x \\ y - c_y \\ f_x \end{bmatrix} \tag{4.5}$$

where c_x and c_y are the coordinates of the principal point (optical center) on the image plane, and f_x is the focal length of the camera in pixel units.

2. **Vector d**: The unit direction vector **d** represents the normalized direction from the camera center towards a point on the environment, expressed in our case in spatial euclidian coordinates, while normal spherical ones could be used. It is mathematically defined as:

$$\mathbf{d} = \frac{\mathbf{a}}{\|\mathbf{a}\|}$$

where **a** is the feature vector defined earlier, and $\|\mathbf{a}\|$ is its magnitude.

The **Vector R** is composed by the gyroscope data, as $\mathbf{R} = (g_x, g_y, g_z)$.

4.4.2 Planar Derotation

This method is the simplest one, and it based on the fact that the 3$^{\text{rd}}$ component of the OF vector is not considered, so only the x and y components are taken into account and subtracted from the computed **u**.

The rotational OF, denoted by $\mathbf{P}_{\text{rot,pix/s}}$, and expressed directly in pixel coordinates, is induced by this rotational velocity and can be computed using the cross product between **R** and **a**:

$$\mathbf{P}_{\text{rot,pix/s}} = -\mathbf{R} \times \mathbf{a}$$

This equation represents the OF caused purely by the rotational motion of the camera, in units of pixels per second.

In two dimensions only, the components of the derotated OF can be expressed as:

$$\mathbf{u}_{\text{trans}} = \begin{bmatrix} u_{\text{pixel/sec}} - \mathbf{P}_{\text{rot,x}} \\ v_{\text{pixel/sec}} - \mathbf{P}_{\text{rot,y}} \end{bmatrix} \tag{4.6}$$

Equation (4.6) highlights the simplicity of the derotation method, in fact the $\mathbf{u}_{\text{rot,z}}$ is not considered, to make the method extremely simple. Since the 3$^{\text{rd}}$ component is neglected, the method is called *Planar Derotation*, which offers a simple and fast way to compensate for the rotation, but it is not as accurate as the other methods, as it does not consider the full 3D rotation of the camera.

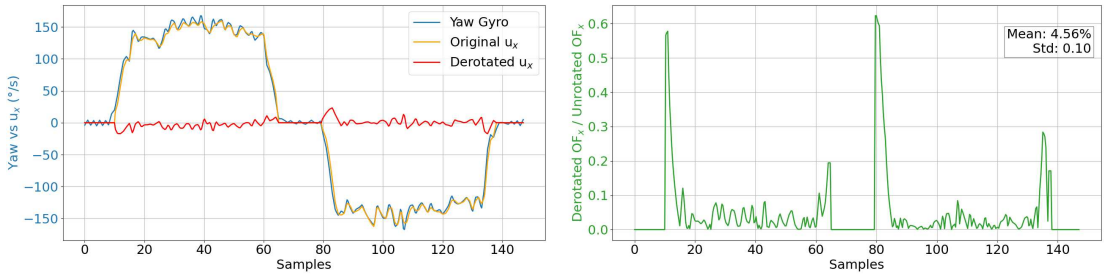

Figure 4.16: Results of the Planar Derotation method, applied onto a sequence of slices. The camera has been mounted on a servo motor performing rotations @ 150 deg/s. (Left) Comparison of the yaw gyro data (blue), the horizontal OF $\mathbf{u}_x$ (orange) and the derotated OF $\mathbf{u}_{der,x}$, (red). (Right) Ratio between the derotated and the raw OF.

4.4.3 Derotation using Quaternions

This method is an enhancement of the approach proposed in [51], offering improved computational efficiency by utilizing quaternions instead of Euler angles and rotation matrices.

The core idea is to compensate for the rotational effects in the OF by reprojecting each feature point as if the rotation did not occur. This is achieved by transforming the direction vector of each feature using quaternions derived from the gyroscope data.

For each gyroscope measurement within the slicing window, the instantaneous rotation of the camera is represented as a quaternion $\mathbf{q}_{\text{imu}}$. This quaternion encapsulates the rotational change between consecutive IMU readings without the need for averaging, providing high-fidelity rotational information.

When dealing with n contiguous rotations during the slicing window, each represented by quaternions $\mathbf{q}_1, \mathbf{q}_2, \ldots, \mathbf{q}_n$, the total rotation can be obtained by the ordered product of each quaternion relative to the IMU angular rate:

$$\mathbf{q}_{\text{tot}} = \mathbf{q}_n \otimes \mathbf{q}_{n-1} \otimes \cdots \otimes \mathbf{q}_1$$

The associativity of quaternion multiplication ensures that the total rotation is correctly represented, regardless of how the quaternions are grouped in the

multiplication.

To compensate for the rotation, we transform the vector $\mathbf{a}$ at time t_2 back to its orientation at time t_1 by applying the inverse of the total rotation quaternion (see fig. 4.17). The corrected vector $\mathbf{a}'$ is computed using quaternion multiplication:

$$\mathbf{a}' = \mathbf{q}_{\text{tot}}^{-1} \otimes \mathbf{a} \otimes \mathbf{q}_{\text{tot}}$$

Here, $\mathbf{q}_{\text{tot}}^{-1}$ is the inverse of the total rotation quaternion, and $\mathbf{a}$ is treated as a pure quaternion (with zero scalar part), represented as $\mathbf{a} = [0, a_x, a_y, a_z]$.

The corrected vector $\mathbf{a}'$ points at the feature in the frame as if no rotation occurred, allowing us to compute the derotated OF vector $\mathbf{u}_{\text{trans}}$.

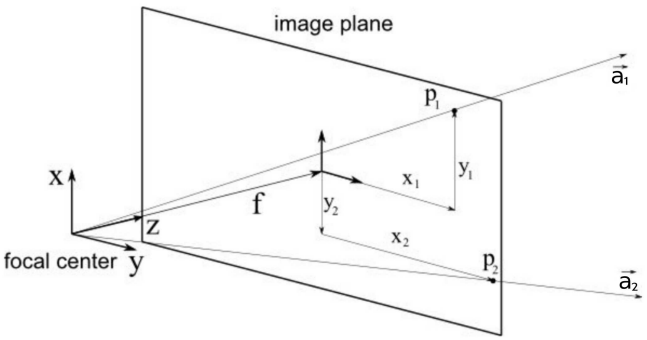

Figure 4.17: An example of the vectors involved. After a pure rotation, the point p_2 should ideally be projected back to p_1, as if no rotation occurred. *Source:* [51]

This solutions offers more accuracy than the Planar Derotation, but it is more complex and computationally expensive, especially the original one proposed in the work in [51], where they use 3 rotational matrixes for each IMU sample, and the total rotation is computed by multiplying all of them.

For example, if a gyroscope with sampling frequency of 800 Hz is used (as the one in this work), and the slicing window is of 50 hz, the total number of rotations for each edge image is 16, which means that the algorithm has to compute for 16 times the total rotation matrix (consisting of a 3x3 matrix for roll, pitch, and yaw). The number of operations for a matrix multiplication is $O(n^3)$, so considering that the 3 matrixes are multiplied together, then for each gyro sample and for each detected feature, the number of operations could be consistant.

On the other hand, modern embedded platforms such as the Raspberry Pi 5, which is used in this work, are capable of executing around 18 GFLOPS, which means that the algorithm would only require hundreds of *microseconds* to compute the derotation for hundreds of features in a single slice, resulting in a negligible computational cost with respect to the total time required to perform one iteration of the algorithm.

However, there are other methods that could perform even better and faster, such as the one described next.

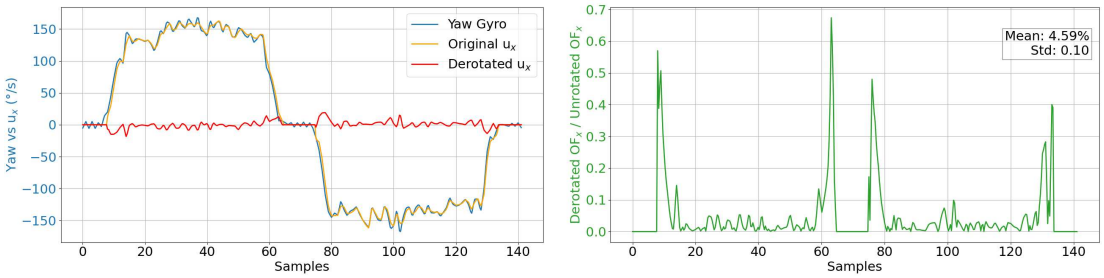

Figure 4.18: Results of the Quaternion method, applied onto a sequence of slices. The camera has been mounted on a servo motor performing rotations @ 150 deg/s. (Left) Comparison of the yaw gyro data (blue), the horizontal OF $\mathbf{u}_x$ (orange) and the derotated OF $\mathbf{u}_{der,x}$, (red). (Right) Ratio between the derotated and the raw OF.

4.4.4 Derotation using Sphere Reprojection

This method is an enhancement of the *Planar Derotation* method, in fact it is based on the same principle, but it considers also the 3$^{\text{rd}}$ component of the OF vector $\mathbf{P}$.

It directly applies the eq. (2.3), while computing the real OF vector $\mathbf{P}$ by knowing its image projection $\mathbf{u}$ and the coordinates of the feature in the image plane $\mathbf{a}$.

The equation describes the following scenario: a relative motion between the observer and the environment at depth D, can be represented as the P vector in the sphere of unit radius centered in the camera (as if the optical center was the origin of the sphere). By reconstructing the real OF vector $\mathbf{P}$, the rotational component can be removed, and the translational component can be obtained, by subtracting the rotational component from the real OF vector.

A generic vector $\mathbf{P}$ in the sphere can be expressed as:

$$\mathbf{P} = \mathbf{P}' - \mathbf{P}'' \tag{4.7}$$

where $\mathbf{P}'$ is the component of the vector that is parallel to the bi-dimensional $\mathbf{u}$ vector in the image plane, and $\mathbf{P}''$ is the component of the vector that is orthogonal to the $\mathbf{u}$ vector, which is not considered in the planar derotation method.

The $\mathbf{P}'$ component can be expressed as:

$$\mathbf{P}' = \begin{bmatrix} \frac{\mathbf{u}_x}{\|\mathbf{a}\|} \\ \frac{\mathbf{u}_y}{\|\mathbf{a}\|} \\ 0 \end{bmatrix} \tag{4.8}$$

This vector is simply a scaled version of the $\mathbf{u}$ vector, projected on the sphere, and in fact the 3rd component of the vector is set to 0, since the computed OF vector is projected on the image plane.

The $\mathbf{P}''$ is the coplanar component of the vector, and it can be expressed as:

$$\mathbf{P}'' = (\mathbf{P}' \cdot \mathbf{d}) \cdot \mathbf{d} \tag{4.9}$$

As the eq. (4.7) states, the real OF vector $\mathbf{P}$ can be obtained by subtracting the $\mathbf{P}''$ component from the $\mathbf{P}'$ component, resulting in the following equation:

$$\mathbf{P} = \mathbf{P}' - (\mathbf{P}' \cdot \mathbf{d}) \cdot \mathbf{d} \tag{4.10}$$

This way, the original OF vector $\mathbf{P}$ can be obtained, and the rotational component can be removed with the same expression described before :

$$\mathbf{P}_{trans} = \mathbf{P} - \mathbf{R} \times \mathbf{d}$$

The $\mathbf{R}$ vector has been obtained by averaging the gyroscope data in the slicing window, and the $\mathbf{d}$ vector is the same as the one used in the eq. (4.10).

Finally, as will be described in the next section (see chapter 3), the depth can be estimated by the the vector $\mathbf{P}_{trans}$, so the image projection $\mathbf{u}_{trans}$ of the derotated vector is not needed, because the depth is estimated directly from the 3D vector.

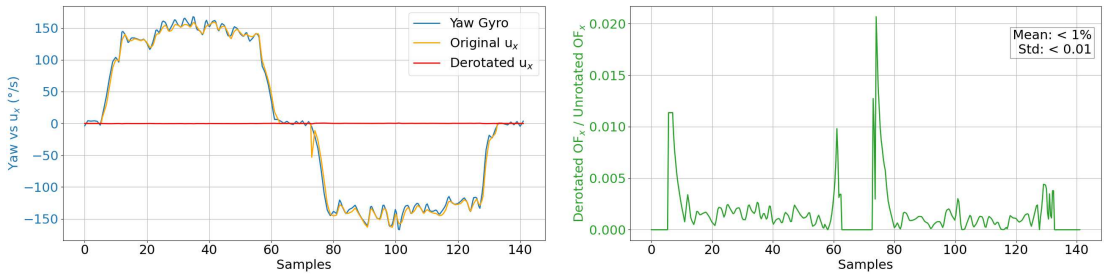

Figure 4.19: Results of the *Spherical Reprojection* method, applied onto a sequence of slices. The camera has been mounted on a servo motor performing rotations @ 150 deg/s. (Left) Comparison of the yaw gyro data (blue), the horizontal OF $\mathbf{u}$ (orange) and the derotated OF $\mathbf{u}_{der,x}$, (red). (Right) Ratio between the derotated and the raw OF.

By comparing the results of sections 4.4.2 to 4.4.4, it can be seen that the *Spherical Reprojection* method offers the best results, as it is able to remove more

than 99% of the rotational component from the OF vectors, while the other methods are less capable, especially during fast transitions, that generates those spikes in the ratio plots.

On top of that, the last method is really fast and efficient, generating a 3D derotated vector that is directly used to estimate the depth, without the need of projecting it back to the image plane, as the other methods do, which is a further advantage of this method.

4.5 Altitude Estimation

After the *Derotation* process has been applied to the OF vectors, now eq. (2.3) can be directly applied, considering that now the **P** vector (OF in 3D space) only corresponds to its *translational component*. This is reasonable in numerical terms, since less than 1% of the rotational component has been removed (see section 4.4.4).

Now, it is possible to rewrite eq. (2.3) as follows:

$$\mathbf{P}(\theta,\psi) = \mathbf{P}_{trans}(\theta,\psi) = \frac{\mathbf{V} - (\mathbf{V}\cdot\mathbf{d}(\theta,\psi))\cdot\mathbf{d}(\theta,\psi)}{D(\theta,\psi)} \tag{4.11}$$

This equation represents the base for the altitude estimation. The idea is in fact to extract the **Depth** D from eq. (4.11), and then use it to estimate the altitude over ground. The translation vector $\mathbf{V}$ is known from the *airspeed sensor*, that although only capable of measuring the airspeed of the plane, it can be precise enough when the wind speed is not too strong. Mathematically, since we are dealing with a scalar value only (which we denote as s), the vector $\mathbf{V}$ is represented as follows:

$$\mathbf{V} = \begin{bmatrix} s \\ 0 \\ 0 \end{bmatrix}$$

where the x component of the vector is the airspeed sensor reading s, and the other two components are zeroed.

4.5.1 Method

A simplified scheme of the scenario is showed in fig. 4.20, where the involved quantities are highlighted.

Computing the **Depth** D from eq. (4.11) is trivial, since all the quantities are known. So, for each feature i in the sparse OF field, the depth D_i can be computed as follows:

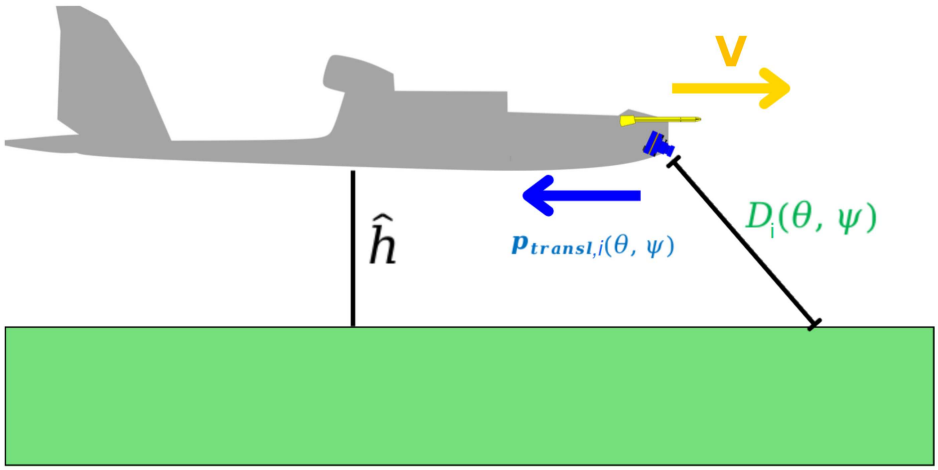

Figure 4.20: Plane over ground. $\hat{h}$ is the *estimated altitude*, D_i is the generic *depth* of feature i, and $\mathbf{p}_{transl,i}$ is the *translational component* of the OF vector of feature i. The velocity vector V is known from the *airspeed sensor*.

$$D_i = \frac{|\mathbf{V} - (\mathbf{V} \cdot \mathbf{d}_i) \cdot \mathbf{d}_i|}{|\mathbf{P}_{trans,i}|} \tag{4.12}$$

For simplicity, the angles θ and ψ are omitted in the notation.

Once the depth D_i is computed for each feature i, we have a data structure that can be referred to as **Sparse Depth Map**, where each element is an estimation of the depth at a specific location in the image. The ideal case would be to have a *dense depth map*, but it would be computationally expensive and not strictly necessary for the purpose of this work.

Given the depth map, we proceed with estimating the altitude by transforming the direction vectors associated with each feature from the camera frame to the inertial frame. These transformations involve rotation matrices that account for the camera inclination angle, as well as the roll and pitch angles of the platform.

For each feature i in the sparse OF field, we begin by computing the direction vector in the camera frame, $\mathbf{d}_i^{\text{cam}}$, as previously defined:

$$\mathbf{d}_i^{\text{cam}} = \frac{\mathbf{a}_i}{\|\mathbf{a}_i\|}$$

where $\mathbf{a}_i$ is the vector from the camera's optical center to the feature in the image, expressed in pixel coordinates.

Next, we apply the transformation matrix $T_{\text{cam}\to\text{body}}(\phi)$, which converts the direction vector from the camera frame to the body frame. The matrix $T_{\text{cam}\to\text{body}}(\phi)$

is a function of the camera inclination angle ϕ, which has its zero value when the camera is facing forward, and positive values represent downward inclination. In our case, for the reasons explained in chapter 3, the camera is inclined by an angle of 45 degrees.

The transformation matrix is:

$$T_{\text{cam}\to\text{body}}(\phi) = \begin{bmatrix} 0 & -\sin\phi & \cos\phi \\ 1 & 0 & 0 \\ 0 & \cos\phi & \sin\phi \end{bmatrix}$$

Thus, the direction vector in the body frame is computed as:

$$\mathbf{d}_i^{\text{body}} = T_{\text{cam}\to\text{body}}(\phi) \cdot \mathbf{d}_i^{\text{cam}}$$

Once the direction vector is expressed in the body frame, we transform it into the inertial frame using the rotation matrix $T_{\text{body}\to\text{inertial}}(\theta_{\text{roll}}, \theta_{\text{pitch}})$, which accounts for the roll angle θ_{roll} and pitch angle θ_{pitch} of the platform.

The attitude angles θ_{roll} and θ_{pitch} are obtained from the **Pixhawk** flight controller, which provides the platform's orientation in the inertial frame.

The transformation matrix $T_{\text{body}\to\text{inertial}}(\theta_{\text{roll}}, \theta_{\text{pitch}})$ is given by:

$$T_{\text{body}\to\text{inertial}}(\theta_{\text{roll}}, \theta_{\text{pitch}}) = \begin{bmatrix} \cos\theta_{\text{pitch}} & \sin\theta_{\text{pitch}}\sin\theta_{\text{roll}} & \sin\theta_{\text{pitch}}\cos\theta_{\text{roll}} \\ 0 & \cos\theta_{\text{roll}} & -\sin\theta_{\text{roll}} \\ -\sin\theta_{\text{pitch}} & \cos\theta_{\text{pitch}}\sin\theta_{\text{roll}} & \cos\theta_{\text{pitch}}\cos\theta_{\text{roll}} \end{bmatrix}$$

Therefore, the direction vector in the inertial frame is computed as:

$$\mathbf{d}_i^{\text{inertial}} = T_{\text{body}\to\text{inertial}}(\theta_{\text{roll}}, \theta_{\text{pitch}}) \cdot \mathbf{d}_i^{\text{body}}$$

In the inertial frame, the altitude is related to the z-component of the direction vector, which gives the cosine of the angle φ_i between the direction vector and the vertical (z-axis). The cosine of the angle φ_i is simply the third component of the direction vector in the inertial frame:

$$\cos\varphi_i = \mathbf{d}_i^{\text{inertial}}[2]$$

Thus, for each feature i, the altitude h_i is estimated as:

$$h_i = D_i \cdot \cos\varphi_i$$

Finally, the overall altitude $\hat{h}$ is estimated by averaging the altitudes from all valid features in the sparse depth map:

$$\hat{h} = \frac{1}{N}\sum_{i=1}^{N} h_i = \frac{1}{N}\sum_{i=1}^{N} D_i \cdot \cos\varphi_i$$

where N is the total number of detected features in the sparse OF field.

This method provides a robust estimation of the platform's altitude by leveraging both the sparse depth map and the orientation of the platform, taking into account the camera inclination, roll, and pitch angles.

A simpler approach could have been used, by simply averaging the depth values and multiplying by the cosine of the camera inclination angle. However, this would not take into account the contribution of the position of the features in the image, which can provide a more accurate estimation of the altitude. In fact, a feature in the bottom corner of the image will usually a smaller depth value than a feature in the center of the image, and this information could be used to improve the altitude estimation, at a cost that again is not computationally expensive, and hence negligible.

4.5.2 Filtering

The estimated altitude computed at each new iteration, as described in section 4.5.1, is subject to noise and variations due to the nature of the entire algorithm, which comprises a lot of uncertainties, such as the quality of the OF vectors, the accuracy of the airspeed sensor and the attitude angles, and the goodness of the algorithm itself.

To mitigate these issues and to feed the controller with more stable values and less fluctuations, a filter is applied to the *Raw Altitude* $\hat{h}$.

Many filters could be used, each with its own advantages and disadvantages. Usually, the computational times are not a problem, since the frequency of the data is not too high, and the processing power is more than enough to handle the filtering.

For simplicity and efficiency, a **First Order Low-Pass Filter** is used, which is a simple filter that attenuates high-frequency components of the signal, leaving only the low-frequency components. This is useful in this case, since the high-frequency components are usually the noise and the fluctuations, while the low-frequency components are the actual variations of the altitude.

The filter recursively updates the altitude estimate by blending the previous filtered value with the current measurement. This is governed by the following formulation:

Let $\hat{h}_{\text{current}}$ represent the current altitude measurement and $\hat{h}_{\text{filtered}}(t - \Delta t)$ represent the filtered altitude from the previous time step. The filtered value at time t, denoted by $\hat{h}_{\text{filtered}}(t)$, is updated using the following steps:

1. Compute the error or difference between the current measurement and the previous filtered value:

$$v_k = K \cdot (\hat{h}_{\text{current}}(t) - \hat{h}_{\text{filtered}}(t - \Delta t))$$

where v_k is the update increment, K is a the filter constant, and Δt is the time step between two consecutive measurements, in this case the slicing window where the events have been processed.

2. Update the filtered altitude by applying the increment v_k over the time step Δt:

$$\hat{h}_{\text{filtered}}(t) = v_k \cdot \Delta t + \hat{h}_{\text{filtered}}(t - \Delta t)$$

This formulation smooths the altitude estimate by gradually incorporating the current measurement $\hat{h}_{\text{current}}$, ensuring that the changes in altitude over time are more stable and less sensitive to noise.

The parameter K controls the strength of the filtering, where a smaller K results in heavier smoothing and a slower response to changes in the input signal, while a larger K allows the filtered value to react more quickly to changes in altitude.

Empirically, a value of $K = 2.5$ has been chosen to provide a good balance between smoothing and responsiveness to changes in altitude, resulting in a stable and accurate altitude estimate.

A visual result of the applied filter is shown in fig. 4.21, where the ground truth is compared to the raw altitude and the filtered altitude.

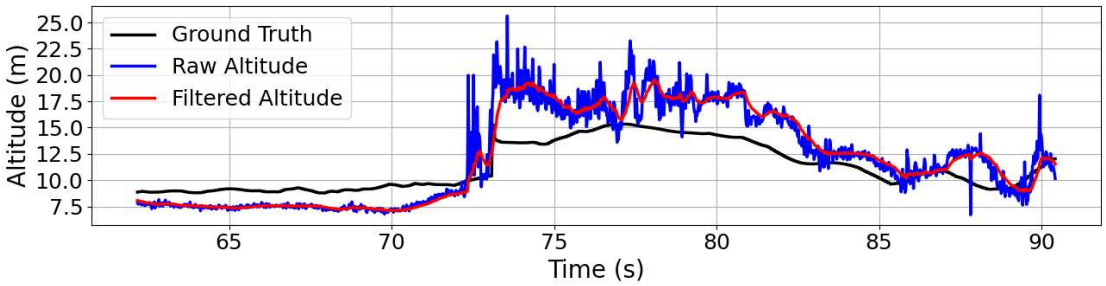

Figure 4.21: Comparison between the ground truth (black), the raw altitude (blue), and the filtered altitude (red). The filtering process is able to smooth the raw altitude and provide a more stable and accurate estimate of the platform's altitude. The data shown is from a section of a flight test.

4.5.3 Estimation in case of high roll angles

The method described in section 4.5.1 works well when the roll angle of the platform is within reasonable values, as the camera is facing mostly downwards and the OF vectors are mostly in the vertical direction.

In fact, thanks to the transformation described in section 4.5.1, the depth values are computed correctly and compensated for the camera inclination angle. However,

when the roll angle is high, the camera is facing sideways, and so the visual scene is pointing at far distances, which can lead to inaccuracies in the depth estimation.

A method to mitigate this issue is to apply a *Gaussian Weighting* to the depth values, based on the roll angle of the plane. When the roll angle is high, the depth values are weighted by a Gaussian function that gives more importance to the depth values closer to the part of the frame that is facing downwards, and less importance to the depth values that are far away from the center of the frame.

However, some tests proved that this method is not always effective, and in some cases, it can even worsen the estimation. This is due to the fact that sometimes the features that are referred to the ground, are not reliable and they are not enough to provide a good estimation of the altitude (see fig. 4.22). In these cases, the weighting can give more importance to these unreliable features, leading to a worse estimation.

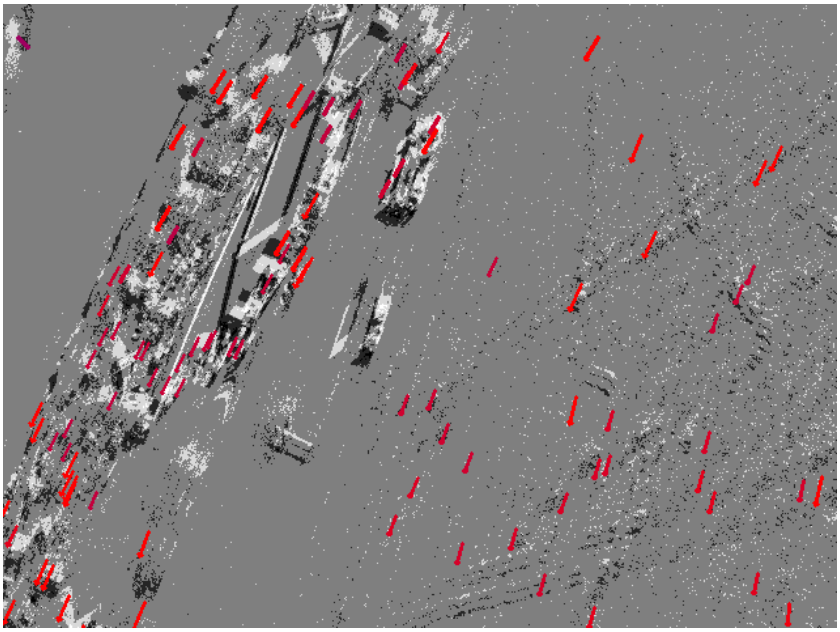

Figure 4.22: Edge image generated in case of high roll angle (> 40 degrees). The majority of the features and reliable points are not referred to the ground, instead they are referred to the buildings that are far away, resulting in a bad estimation of the altitude.

Hence, the method is not used in the final implementation, and the altitude estimation is performed as described in section 4.5.1, without any additional weighting, as it results in a more stable and accurate estimation, although it is not perfect in all conditions where the roll angle is over ~ 35 degrees.

4.6 Control

After the altitude estimation has been performed (see section 4.5), the system for controlling the landing and flight stablization can be implemented.

As described in section 5.1, the goal of the mission was to flight and land autonomously.

The control system in this project is implemented within the **Pixhawk** flight controller. The Pixhawk is a module that is capable of running the PX4 firmware, which is an open-source autopilot system (see chapter 6). Depending on the type of system chosen (in this case a *Fixed Wing Drone*), the firmware will provide the necessary control loops to stabilize the drone in flight and to control the landing, depending on the chosen mission.

For each of the possible states of the drone, various sources of input can be configured, such as GPS, IMU, and other sensors. In our case, the majority of them were left to the GPS, but for the *Z- coordinate* of the *Local Reference System*, the **Vision** has been used as the primary source of information.

4.6.1 Altitude Control: TECS Controller

For *Fixed Wing Drones*, the **TECS** (Total Energy Control System) controller is the de-facto standard for low-level control of altitude and speed, and it is implemented in the PX4 used in this project. The TECS controller adjusts the throttle and pitch of the drone to control the reference altitude and speed based on the commands from the mission planner [53].

The controller operates on the principle of managing the total energy of the aircraft, which consists of two components: *potential energy* and *kinetic energy*.

Potential and Kinetic Energy

1. **Potential Energy (PE)**: This is the energy associated with the altitude of the drone. It is given by:

$$PE = mgh$$

where: - m is the mass of the drone, - g is the acceleration due to gravity, - h is the altitude of the drone.

2. **Kinetic Energy (KE)**: This is the energy associated with the speed of the drone and is given by:

$$KE = \frac{1}{2}mv^2$$

where: - v is the velocity (airspeed) of the drone.

Total Energy (TE)

The total energy of the drone is the sum of the potential energy and kinetic energy:

$$TE = PE + KE = mgh + \frac{1}{2}mv^2$$

TECS aims to manage this total energy by adjusting the throttle and pitch to balance the energy distribution between the two components. By doing so, it achieves the desired altitude and speed.

Control Strategy

Two key parameters are maintained:

- Specific Total Energy (STE): This represents the total energy per unit mass and is defined as:

$$STE = gh + \frac{1}{2}v^2$$

- Specific Energy Rate (SER): This is the rate of change of the specific total energy, controlled by adjusting the throttle and pitch. It is defined as:

$$SER = g\dot{h} + v\dot{v}$$

where: - $\dot{h}$ is the rate of climb (vertical velocity), - $\dot{v}$ is the change in airspeed (acceleration).

Two control loops are implemented:

1. Throttle Control (for Energy Rate Management): The throttle is adjusted to manage the total energy rate (SER). By increasing throttle, the drone can gain more kinetic energy (speed), or if the drone is climbing, it can compensate for the loss in potential energy (see section 4.6.1).

2. Pitch Control (for Energy Distribution): The pitch angle is adjusted to manage how the total energy is distributed between kinetic and potential energy. A nose-up pitch increases potential energy (altitude) at the cost of airspeed, while a nose-down pitch does the opposite (see section 4.6.1).

TECS Equations

The TECS controller adjusts throttle T and pitch θ to meet the desired altitude h_{ref} and airspeed v_{ref}. The control laws for throttle and pitch are derived from the following equations:

- **Throttle Control**:

$$T = K_T \left(g(h_{\text{ref}} - h) + \frac{1}{2}(v_{\text{ref}}^2 - v^2) \right)$$

where K_T is a throttle gain constant, and h_{ref}, v_{ref} are the reference altitude and airspeed, respectively.

- **Pitch Control**:

$$\theta = K_\theta \left(g(h_{\text{ref}} - h) - \frac{1}{2}(v_{\text{ref}}^2 - v^2) \right)$$

where K_θ is a pitch gain constant.

These equations ensure that both the energy rate (throttle control) and energy distribution (pitch control) are maintained to achieve the desired flight parameters.

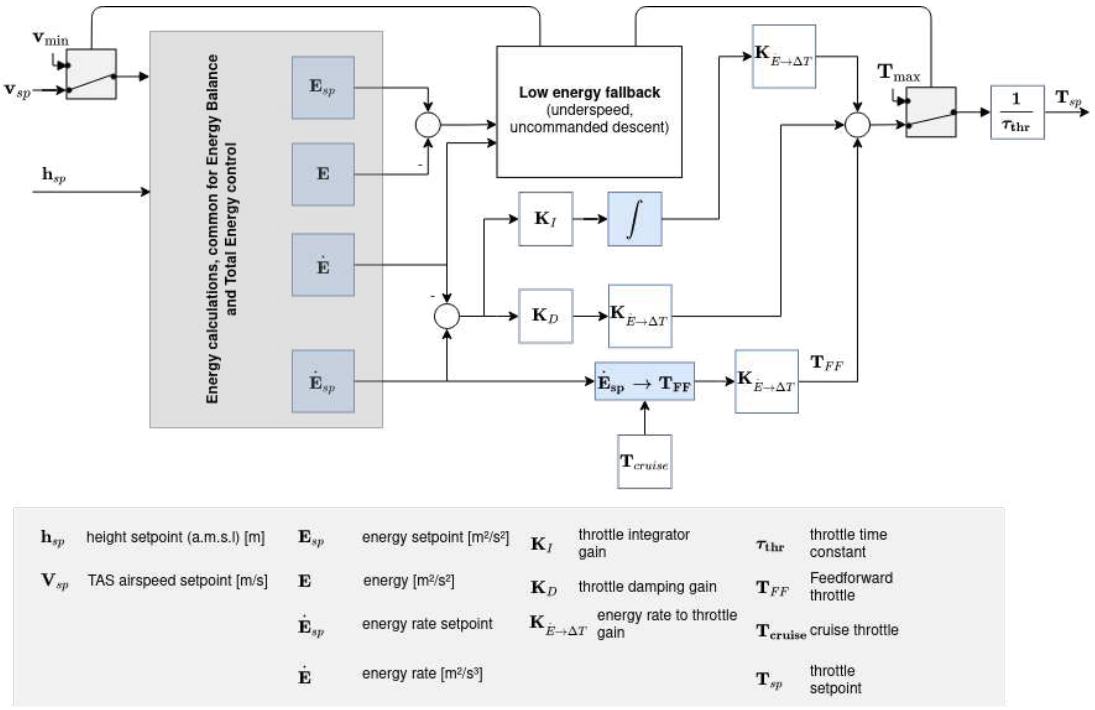

Figure 4.23: TECS Throttle Control

4.6.2 Attitude Control: PX4 Fixed-Wing Attitude Controller

The PX4 FW Attitude Controller uses a **Cascaded loop** structure to control the drone's orientation:

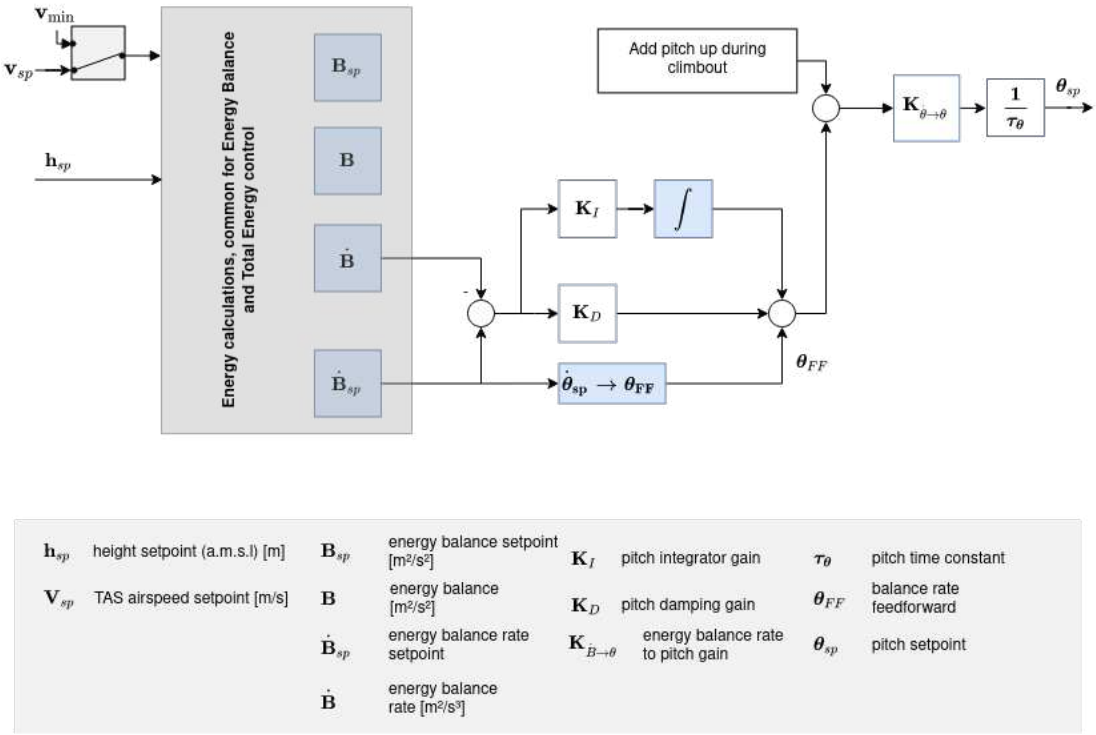

Figure 4.24: TECS Pitch Control

- **Outer Loop**:
  - Computes the error between the desired attitude (setpoint) and the estimated attitude.
  - This error is multiplied by a proportional gain to generate the desired angular rate setpoint.
- **Inner Loop**:
  - Computes the error between the angular rate setpoint and the measured angular rates.
  - A PI (Proportional-Integral) controller then calculates the required angular acceleration to minimize this error.

The control surfaces (e.g., ailerons, elevator, rudder) are adjusted based on the computed angular accelerations. Since control surfaces are more effective at higher speeds, the controller scales its output based on airspeed measurements when available [54].

4.7 Adaptive Time Slicing

Slicing by a fixed-time window could present the problems described in section 4.1.1, where the event batch could contain no reliable information due to the time window being too small or too large for the captured scene. Another limitation of the fixed-time window slicing is that it does not adapt to the scene's dynamics, and the resulting displacement between the slices could be too large, especially when the plane approaches the ground.

To overcome this limitation, an adaptive time slicing mechanism has been implemented.

Other methods have been considered to produce adaptive mechanism, such as the one in [55], but they only sliced the stream based on a pre-defined number of events threshold. This method would indeed produce slices that are able to adapt to the density of information in the scene, but it would not be able to distinguish between fast scenes with little information and slow scenes with a lot of information, as they would both produce roughly the same event rate in the same time window.

The *ABMOF* mechanism proposed in [5] adjusts the slicing duration based on the OF data by calculating the average match distance from a histogram of motion vectors. This is compared to a predefined threshold (half the search radius), and a simple bang-bang controller is used to adjust the slice duration by a fixed percentage. While this method provides a straightforward feedback control, it suffers from potential oscillations and lacks fine-grained control due to its fixed adjustment step. Their method works for scenes that are mostly static, since they based their OF algorithm on the matching of 2 past slices, instead of the current one and the previous one. Their search radius is reported to be no more than 12 pixels, which is not enough for our application, where the scene is dynamic and the objects are moving fast. Also, they have used raw OF vectors, which do not take into account the ego-motion of the camera, which is a crucial aspect of our application.

4.7.1 Algorithm

The algorithm we developed is an adaptation of the classical PI controller, where the error is the difference between the average derotated OF magnitude and a predefined setpoint.

The idea behind it is to adjust the slicing duration based on the average OF magnitude, so that the slices contain a similar amount of information, regardless of the scene's dynamics. Large displacements are intrinsically related to a too wide slicing duration, while small displacements are related to a too narrow slicing duration, resulting in little information in the slices.

A block diagram of the Adaptive Time Slicing algorithm is shown in fig. 4.25.

Algorithm 1 PI-Based Adaptive Time Slicing Mechanism

Input:

- μ_{set}: Desired average OF vector magnitude (setpoint)
- T_0: Initial timing window
- K_P, K_I: PI controller coefficients (proportional, integral)
- τ_{min}, τ_{max}: Minimum and maximum allowed timing window values
- ϵ: Threshold for significant PI output changes
- ΔT: Adaptive timing step

Initialize:

- Integral error $I_e \leftarrow 0$

while Adaptive Slicing is Enabled **do**
  if New OF Vectors Available **then**
    Compute average OF vector magnitude μ
    Calculate the error $e \leftarrow \mu_{set} - \mu$
    Update integral term $I_e \leftarrow I_e + e$
    Compute the PI controller output: $u_{PI} \leftarrow K_P \cdot e + K_I \cdot I_e$
    if $|u_{PI}| > \epsilon$ **then**
      if $u_{PI} > 0$ **and** $T_0 < \tau_{max}$ **then**
        Increase timing window: $T_0 \leftarrow T_0 + \Delta T$
        Clamp $T_0 \leftarrow \min(T_0, \tau_{max})$
      else if $u_{PI} < 0$ **and** $T_0 > \tau_{min}$ **then**
        Decrease timing window: $T_0 \leftarrow T_0 - \Delta T$
        Clamp $T_0 \leftarrow \max(T_0, \tau_{min})$
      end if
      Update slicer timing window to T_0
    else
      Reset integral term $I_e \leftarrow 0$
    end if
  end if
end while

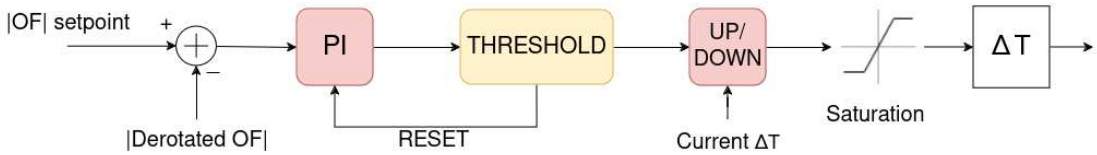

Figure 4.25: Block diagram of the Adaptive Time Slicing algorithm

The pseudo-code of the algorithm is shown in algorithm 1.

The choice behind the use of a PI controller is due to its simplicity and effectiveness in controlling systems with a high degree of uncertainty. The proportional term is used to provide a fast response to the error, while the integral term is used to eliminate the steady-state error.

This is very important in OF based applications, since the OF magnitude can be very noisy or can have a lot of outliers, which could lead to a lot of oscillations in the slicing duration, as in the case of the ABMOF algorithm.

Also, the use of a threshold ϵ to determine if the PI output is significant is crucial to avoid unnecessary changes in the slicing duration, and that act as a sort of hysteresis. It is important to allow changes in the slicing window only when the error is significant, and especially consistant over time, and this implementation allows for that.

However, the reset of the integral term to zero when the update takes place is also needed to avoid the integral windup problem, so whenever a trigger is detected, the controller starts again from scratch.

The saturation block is used to clamp the slicing duration to the minimum and maximum allowed values, to avoid the slicing duration to go out of bounds.

Table 4.1: Parameters for Adaptive Slicing Mechanism

Parameter	**Value**
K_P (Proportional Coefficient)	0.5
K_I (Integral Coefficient)	0.05
ϵ (PI Threshold)	10
μ_{set} (Setpoint for OF magnitude)	7 pixels
τ_{min} (Minimum Timing Window)	10 ms
τ_{max} (Maximum Timing Window)	25 ms
ΔT (Adaptive Timing Step)	1 ms
T_0 (Initial Timing Window)	20 ms

Table 4.1 shows the parameters used in the Adaptive Time Slicing mechanism. These have been tuned to provide a good balance between responsiveness and stability, and to avoid unnecessary changes in the slicing duration.

4.7.2 Application to the scenario

This mechanism is very useful in our application, since our mission is comprised of relative high altitudes (> 20 meters) and landings, where the plane is very close to the ground fig. 4.26.

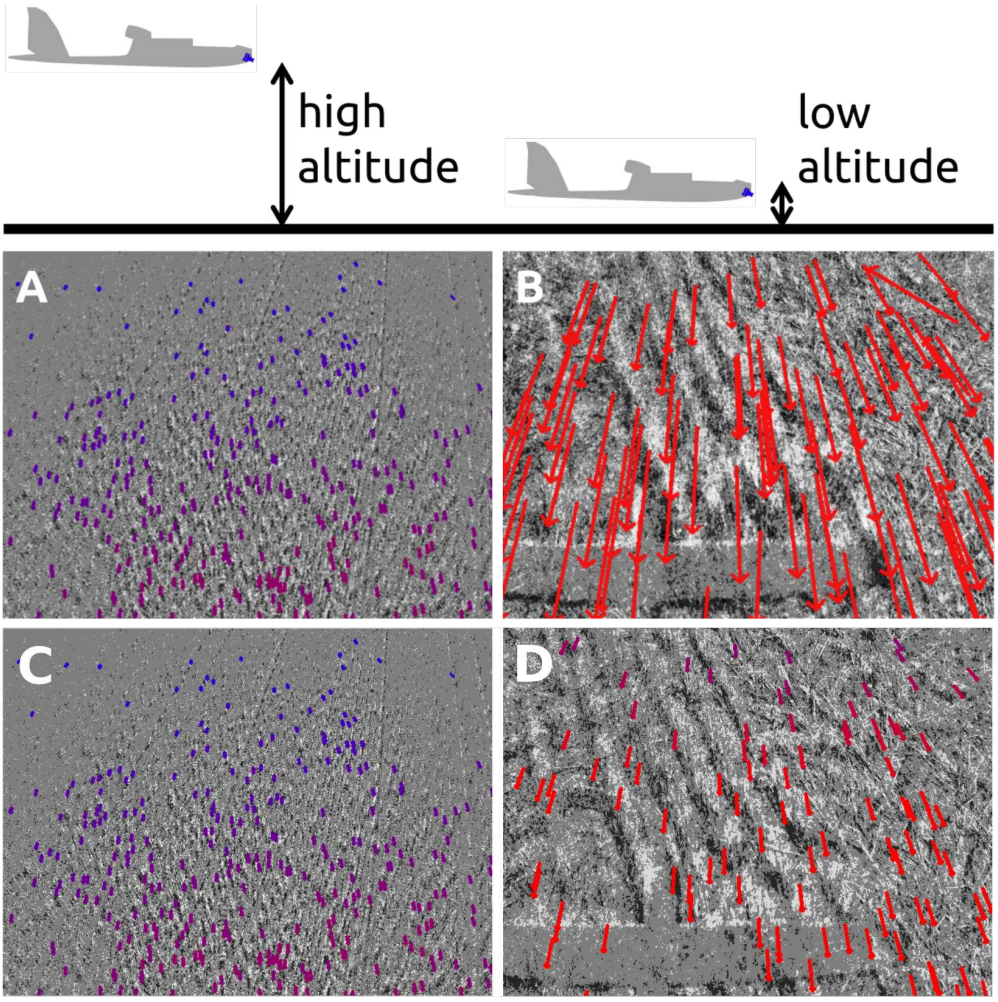

Figure 4.26: OF vectors at different altitudes, with fixed time slicing and Adaptive Slicing. (A) High altitude, *fixed time slicing*. (B) Low altitude, *fixed time slicing*. (C) High altitude, with Adaptive Slicing. (D) Low altitude, with Adaptive Slicing.

This very different scenarios would produce various visual information, and the adaptive slicing mechanism would be able to adapt to these changes, providing a more consistent slices to process.

Due to the close proximity to the ground, the plane would produce large

displacements between the slices (see fig. 4.26 (C)), which would result in *High Inaccuracy* in the estimation, due to the **Brightness Constancy** assumption being violated.

Instead, fig. 4.20 (D) shows the OF vectors when the adaptive slicing mechanism is enabled, resulting in shorter vectors due to the smaller slicing duration, which would provide a more accurate estimation of the plane's motion. A visual comparison between altitude estimation before landing is shown in fig. 4.27. Due to the high inaccuracy during the descent, the plane's altitude estimation is totally off, while the adaptive slicing mechanism provides a more accurate estimation.

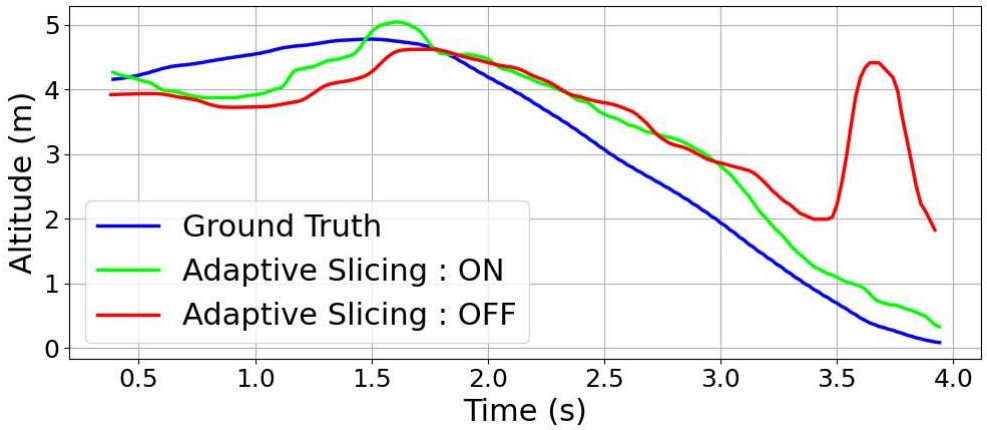

Figure 4.27: Altitude estimation during landing. The red curve showing the estimation *without* adaptive slicing results in a totally wrong estimation, while the blue curve shows the estimation *with* adaptive slicing, which provides a more accurate estimation.

In case of high altitudes however does not provide any difference in the OF vectors, as the plane is far away from the ground, and the slicing duration is already small enough to provide a good estimation of the plane's motion.

Figure 4.28 shown how the *Slicing Window* changes with the plane's altitude, and how the adaptive mechanism is able to adapt to the scene's dynamics.

To be noted that although there are similarities between the green curve (Altitude Estimation) and the blue curve (Slicing Window), the two are not directly related : in fact in case the flight speed of the aircraft would decrease while approaching the ground, the slicing window would roughly remain the same, while the altitude estimation would change. This is because at lower speeds, the magnitude of the estimated OF vectors would be smaller, hence the slicing window does not detect the need of a smaller window.

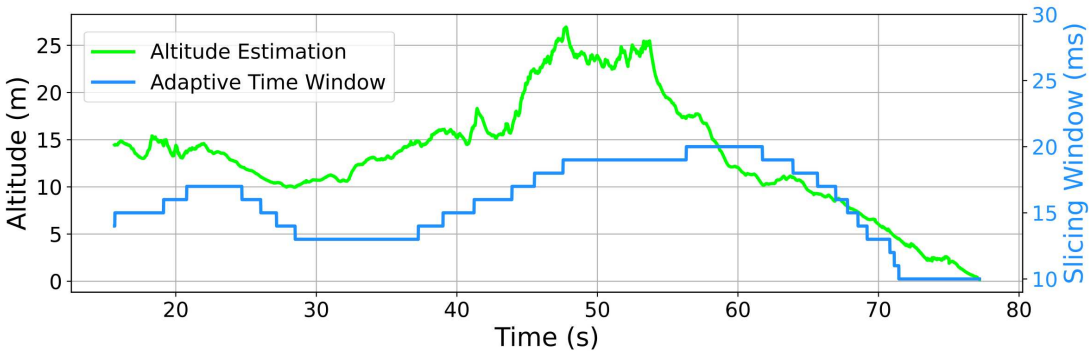

Figure 4.28: Adaptive Slicing mechanism in action.

Chapter 5

Results

The algorithm was initially tested on existing datasets for validation purposes, and then evaluated in a real-world scenario using the Bixler 3 (see chapter 6).

The most used dataset used at the beginning was the *UZH-FPV Drone Racing Dataset* [56], which contains a variety of challenging scenarios for event-based vision, such as fast motion, high-speed turns, and occlusions, all of them recorded from quadrotors, both indoor and outdoor.

The only difference is the camera used, which is a DAVIS346, with a resolution of 346x260 pixels, and a maximum event rate of 1 MEPS, which is significantly lower than the camera used in this work, causing a lower computational load.

This has been a valid platform for testing the algorithm in a variety of scenarios, but the real flight scenarios were significantly more challenging, due to the higher speed of the aircraft and the higher resolution of the camera.

As explained in section 3.3, the altitude estimation through vision has been tested in various light conditions, and with different camera configurations, to evaluate the performance of the algorithm in a real-world scenario, and to understand how the camera settings can affect the system's performance.

5.1 Mission Scenario

The evaluation of the algorithm under various camera configurations was carried out by performing the same flight mission for each test.

As shown in fig. 5.1, the scenario consisted in a first descent from 20 meters to 15, then a climb to 20 meters, and finally a landing phase. Before landing, the drone performs a turn with a maximum bank angle of 30°, in order to align with the landing strip. The total flight distance covered was 1.2 kilometers, with a maximum speed of 20 m/s. The entire flight duration was approximately 1 minute and 40 seconds, during which the drone maintained this controlled flight pattern,

allowing consistent testing of various camera configurations and sensor settings across each mission.

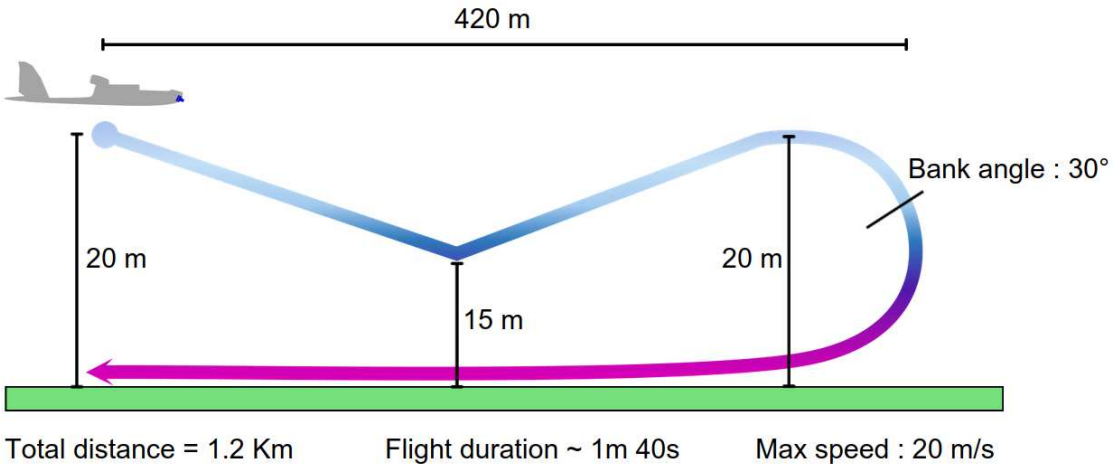

Figure 5.1: Mission scenario for the landing test.

The tests conducted during the day took place on a sunny afternoon at around 3 PM (see section 5.1 (Left)). The clear skies ensured sufficient light intensity for the camera to function effectively. In contrast, the nighttime tests were performed at approximately 8:30 PM, after sunset. However, the combination of ambient light from streetlights and other artificial sources provided adequate illumination for the camera to detect various events (see section 5.1 (Right)).

Figure 5.2: Comparison between the light conditions during the day and the night.

5.2 Ground Truth : RTK GPS

The ground truth for altitude estimation was obtained using a Real-Time Kinematic (RTK) GPS system, known for its centimeter-level accuracy. This high-precision system comprises two main components: a base station and a rover. The base station is positioned at a fixed location— in this case, on the ground and connected to a laptop— while the rover is mounted on the aircraft. The base station transmits correction data to the rover, enabling it to refine the GPS signal and deliver a more precise position estimate.

Figure 5.3: RTK GPS module for the base rover. A 3D printed case was designed to protect the module from the weather.

Evaluating the Estimation Accuracy

Figure 5.4 shows an example of one of the flights performed, where the OF-based estimation has been compared over time to the RTK GPS data.

The performance metric used to evaluate the algorithm was the Mean Relative Error (MRE), which is defined as the average of the absolute differences between the estimated and the ground truth values, divided by the ground truth value.

This choice is motivated by the fact that using a more classical metric such as the Mean Absolute Error (MAE) would have been misleading, as the altitudes reached during the flight were not constant, and the MAE would have been affected by the different altitudes reached during the flight.

In this case instead, the MRE provides a more accurate representation of the algorithm's performance, as it is normalized by the ground truth value.

As it can be observed, the highest MRE values are reached during the landing phase, where the drone is closer to the ground. This is due to the fact that since the ground truth is a small value, the error is more significant in relative terms.

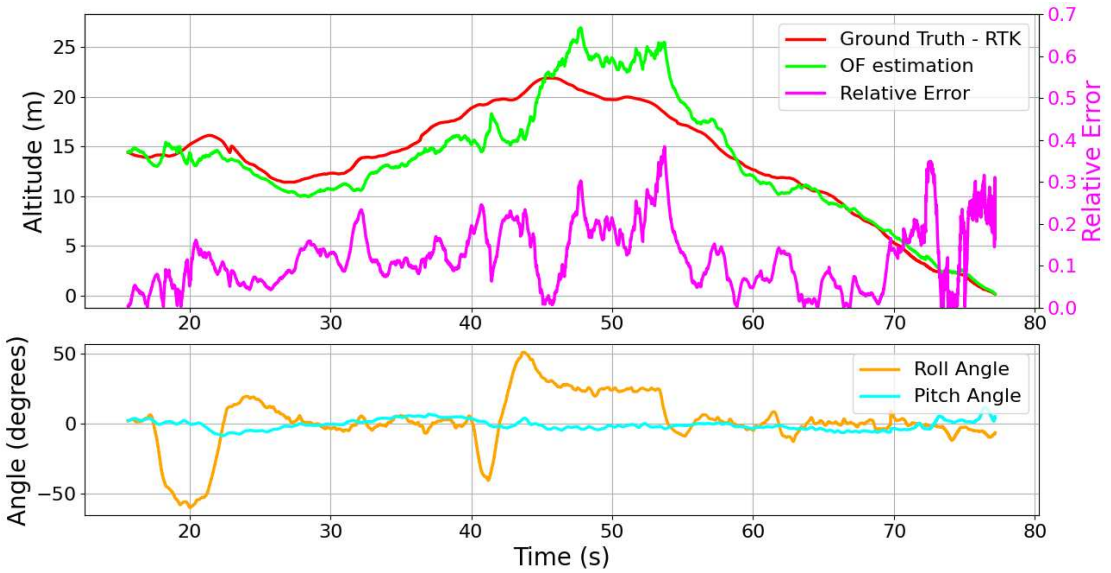

Figure 5.4: Altitude estimation accuracy and roll and pitch angles during a flight.

Another large source of error in this flight scenario is registered in the interval between 45 and 55 seconds, where the drone is performing a turn, and consequently the roll angle is quite high (over 30°). As mentioned in section 4.5.3, this is a challenging scenario for the algorithm, as the visual scene could contain no reliable information in the part of the frame pointing towards the ground, but rather is rich of information in the far distance.

5.3 Daylight conditions

The main purpose of this test was to understand how the algorithm behaves in different **event time resolutions**, described in section 3.3. It represents the minimum time interval between two consecutive events, and it is somehow related to a conventional camera's frame rate, but only referred to changes in brightness.

In our case, the DVXplorer camera offers a configurable event time resolution ranging from 66 μs to 10 ms, and the tests were carried out with the following configurations: 66 μs, 200 μs, 1 ms, 2 ms, and 10 ms.

The results of the tests are shown in fig. 5.5, where the MRE is plotted against the event time resolution.

As expected, the MRE decreases as the event time resolution increases, as the algorithm produces usually more events and hence the quality of the images are better.

Especially for the case of 10 ms, equivalent to a EFPS of 100, the slicing

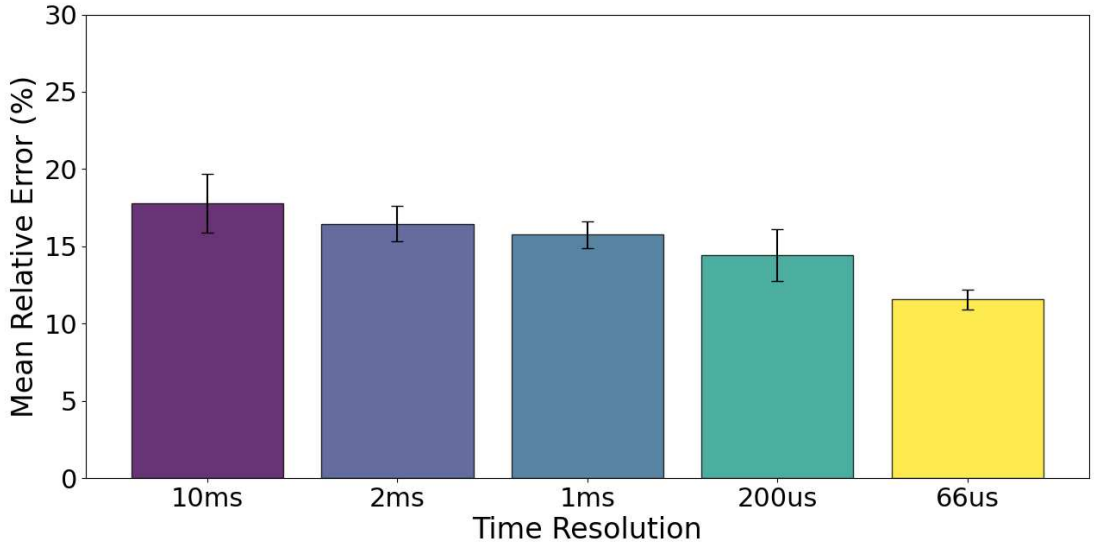

Figure 5.5: Estimation accuracy depending on the event time resolution.

algorithm could group event batches that are sampled in a time window of 10 ms, and basically the generated *Edge Image* could result in a simple binary image, where pixel only describe if an event at that location occurred or not, resulting in a less accurate estimation.

However, all these results are quite promising, as the MRE is always below 18%, which is a good result for a system that consists of a multiple steps, each with its error sources.

The best results are achieved with EFPS equal to 15000, which is the maximum value for the DVXplorer camera, and the MRE is around 12%.

5.4 Low light conditions

The second part of the study was focused on the performance of the algorithm in low light conditions, carried out after the sunset, aided by artificial light sources, such as streetlights.

Measuring the light intensity in that case was not possible due to the lack of a luxmeter, but due to the conditions, we could conclude that the light intensity was not more than 50 lux, which is considered a low light condition.

The parameter that is able to regulate the camera's threshold for generating events is called **Sensitivity**, and in our case 5 different values were tested: from *Very Low* to *Very High*. The higher the value, the more sensitive the camera is to light changes, and the more events are generated. However this could lead to a

higher noise level, and hence a lower quality of the images. A physical quantity that is related to the sensitivity is the *Voltage Threshold* applied to the comparator that generates the events, one for each pixel, that could be either applied to the positive side or negative, allowing for unbalanced sensitivity levels between the two polarities.

The vendor provides an easy way to regulate the levels in an even simpler way, by providing preset values for the sensitivity, that are the ones listed above, where each of them corresponds to a different voltage threshold. The **Sensitivity Threshold** parameter could assume 18 different values, from 0 t 17, and the correspondence between the sensitivity and the threshold is shown in table 5.1. The higher the threshold, the lower the sensitivity, and the less events are generated.

Sensitivity	**Threshold**
Very Low	15
Low	12
Medium	9
High	5
Very High	2

Table 5.1: Correspondence between the Sensitivity and the Threshold parameter.

A visual comparison between the same scenario captured by difference sensitivity is shown in section 5.4. As it can be observed, the higher the sensitivity, the more events are generated, and the more noise is present in the image.

The images on the top show a visual scene while the plane was flying over a road, and in all the images it seems the information is quite reliable, but in the one of the bottom, since the plane was flying over the grass, the low sensitivity levels are not able to provide a reliable visual information.

Since the majority of the scenes were similar to the one shown in the bottom, the accuracy tests show poor results in those cases.

A fixed EFPS value of 5000 was used for all the tests, because it offers a good compromise between the number of events generated and the quality of the images.

In order to evaluate the accuracy of the algorithm in low light conditions, 2 recordings were made, the first one called *Night 1* was carried out at around 8.20 pm, and the second one called *Night 2* was carried out at around 8.30 pm.

In every of the two recordings, the mission performed was very similar to the one described in section 5.1, with the only difference that this time at least a complete turn was performed for each of the sensitivity value, in order to have a more complete dataset.

However, the landing was performed only with the *Very High* sensitivity level, as it was the last.

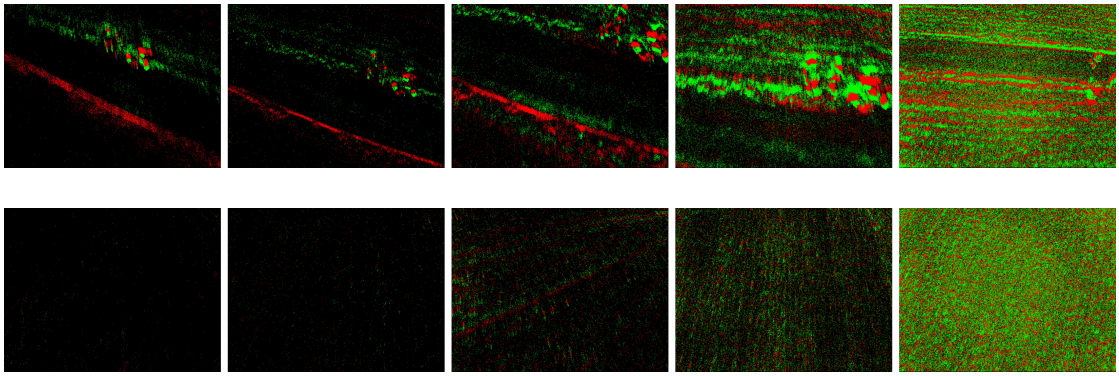

Figure 5.6: Comparison between the same scenario captured by different sensitivity levels. (Top) shows a visual scene while the plane was flying over a road, and (Bottom) shows the grass the plane was flying over. From left to right: *Very Low*, *Low*, *Medium*, *High*, *Very High*.

Figure 5.7 shows the mission scenario of the night tests, and it is clear that each of the repeated paths corresponds to a different sensitivity level.

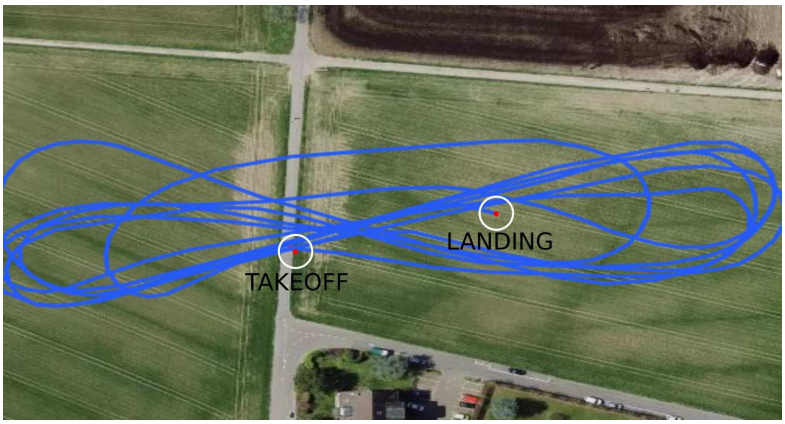

Figure 5.7: Mission scenario for the night tests.

It is also important to mention that in these low light condition scenarios, the test have been performed **offline**, meaning that the algorithm was not running on the aircraft, but the *.aedat4* files were recorded by the camera and then processed on the Raspberry Pi 5, after the flight.

The reason for this choice is that the scenario was quite challenging, and so for **safety reasons**, it was better to perform the tests offline, in order to avoid any possible crash due to the uncertainty of the generated events.

Night-1 Test

As specified above, this test was carried out at around 8.20 pm, and below a visual representation of the results are shown in fig. 5.9 and fig. 5.8.

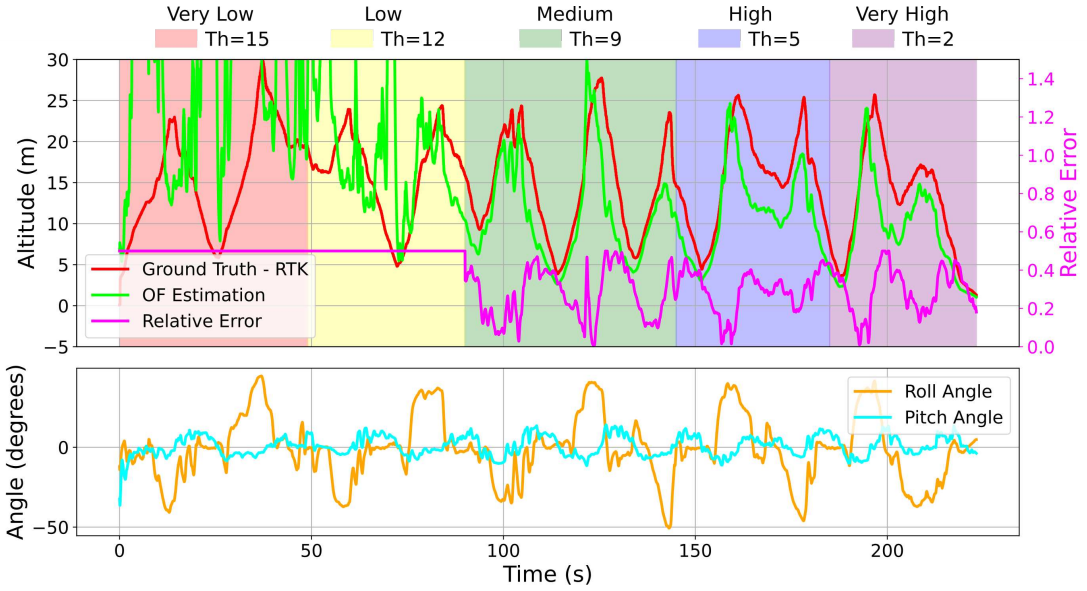

Figure 5.8: Altitude estimation accuracy during the *Night 1* test.

As it can be observed, as soon as the sensitivity value is changed from *Low* to *Medium*, the accuracies improve significantly, and are kept quite reliable for the higher sensitivity levels.

The error bars for the *Very Low* and *Low* sensitivity levels are so high that the algorithm is not able to provide a reliable estimation of the altitude, with values completely out of the range. Instead, the best performance is registered with the *Medium* sensitivity level, with a MRE of around 20%, which is still a high value, but it is a good result for a low light condition. Regarding the higher sensitivity levels, they still perform well, but their higher MRE is mostly due to the very high noise level.

Night-2 Test

The second test was carried out at around 8.30 pm, and the results are shown in fig. 5.11 and fig. 5.10.

The most significant difference with the previous test is that now the *Medium* sensitivity level is not able to provide a reliable estimation of the altitude, and the best performance is registered with the *High* sensitivity level, with a MRE of a little over 20%.

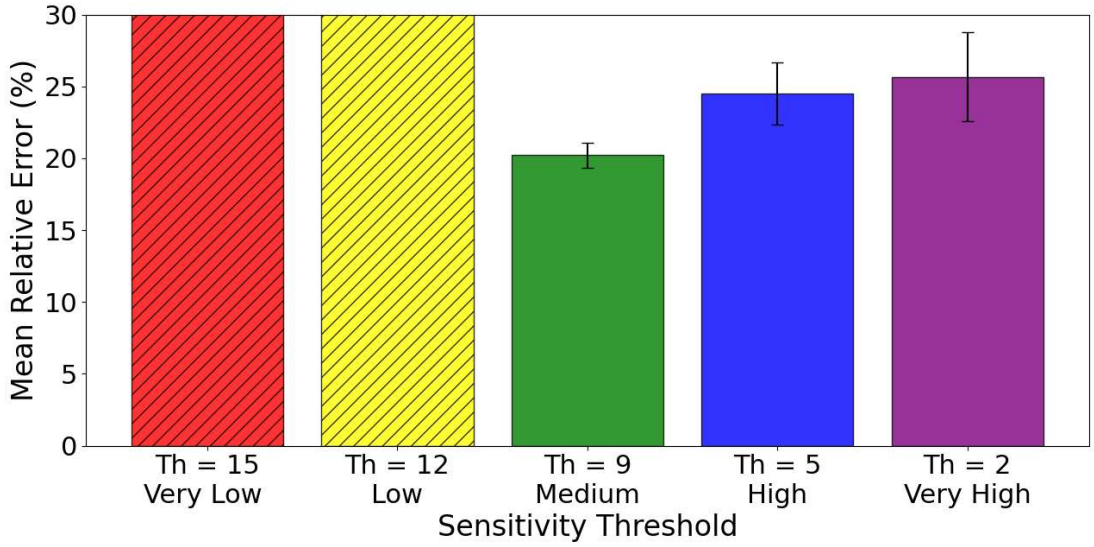

Figure 5.9: Results of the *Night 1* test, by varying the sensitivity levels.

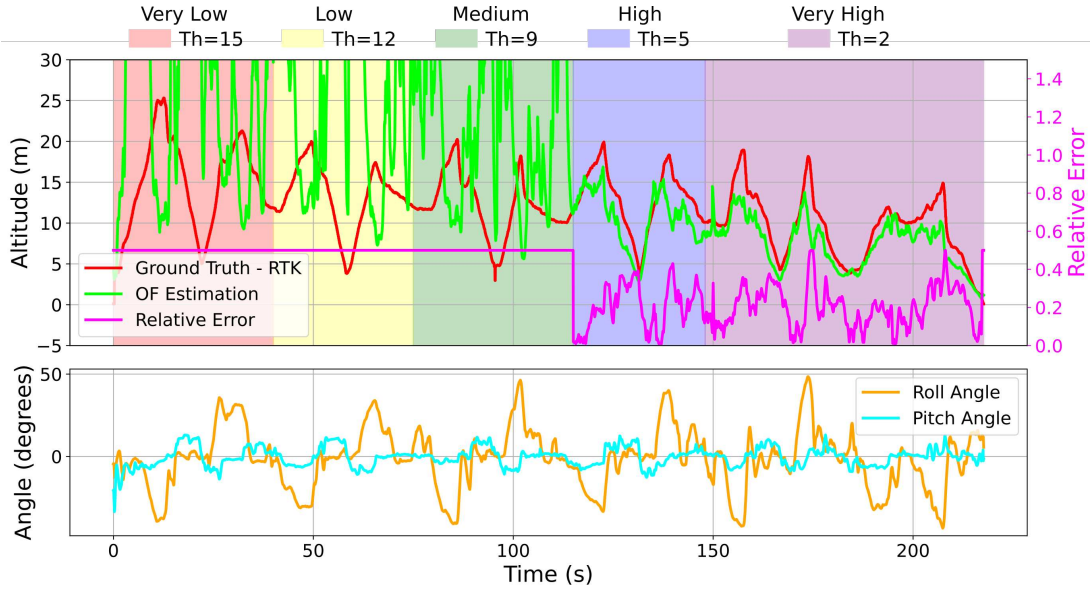

Figure 5.10: Altitude estimation accuracy during the *Night 2* test.

This is reasonable because as the light intensity decreases, the camera needs to be more sensitive to light changes, and hence the higher sensitivity levels are able to provide a better estimation.

Although only 10 minutes have passed between the two tests, the light intensity

has decreased quite significantly, and this is reflected in the results.

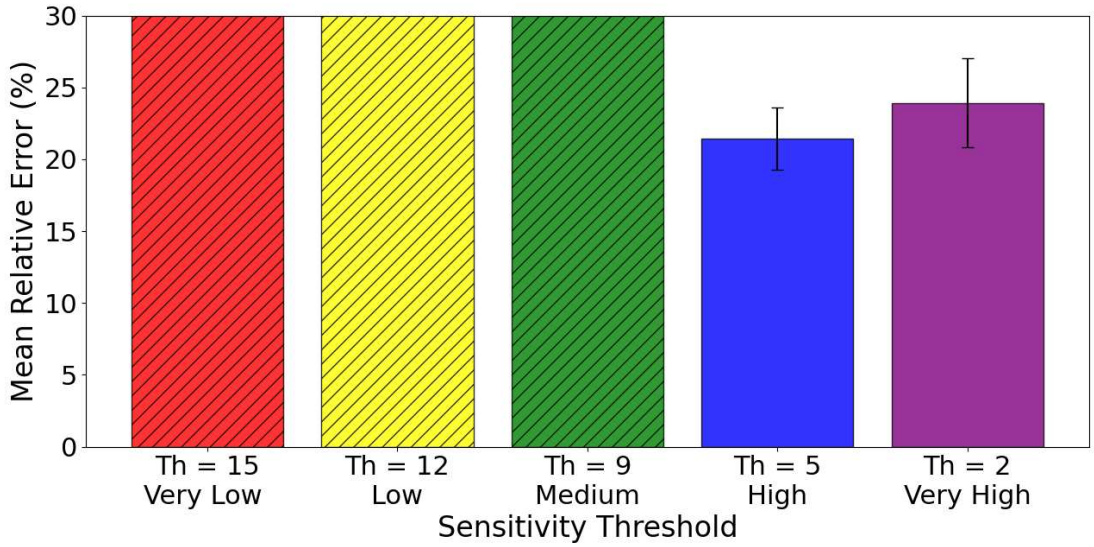

Figure 5.11: Results of the *Night 2* test, by varying the sensitivity levels.

5.4.1 Computational Performance

Along with the accuracy of the algorithm in estimating altitude over ground, a considerable amount of effort has been put into optimizing the algorithm's computational performance, in order to make it suitable for real-time applications.

In this regard, this section provides an overview of the computational speed in case of the *Daylight* test, since the majority of the events were registered in those scenarios, where due to the high light intensity, the camera was able to generate a high number of events.

As the main metric, the **Slack Time** (ST) has been used, which is defined as the time difference between the slicing window and the total time to process one edge image. Formally, the Slack Time is usually used in real-time systems to measure the time available for a task to be completed, before the next task is scheduled to run.

However, in this case, it could represents qualitatively assess the real-time performance of the code, because in absence of a fixed FPS or slicing window, using the total time to process one slice would not be enough.

The other important value to take into consideration, is the **Event Ratio**, expressed in **Million of Events per second (MEPS)**, which is the number of events generated by the camera in one second, on average.

The results of the computational performance relative to the case of event time resolution of 1 ms, are shown in fig. 5.12

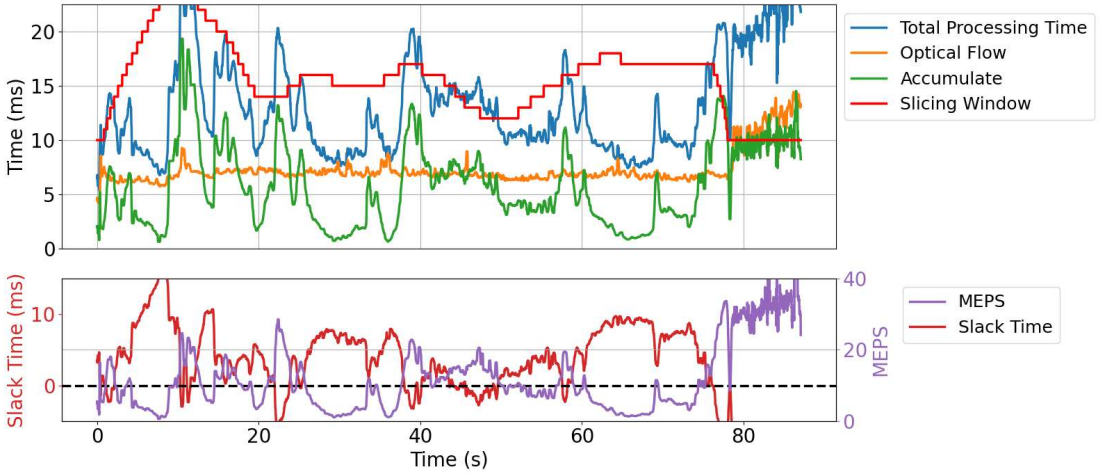

Figure 5.12: Computational performance of the algorithm with an event time resolution of 1 ms (1000 EFPS).

The first plot on the top shows how the total time to process each slice changes over time, compared to the *Slicing Window*, which varies over time as well, while the one on the bottom ST and MEPS over time.

It can be observed that for the entire duration of the flight, that lasts for about 95 seconds, the ST is almost always positive, meaning that the algorithm is able to process the events in real-time.

The main criticality is registered during the landing phase, where, due to the extremely high number of event rates (up to 40 MEPS), the ST is negative, meaning that the algorithm is experiencing some delays in estimating the altitudes, but still the algorithm is able to provide a reliable estimation.

This test is the heaviest in terms of computational capabilities, because as explained in section 3.3, the camera is able to provide a fixed sampling time up to 1 ms resolution, while for higher resolutions (5000 and 15000 EFPS) the DVXplorer tries to maintain the event resolution as high as possible but with some losses, end hence the number of events generated is lower.

The most expensive task is the *Edge Image* generation when the number of events is really high (computational complexity of $O(n)$), while the OF computation remains usually constant at less than 6 ms.

Instead, the simplest case is presented in the case of 10 ms, where smaller event ratios are generated (see fig. 5.13).

Here, the case is totally different, since the ST is always positive, even during the landing phase where the events are really high.

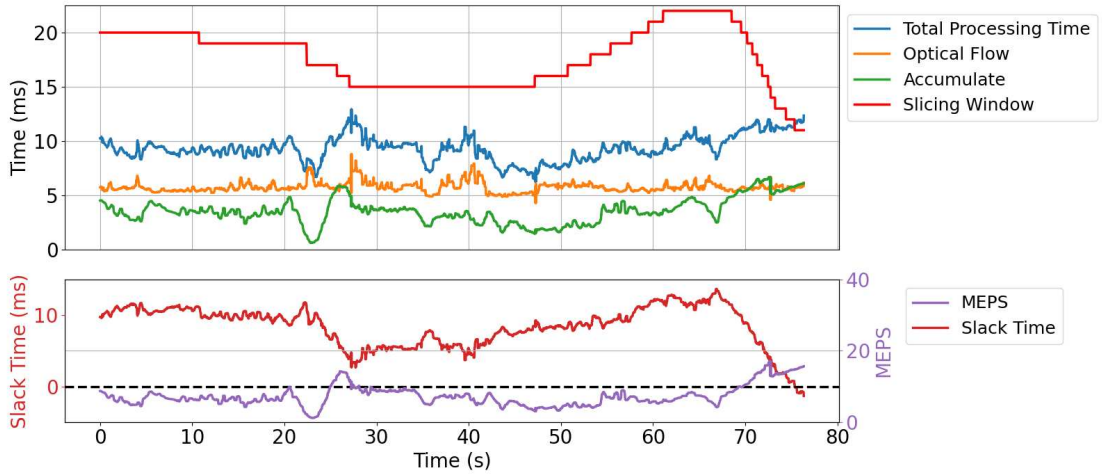

Figure 5.13: Computational performance of the algorithm with an event time resolution of 10 ms (100 EFPS).

By the study conducted, it is shown that the algorithm is able to run in **Real-Time for event rates up to 20 MEPS**, which is a value that is reached only in cases where the texture is very rich of features and the ground is approached with a high speed.

The overall results performed during the 5 flights at the different time resolution, is shown in fig. 5.14.

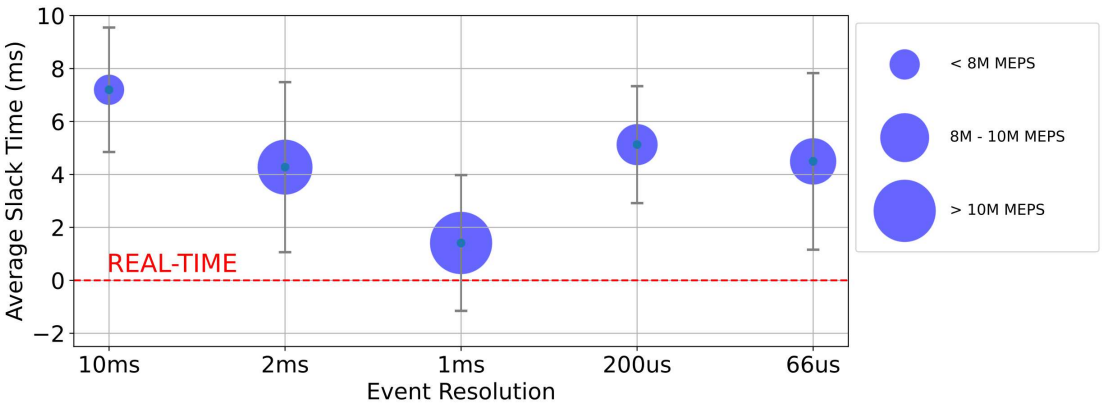

Figure 5.14: Performances for all the *Day Tests*.

The larger the spheres, the higher the number of events. So it is clearly visible that it exists a direct correlation between the MEPS and the Slack Time. The only one that is at the limit of real-time capabilities is the one showed in fig. 5.12, where extremelly high event rates are reached, especially during the landing phase.

Chapter 6

Hardware Setup

This chapter provides a detailed description of the Bixler 3 aircraft used in the experiments. The first section covers the aircraft's mechanical design and the operation of its actuators. Following this, a comprehensive explanation of the onboard electronics and sensors and their interconnections within the system is presented.

6.0.1 The Aircraft

The Bixler 3 is a popular RC model plane, widely used by hobbyists and amateurs. In recent years, it has also gained popularity in academic research due to its simplicity, affordability, and maneuverability.

Similar to many other platforms, the Bixler 3 offers 5 degrees of freedom and is equipped with 5 actuators.

The primary thrust is generated by a single brushless motor, which serves as the main power source. The other four actuators are servo motors responsible for controlling the movement of the control surfaces: two servos operate the right and left ailerons, while the remaining two control the elevator and rudder.

The ailerons enable roll movements, which are essential for turning the plane to the left or right, while the elevator regulates the pitch angle. The rudder primarily assists in roll movements and aids in turning the plane.

Although this model is capable of reaching high speeds of up to 25 m/s, for our experiments, the peak speed was limited to 20 m/s, with a regular setpoint around 15 m/s.

6.0.2 Onboard Electronics

The majority of the electronic components and systems are based on the work of [57], where the objective was to demonstrate high accuracy in flight using a

front-facing frame-based camera.

The authors modified the off-the-shelf plane to integrate additional sensors and computers, enabling it to perform various tasks.

Figure 6.1 shows the block scheme of the setup.

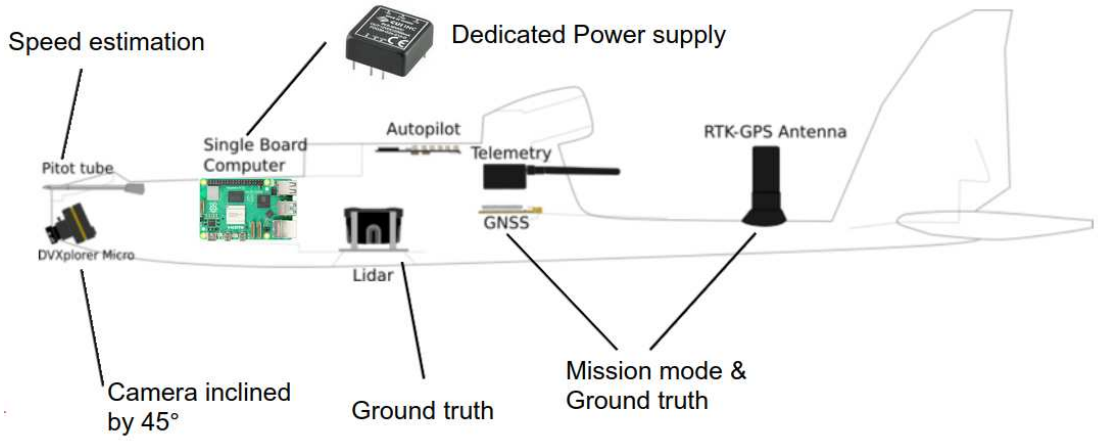

Figure 6.1: Block scheme with main components of the plane used in this work.

The RTK GPS was mounted internally, along with a downward-facing Lidar sensor, positioned near a PX4 Optic Flow sensor to enhance the drone's state estimation.

The airspeed sensor is mounted near the nose of the plane to capture forward wind speed and improve measurement accuracy. This sensor consists of a differential pressure mechanism that estimates wind speed by comparing the incoming airflow with the static one.

For this project, the camera was replaced with the DVXplorer Micro from IniVation, installed at a 45-degree inclination for reasons detailed in section 3.3.

The aircraft was equipped with a PX4 autopilot, widely used in such devices due to its seamless integration, accurate state estimation, and robust technical support.

When not in automatic mode, the plane is controlled via RC commands using a FrSky receiver connected to the autopilot through the SBUS protocol.

To facilitate programming and interaction with the system, a telemetry module was connected to the autopilot. This allowed a ground control station (GCS) software, such as QGroundControl, to wirelessly communicate with the plane in action, displaying live data from the platform.

The initial companion computer used for running the algorithm was the Jetson Nano, a relatively powerful embedded system equipped with a low-power GPU. It communicated with the autopilot via the MavLink protocol to control the actuators.

For this application, the Jetson Nano was replaced with a Raspberry Pi 5 due to its superior performance, enabling the algorithm to run on a CPU without requiring a GPU, making the system more portable. While the Raspberry Pi 5 lacks parallel computation capabilities, its quad-core CPU runs at 2.4 GHz and has double the memory (8 GB) at a lower cost.

The original power supply module, a PM06 v2 from Holybro, could step down LiPo battery voltages (up to 6S) and deliver 3A. This module powered all sensors, the GPS, servo motors, and the autopilot, along with its connected peripherals.

However, the PM06 v2 was insufficient to power the Raspberry Pi stably, causing voltage issues and power shutdowns during flights. To resolve this, a dedicated power supply was introduced, capable of delivering up to 5A, as recommended by the vendor. While the 5A limit was rarely reached, current spikes posed a challenge. The new dedicated supply was a step-down converter, *PDQ30-Q24-S5-D*, capable of stepping down higher voltages (up to 36V) to 5V and delivering the necessary 5A.

A list of components used in this setup is shown in fig. 6.2.

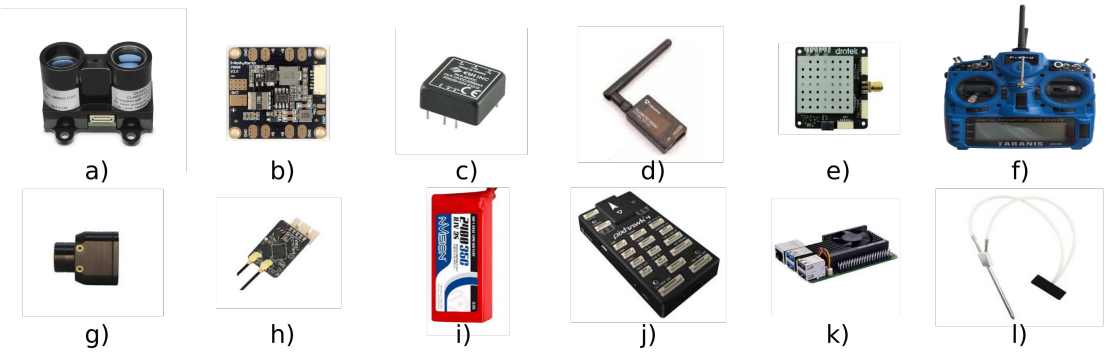

Figure 6.2: Components used in this work. (a) Lidar sensor (*Lidar Lite v3*), (b) Power Module (*PM06 v2*), (c) Step-down converter (*PDQ30-Q24-S5-D*), (d) Telemetry module (*Holybro 915 MHz*), (e) GPS rover module (*Drotek DP0601*), (f) RC remote controller (*FrSky Taranis X9D*), (g) Event-based camera (*DVXplorer Micro*), (h) RC receiver (*FrSky X8R*), (i) LiPO battery 3S 2400 mAh, (j) Autopilot (*PX4*), (k) Companion computer (*Raspberry Pi 5*), (l) Airspeed sensor (*Sensirion SDP3x*)

6.0.3 Software

The software stack comprises the following components:

- **PX4 Autopilot**: This NuttX-based software is responsible for state estimation, control, and communication with peripherals. Key parameters include:

  - *EKF2_HGT_REF* set to 3 (vision): This sets the height estimation to vision mode, using the camera as the altitude estimation source from the companion computer.
  - *EKF2_EV_CTRL* set to 2: This allows the vision system to receive state estimation data from the companion computer, provided pose messages are sent at 50 Hz.

- **Raspberry Pi 5**: The companion computer running the algorithm and sending estimations to the autopilot. It operates on Ubuntu 23.04, with ROS 1 Noetic running inside a Docker container for compatibility. The algorithm, written in C++, communicates with the autopilot through the MavROS package.

Only two ROS nodes are used: the *mavros* node for communication with the autopilot, and the *event_based_control* node, which processes events and estimates altitude. Unlike other implementations that use separate nodes for handling events and computations, this streamlined approach avoids overhead, especially with large data sets such as event streams, optimizing performance.

6.0.4 Compiler Optimization

For onboard processing, optimizing software compilation was crucial to ensure efficient performance on the limited resources of the Raspberry Pi 5. The compilation process was managed through CMake, and several key parameters were configured in the ‘CMakeLists.txt‘ file to maximize performance.

The use of C++11 ensures modern language features while maintaining efficiency. The primary library dependency, *DV-processing*, provides functions to retrieve event data from the camera.

Several optimization commands were applied to the compilation process, including:

- **-O3 and -Ofast**: These flags enable aggressive optimization techniques, with `-O3` focusing on advanced loop unrolling and inlining, while `-Ofast` disregards strict compliance with standards for even faster execution, potentially affecting floating-point precision.
- **-march=native and -mtune=native**: These flags optimize the code specifically for the ARM architecture of the Raspberry Pi.
- **-funroll-loops**: Reduces loop control overhead by precomputing iterations, improving CPU pipeline usage.
- **-fomit-frame-pointer**: Omits the frame pointer, freeing up a register and improving performance.

- **-flto**: Link-Time Optimization (LTO) enhances performance across multiple files by reducing binary size and allowing optimizations during linking.
- **-ftree-vectorize**: Enables automatic vectorization to perform multiple operations in parallel, utilizing the ARM architecture's NEON SIMD instructions.
- **-fgraphite-identity and -floop-nest-optimize**: These advanced loop optimization flags use the Graphite framework to improve nested loop performance, commonly found in image processing algorithms.
- **-frename-registers**: Reduces register contention, further improving the algorithm's speed.

These optimizations significantly improved performance, allowing the vision-based algorithm to run in real-time on the Raspberry Pi 5, despite its limited resources.

Additional flags that could have been utilized for further optimization included:

- *-funsafe-math-optimizations*: Enables more aggressive mathematical optimizations at the potential cost of numerical accuracy.
- *-fassociative-math*: Allows reordering of floating-point operations based on the associative property to enhance performance.
- *-freciprocal-math*: Allows the use of faster reciprocal approximations, improving performance for division operations.
- *-fno-trapping-math*: Disables floating-point operation traps, which can enhance performance where trapping is not necessary.

However, these flags were not employed due to errors encountered during some mathematical operations. They introduced instability and precision issues, negatively impacting the reliability of the algorithm, particularly when processing sensor data.

Chapter 7

Conclusion

This thesis explored the use of NC in high-speed scenarios, particularly in FW UAVs. It is the first time event-based vision has been applied in this specific area of robotics, and the focus was on developing a real-time pipeline for altitude estimation using OF algorithms, necessary for estimating depth from monocular vision.

Given the novelty of this field, we started with a detailed review of current OF techniques designed for the sparse, event-based data produced by NCs. The techniques fall into two categories: model-based and learning-based, each with its own trade-offs between accuracy, processing speed, and adaptability for real-time use. After careful analysis, we implemented an optimized version of the sparse LK algorithm adapted to event based data, as it provided a good balance between performance and computational efficiency, particularly for the high event rates seen in fast-flight scenarios, while still being able to run on embedded platforms.

Additional techniques were incorporated to boost pipeline performance, such as OF derotation, adaptive event stream slicing, and the altitude estimation algorithm. The derotation process was crucial for compensating rotational motion, achieving up to 99% rejection of rotational components. The adaptive slicing strategy adjusted the event window dynamically based on the magnitude of the OF vectors, improving flow accuracy, and has been implemented through a PI controller varying on the OF vectors' magnitude.

The altitude estimation algorithm was thoroughly tested in outdoor flights, where we analyzed how event-time resolution impacted accuracy. As expected, higher resolutions (up to 66 µs) led to better accuracy, though even lower resolutions yielded acceptable results with an average relative error of 17%. Tests in low-light conditions (around 50 lux) showed that with medium to very high sensitivity, reliable altitude estimations could still be achieved, maintaining an error of about 20%. The ground truth for the altitude was provided by an RTK GPS mounted on the drone, which offers higher accuracies than normal standard GPS.

One of the main challenges was balancing the limited processing power of the *Raspberry Pi 5* with the high event rates caused by fast motion (15 m/s), which sometimes reached 40 MEPS, especially during landings. However, the algorithm successfully ran in real-time under conditions generating up to 20 MEPS, which is higher than typically encountered in most applications under normal lighting.

This solution is flexible in terms of OF vector sparsity, allowing adjustments to the number of flow vectors based on the application. Other research suggests that such high data rates are uncommon in robotics applications with slower velocities, making it easy to adapt this solution for different use cases with minimal modifications.

7.0.1 Future improvements

Although this research has demonstrated promising results, several areas could be further improved to enhance the system's capabilities and performance. Below are some suggestions for future work:

- **Implementation of a custom OF algorithm based on machine learning:** Currently, the sparse LK algorithm was optimized for event-based data due to its balance between accuracy and computational efficiency on resource-constrained platforms. The current state-of-the-art algorithms for OF computation were not fast enough to cope up with really high event rates, so due to limited time a pre-made algorithm has been chosen. Although this the FAST+LK method developed in this work is somehow event-driven, it does not directly exploit the sparsity of the generated events, but rather it works in a conventional way by collecting Edge Images. With the increasing availability of embedded GPUs, particularly in platforms like the NVIDIA Jetson, a custom OF algorithm based on machine learning could be developed. By leveraging an embedded GPU, the system could take advantage of the parallel processing power to handle high event rates and complex motion patterns, offering improved robustness and accuracy in challenging scenarios.

- **Using a forward-facing camera with a wide field of view:** In this work, the NC was mounted at a 45-degree downward tilt to estimate altitude and maintain ground visibility. A potential improvement would be to use a forward-facing camera with a wide field of view (FOV). This configuration would allow the drone to capture a larger portion of its environment, not only for altitude estimation but also for navigation and monitoring tasks. A wide-angle lens would enable more flexible applications, such as obstacle detection and path planning, without losing the ability to estimate altitude during landing.

- **Testing at higher speeds:** While this study focused on flight scenarios with speeds of up to 20 m/s, testing the system at even higher speeds would be beneficial, especially in applications like high-speed racing drones or emergency response missions. At higher speeds, the event rate will increase substantially, which might pose new challenges for the OF algorithm and real-time processing pipeline. Exploring higher-speed tests could help identify potential bottlenecks in the system and push the limits of its current performance capabilities.
- **Obstacle avoidance:** Another promising direction for future development is incorporating obstacle avoidance capabilities into the current system. NCs, with their high temporal resolution and event-driven nature, are particularly suited for real-time obstacle detection and avoidance. By integrating an obstacle avoidance module into the existing pipeline, the system could become more versatile and autonomous, allowing the drone to navigate complex environments safely. Implementing a real-time obstacle avoidance system, particularly at high speeds, would make the overall system more applicable for a variety of UAV missions, including search and rescue, environmental monitoring, and autonomous delivery.
- **Dedicated control system:** The current altitude estimation algorithm relies on the PX4 autopilot that is in charge of the low level actuation controls directed to each motor. This works really well but it has been chosen because the goal of the work was to correctly estimate the altitude and not to develop a control system. In the future, a more complex control system could be developed to fully control the aircraft, estimate altitude and perform autonomous navigation paired with obstacle avoidance, making it a fully autonomous system.

Appendix A

DVS pixel circuitry

Each pixel in a neuromorphic camera generates an event by responding to changes in brightness. The pixel contains a photodiode, which detects light and produces a photocurrent. This photocurrent is then processed by a transistor-based circuit.

The key component in this process is a feedback transistor that converts the photocurrent into a voltage. This voltage is related to the brightness of the light, and the circuit holds the photodiode at a constant reference level (called a virtual ground). This setup allows the pixel to respond quickly, which is particularly useful in low light or high-speed scenarios.

Additionally, the circuit can adjust its power consumption based on the overall brightness in the scene. It does this by using the combined photocurrents from all the pixels to change the bias of one of the transistors. This adaptive feature ensures that the pixel circuit remains efficient and stable even when lighting conditions change.

When a change in brightness occurs, the voltage at the output of the pixel circuit changes. This change is then compared to a preset threshold by a comparator. If the voltage change is large enough, the pixel generates an "ON" event (for brightness increase) or an "OFF" event (for brightness decrease). These events are then sent to the rest of the system for further processing.

Below is the schematic of the pixel's circuit, which visually represents this process:

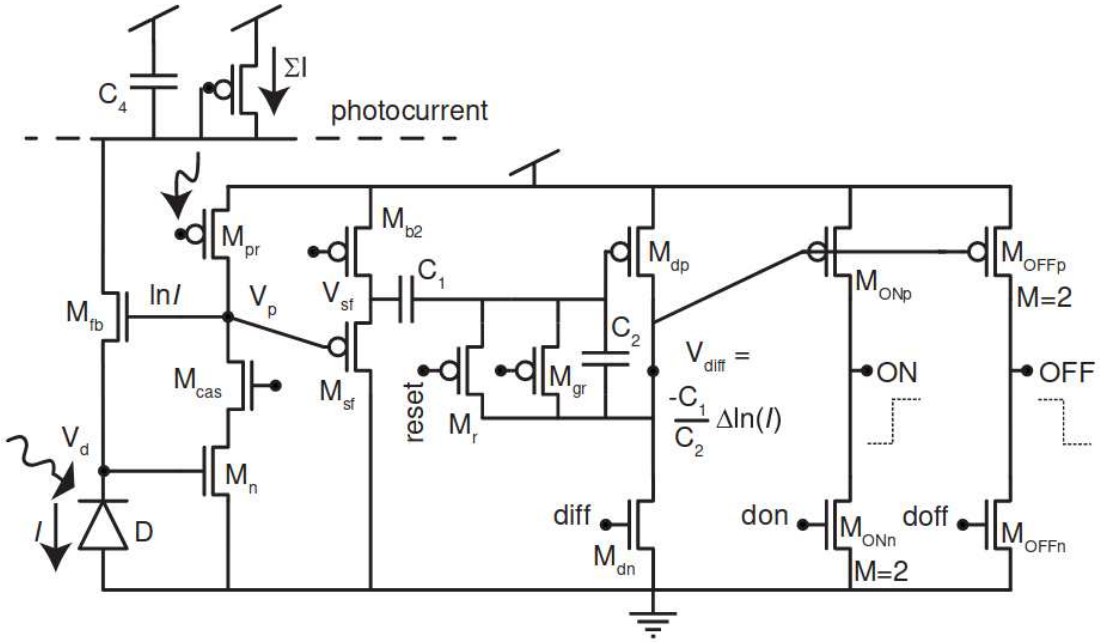

Figure A.1: Detail representation of the pixel's circuitry

Appendix B

Optical Flow for Event-based cameras : an Evaluation

As explained in chapter 1, event-based cameras have several advantages over traditional frame-based cameras, especially in the context of high-speed and dynamic environments.

In fact, only sparse and non-redundant data is generated, while on the other hand frame-based cameras produce a large amount of data.

One of the main phase of this project was to evaluate the performance of OF algorithms on event-based cameras in order to choose the best one that would suit the needs of the project.

As introduced by sections 2.4.1 and 2.4.2, two main categories of OF algorithms exist: model-based and learning-based.

In this appendix, we will present our version of the evaluation in terms of speed and computational load of the main OF methods that exist in the literature. To better group them, the overall evaluation will be divided based on the two categories.

For reference, we have chosen to define as *Real Time* an algorithm that is able to process the data at a speed of at least 30 FPS (for the equivalent frame rate), of a event rate of 5 Million Events per Second (MEPS) for the model-based algorithms.

Not all of the methods have been tested, so the majority of the data is taken from the original papers, or others that have tested the algorithms.

B.1 Learning-based Optical Flow Algorithms

The key metrics to evaluate the current state of the art learning-based OF algorithms are:

- **Equivalent Frame Rate FPS** : The equivalent frame rate that the algorithms is able to process. Although the event-based cameras do not usually have a frame rate, this metric is a good way of representing the speed of the algorithm, usually computed by the inverse of the inference time of the network.
- **Required Power Consumption** : The power consumption of the utilized GPU (for learning based models) to process the algorithm, expressed in Watts. For this metric, other metrics could have been used, such as the TFLOPs, or number of cores, but we have chosen the power consumption as it is a more direct metric to scale the computational load of the methods on different hardware, roughly.

A graph showing the evaluation of these methods is shown in fig. B.1.

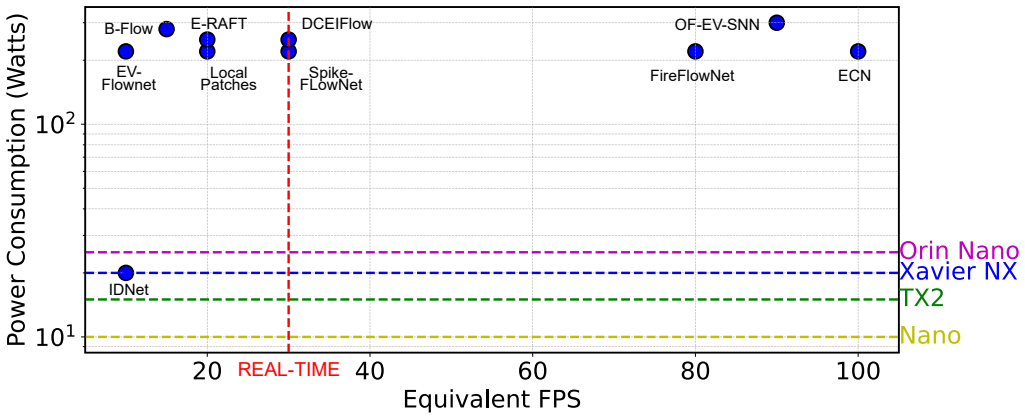

Figure B.1: Evaluation of learning-based OF algorithms. B-Flow [58], E-RAFT [59], DCEIFlow [60], OF-EV-SNN [61], EV-FlowNet [21], Local Patches [62], Spike-FlowNet [63], FireFlowNet [11], ECN [64], IDNet [20]

The mentioned algorithms are referenced as follows:

Since in our project we are interested in robotics applications, embedded platforms are usually needed, and the power consumption is a key metric to evaluate the computational load of the algorithms. For reference, on the graph we have added the power consumption of the most common embedded GPUs, being part of the NVIDIA Jetson family.

It is important to note that the ones showed in the graph are the most common and used algorithms in the literature, and there are many other algorithms that could have been evaluated, but for the sake of simplicity and clarity, we have chosen to evaluate the most common ones.

The majority of the algorithms, except for the IDNet [20], are trained and evaluated on desktop GPUs, and the power consumption is not directly comparable

to embedded GPUs, but it is a good way to have a rough idea of the computational load of the algorithms.

One key missing data that papers do not usually mentions, is the computational time needed to *preprocess* the data before feeding it to the network, in various methods called *Event Representation*, explained in detail in section 1.3. In fact, it would be more fair to evaluate the algorithms based on the overall computational load, including the preprocessing time, but this data is not usually provided in the papers. So, the resulting evaluation is only taking into account the inference time of the network, however sometimes, as shown in section 5.4.1, the preprocessing time can be a significant part of the overall computational load. Hence, in reality, those performance would result in lower FPS with respect to the ones showed in the graph.

A takeaway from the graph is that the majority of these algorithms are not able to run in real time on embedded platforms. As a demonstration, apart from *FireFlowNet* [11], none of them has been used in high speed robotics applications, as other papers mentioned that the actual state of the art OF methods are too computationally expensive to be used [38].

The case of *FireFlowNet* is however to be analyzed carefully, as the only actual application of the algorithm, used for quadrotor landing [42], was run on a *Neuromorphic Processor*, which is a very particular type of platform that is capable of reaching really low inference times. Furthermore, the application was reported to generate really low event rates (less than 300 kEPS), and only 4 blocks of 16x16 pixels within the camera frame were used, making the overall computational load really low.

As stated by [20], "Among all published methods, TID (IDNet) stands out as the only one nearing a real-time processing rate on the DSEC-Flow dataset (10Hz) while still upholding decent accuracy," confirming the unfeasibility of most algorithms to run in real-time on embedded platforms.

B.2 Model-based Optical Flow Algorithms

The key metrics to evaluate the current state of the art model-based OF algorithms are:

- **Maximum Event Rate (MEPS)** : The maximum event rate that the algorithms is able to process. This metric is a good way of representing the speed of the algorithm, as the event-based cameras do not usually have a frame rate.

- **CPU Frequency (GHz)** : The frequency of the utilized CPU to process the algorithm, expressed in GHz. This metric is a good way to evaluate the

computational load of the algorithms, however many factors would affect the performance of the algorithm, such as the number of cores, the cache size, the architecture, etc.

A graph showing the evaluation of these methods is shown in fig. B.2.

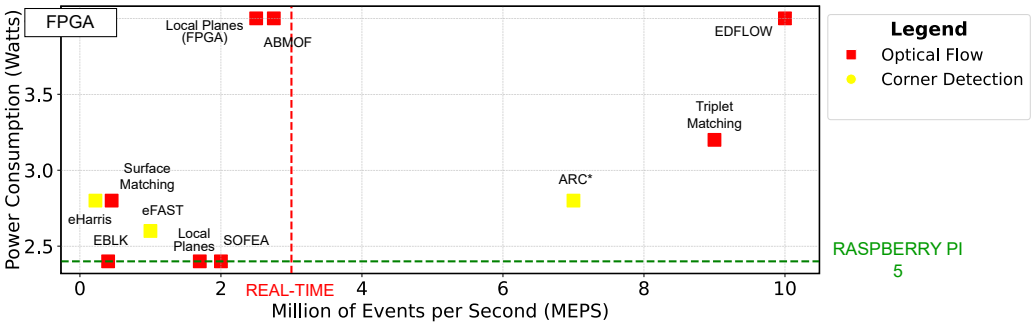

Figure B.2: Evaluation of model-based OF algorithms. eHarris [29], eFAST [7], ARC* [29], EBLK [19], Local Planes [8], SOFEA [10], Surface Matching [65], Triplet Matching [66], Local Planes (FPGA) [9], ABMOF [5], EDFLOW [6]

In this case, using the power consumption as a metric would not be too useful, since all of the algorithms are implemented on CPUs, and the power consumption is not the most relevant metric to evaluate the computational load of the algorithms. Instead, the CPU frequency is a good way to evaluate the performance, since the majority of them has been executed on a single CPU core (not all of them), and nowadays the core frequency is directly proportional to the number of operations that the CPU can perform per second.

As in the previous case, the CPU frequency of the embedded platforms has been added, corresponding in this case to the one of the Raspberry Pi 5, which is the one used in this work.

In *Model Based* algorithms, it is also relevant to mention the **Feature Detection** algorithms, which have been added on the graph with their respective color coding (see the legend). In fact, computing sparse OF only on detected features sometimes is more efficient than computing the OF on the whole image, especially in high-speed scenarios.

Since some of the OF methods used an FPGA to process the data, the power consumption is not directly comparable to CPUs, hence it has been associated to a CPU frequency to 4 GHz, to visually separate and distinguish from the other methods. In fact, FPGAs are usually more efficient and performing when it comes to ad-hoc algorithm implementations, and in these cases they could be considered as a high power consumption requirement.

As it can be seen from the graph, the majority of the model-based algorithms are quite slow in processing the data, and only a few of them are able to keep up with the event rate of the event-based cameras, especially at high-speed scenarios. The FPGA implementations were excluded for the complexity they would add to the system (because it needs to be mounted on a small plane).

The *Triplet Matching* algorithm [66] is theoretically the only one that would be able to run in real time, but the code proposed in the paper is not available, the hardware requirement is quite high since it was run on a *MAC M1 Pro* CPU, and no other references the actual robustness and validity of the algorithm, making it not a good choice for the project.

Regarding the *ARC** feature detector [29], it has been actually tested on the Raspberry Pi 5, performing decently on medium-low event rates, but not being able to keep up with the high-speed scenarios such as the one of the project. Furthermore, it is only a corner detector algorithm, and also an OF algorithm needs to be added to the pipeline, making the overall computational load even higher.

Generally, these methods perform even worse than the learning-based ones, usually because they have been developed in the early stages of the event-based cameras and the knowledge of the field was not as advanced as it is now.

Researchers in [67] have dedicated a lot of effort in order to evaluate the performance of *Model Based* OF algorithms, assessing both accuracy and computational load of the majority of the algorithms in the literature, and the results are taken from their work.

As they also report, "they all run in real-time on contemporary PCs if the event-rate does not exceed 100 kEPS," but for *contemporary PCs* they mean ones that burn hundreds of watts of power, which is not the case for embedded platforms.

The key limitations for those is the nature of the algorithms itself. In fact they process event-by-event, whatever is their approach. This would result in good timing performance for low event rates, but really bad for high-speed scenarios, as the computational load would increase linearly with the event rate.

The evaluation of the OF algorithms has shown that the majority of the algorithms are not able to run in real time on embedded platforms, especially in high-speed scenarios.

To overcome these limitations, two possible solutions can be considered:

- **Adapted Frame-based OF Algorithm for Event-based Cameras:** This approach adapts conventional OF algorithms to work with event-based cameras. Two feasible solutions have been identified:
  - **Sparse FAST[27]+LK[23]** algorithm: A conventional sparse OF algorithm using FAST for feature detection and LK for OF computation, adapted to event streams.

  - **Dense Inverse Search (DIS)** algorithm [68]: A dense OF algorithm, also adapted for event-based cameras.

- **Hybrid Model:** This approach combines *model-based methods* for feature detection with *conventional OF estimators* for the OF computation. The **Sparse ARC[29]+LK** algorithm was originally considered, where ARC serves as the feature detector combined with LK for OF computation.

Appendix C

Result Data

C.0.1 Altitude estimation accuracy during the Day test

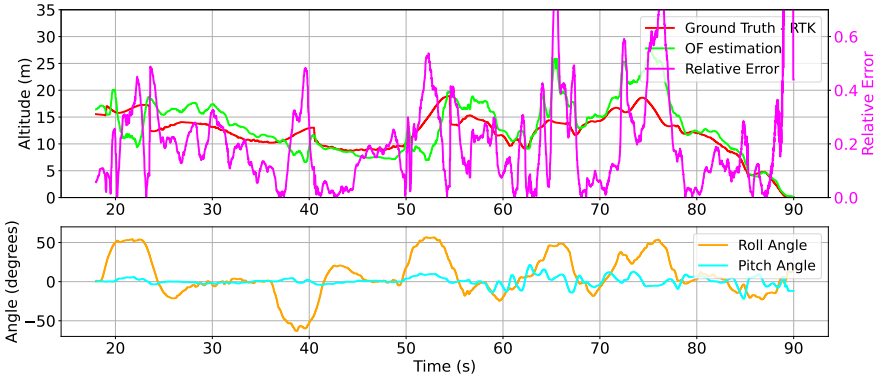

Figure C.1: EFPS = 100

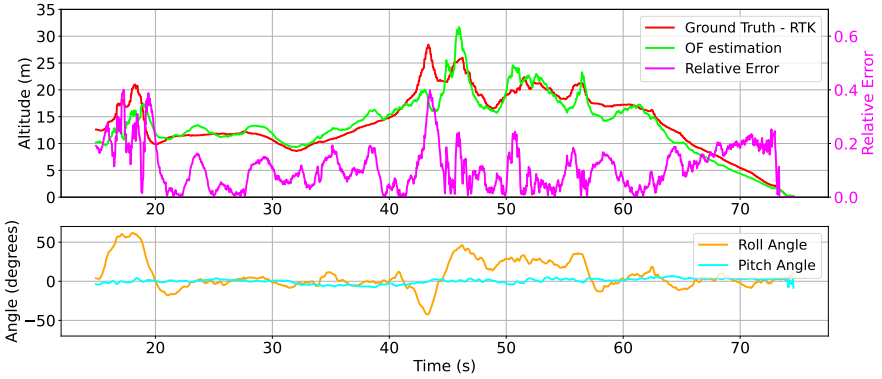

Figure C.2: EFPS = 500

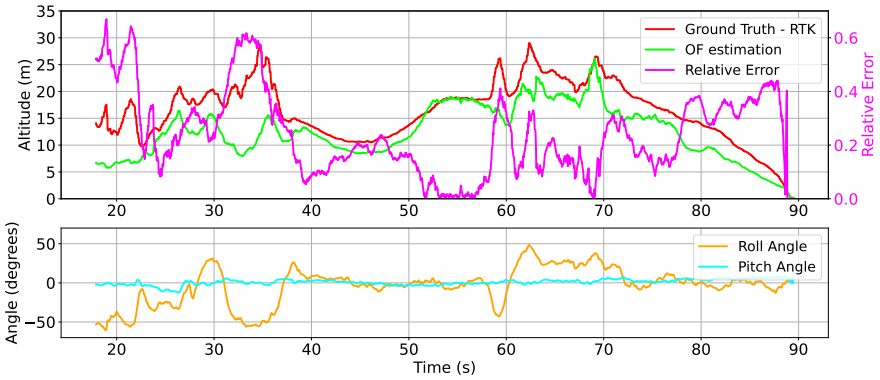

Figure C.3: EFPS = 1000

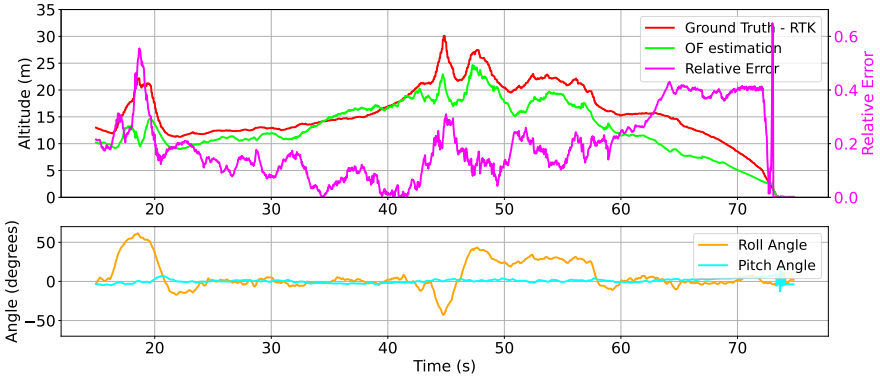

Figure C.4: EFPS = 5000

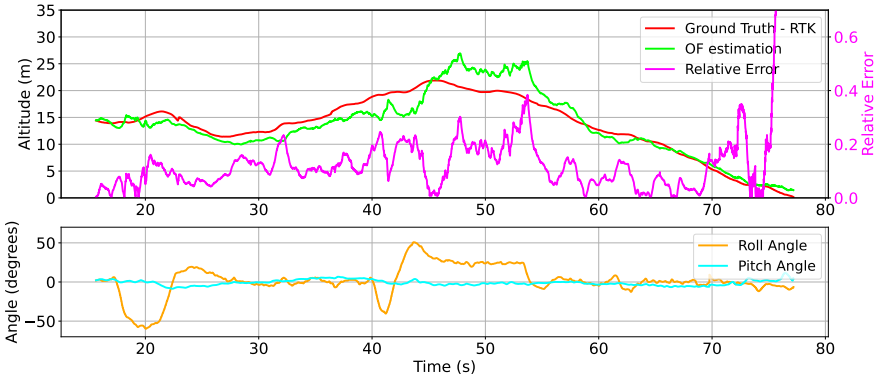

Figure C.5: EFPS = 15000

C.0.2 Computational performance during the Day test

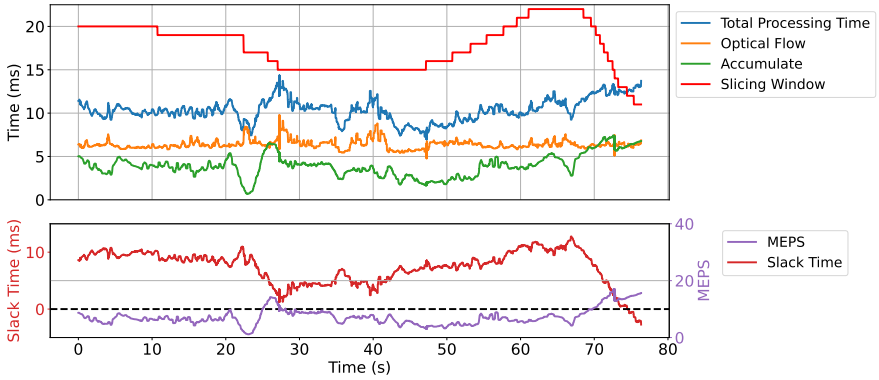

Figure C.6: EFPS = 100

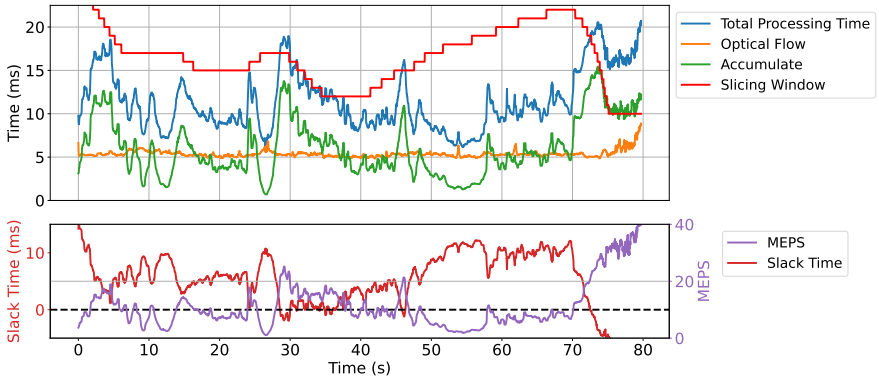

Figure C.7: EFPS = 500

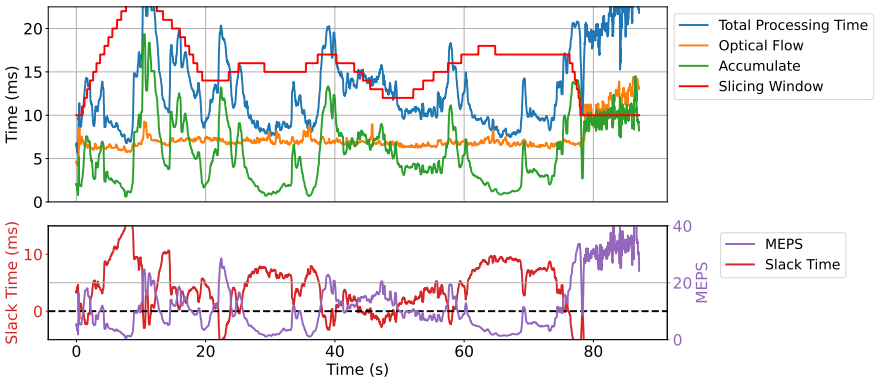

Figure C.8: EFPS = 1000

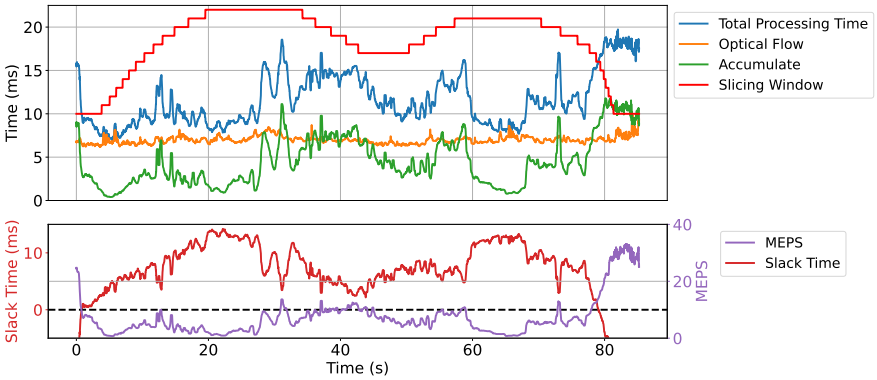

Figure C.9: EFPS = 5000

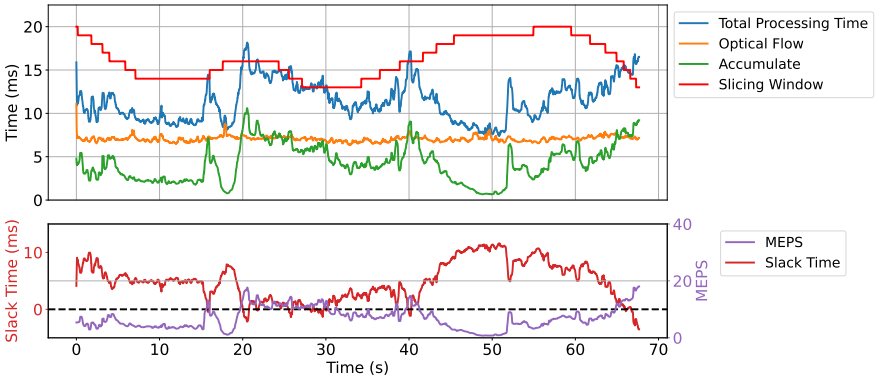

Figure C.10: EFPS = 15000

Bibliography

[1] M. A. Mahowald and C. Mead. «The silicon retina». In: *Scientific American* 264.5 (1991), pp. 76–82. DOI: 10.1038/scientificamerican0591-76 (cit. on p. 1).

[2] T. Delbruck and C.A. Mead. «Adaptive photoreceptor with wide dynamic range». In: *Proceedings of IEEE International Symposium on Circuits and Systems - ISCAS '94*. Vol. 4. 1994, 339–342 vol.4. DOI: 10.1109/ISCAS.1994.409266 (cit. on p. 1).

[3] Patrick Lichtsteiner, Christoph Posch, and Tobi Delbruck. «A 128×128 120 dB 15 µ s Latency Asynchronous Temporal Contrast Vision Sensor». In: *Solid-State Circuits, IEEE Journal of* 43 (Mar. 2008), pp. 566–576. DOI: 10.1109/JSSC.2007.914337 (cit. on pp. 1–4).

[4] Mathias Gehrig, Willem Aarents, Daniel Gehrig, and Davide Scaramuzza. *DSEC: A Stereo Event Camera Dataset for Driving Scenarios*. 2021 (cit. on pp. 6, 23).

[5] Min Liu and T Delbruck. «Adaptive Time-Slice Block-Matching Optical Flow Algorithm for Dynamic Vision Sensors». In: *British Machine Vision Conference (BMVC) 2018*. Newcastle upon Tyne, UK: BMVC, Sept. 2018 (cit. on pp. 7, 29, 64, 95).

[6] Min Liu and Tobi Delbruck. «Event Driven Optical Flow Camera With Keypoint Detection and Adaptive Block Matching». In: *IEEE Transactions on Circuits and Systems for Video Technology* 32.9 (Sept. 2022). Conference Name: IEEE Transactions on Circuits and Systems for Video Technology, pp. 5776–5789. ISSN: 1558-2205. DOI: 10.1109/TCSVT.2022.3156653 (cit. on pp. 7, 95).

[7] Elias Mueggler, Chiara Bartolozzi, and Davide Scaramuzza. «Fast Event-based Corner Detection». In: Sept. 2017. DOI: 10.5244/C.31.33 (cit. on pp. 7, 20, 95).

[8] Ryad Benosman, Charles Clercq, Xavier Lagorce, Sio-Hoi Ieng, and Chiara Bartolozzi. «Event-Based Visual Flow». en. In: *IEEE Transactions on Neural Networks and Learning Systems* 25.2 (Feb. 2014), pp. 407–417. ISSN: 2162-237X, 2162-2388. DOI: 10.1109/TNNLS.2013.2273537 (cit. on pp. 7, 16, 95).

[9] Myo Tun Aung and Rodney Teo. «Event-based Plane-fitting Optical Flow for Dynamic Vision Sensors in FPGA». In: *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*. ISSN: 2379-447X. May 2018, pp. 1–5. DOI: 10.1109/ISCAS.2018.8351588 (cit. on pp. 7, 95).

[10] Weng Fei Low, Zhi Gao, Cheng Xiang, and Bharath Ramesh. «SOFEA : A Non-iterative and Robust Optical Flow Estimation Algorithm for Dynamic Vision Sensors». en. In: *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*. Seattle, WA, USA: IEEE, June 2020, pp. 368–377. ISBN: 978-1-72819-360-1. DOI: 10.1109/CVPRW50498.2020.00049 (cit. on pp. 7, 16, 95).

[11] Federico Paredes-Valles and Guido C. H. E. De Croon. «Back to Event Basics: Self-Supervised Learning of Image Reconstruction for Event Cameras via Photometric Constancy». en. In: *2021 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. Nashville, TN, USA: IEEE, June 2021, pp. 3445–3454. ISBN: 978-1-66544-509-2. DOI: 10.1109/CVPR46437.2021.00345 (cit. on pp. 7, 93, 94).

[12] Simon Baker, Daniel Scharstein, J. P. Lewis, Stefan Roth, Michael J. Black, and Richard Szeliski. «A Database and Evaluation Methodology for Optical Flow». en. In: *International Journal of Computer Vision* 92.1 (Mar. 2011), pp. 1–31. ISSN: 0920-5691, 1573-1405. DOI: 10.1007/s11263-010-0390-2 (cit. on p. 11).

[13] B. K. P. Horn and B. G. Schunck. «Determining Optical Flow». In: *Artificial Intelligence* 17 (1981). Manuscript available on MIT server, pp. 185–203 (cit. on pp. 11, 18).

[14] J. J. Koenderink and Andrea J. van Doorn. «Facts on optic flow». en. In: *Biological Cybernetics* 56.4 (June 1987), pp. 247–254. ISSN: 1432-0770. DOI: 10.1007/BF00365219 (cit. on p. 11).

[15] Graham R. Martin. «What Drives Bird Vision? Bill Control and Predator Detection Overshadow Flight». In: *Frontiers in Neuroscience* 11 (2017). ISSN: 1662-453X (cit. on p. 13).

[16] Ingo Schiffner and Mandyam V. Srinivasan. «Direct Evidence for Vision-based Control of Flight Speed in Budgerigars». en. In: *Scientific Reports* 5.1 (June 2015). Number: 1 Publisher: Nature Publishing Group, p. 10992. ISSN: 2045-2322. DOI: 10.1038/srep10992 (cit. on p. 13).

[17] Sofía Miñano, Stuart Golodetz, Tommaso Cavallari, and Graham K. Taylor. «Through Hawks' Eyes: Synthetically Reconstructing the Visual Field of a Bird in Flight». en. In: *International Journal of Computer Vision* 131.6 (June 2023). Number: 6, pp. 1497–1531. ISSN: 1573-1405. DOI: 10.1007/s11263-022-01733-2 (cit. on p. 13).

[18] Julien R. Serres, Thomas J. Evans, Susanne Åkesson, Olivier Duriez, Judy Shamoun-Baranes, Franck Ruffier, and Anders Hedenström. «Optic flow cues help explain altitude control over sea in freely flying gulls». In: *Journal of The Royal Society Interface* 16.159 (Oct. 2019). Publisher: Royal Society, p. 20190486. DOI: 10.1098/rsif.2019.0486 (cit. on pp. 13, 14).

[19] Ryad Benosman, Sio-Hoi Ieng, Charles Clercq, Chiara Bartolozzi, and Mandyam Srinivasan. «Asynchronous frameless event-based optical flow». In: *Neural Networks* 27 (2012), pp. 32–37. ISSN: 0893-6080. DOI: https://doi.org/10.1016/j.neunet.2011.11.001 (cit. on pp. 16, 95).

[20] Yilun Wu, Federico Paredes-Vallés, and Guido C. H. E. de Croon. *Lightweight Event-based Optical Flow Estimation via Iterative Deblurring*. arXiv:2211.13726 [cs]. May 2024. DOI: 10.48550/arXiv.2211.13726 (cit. on pp. 16, 17, 93, 94).

[21] Alex Zhu, Liangzhe Yuan, Kenneth Chaney, and Kostas Daniilidis. «EV-FlowNet: Self-Supervised Optical Flow Estimation for Event-based Cameras». en. In: *Robotics: Science and Systems XIV*. Robotics: Science and Systems Foundation, June 2018. ISBN: 978-0-9923747-4-7. DOI: 10.15607/RSS.2018.XIV.062 (cit. on pp. 16, 17, 93).

[22] Yannick Schnider, Stanisław Woźniak, Mathias Gehrig, Jules Lecomte, Axel Von Arnim, Luca Benini, Davide Scaramuzza, and Angeliki Pantazi. «Neuromorphic Optical Flow and Real-time Implementation with Event Cameras». en. In: *2023 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*. Vancouver, BC, Canada: IEEE, June 2023, pp. 4129–4138. DOI: 10.1109/CVPRW59228.2023.00434 (cit. on p. 17).

[23] Bruce Lucas and Takeo Kanade. «An Iterative Image Registration Technique with an Application to Stereo Vision (IJCAI)». In: vol. 81. Apr. 1981 (cit. on pp. 18, 96).

[24] Gunnar Farnebäck. «Two-Frame Motion Estimation Based on Polynomial Expansion». en. In: *Image Analysis*. Ed. by Josef Bigun and Tomas Gustavsson. Berlin, Heidelberg: Springer, 2003, pp. 363–370. ISBN: 978-3-540-45103-7. DOI: 10.1007/3-540-45103-X_50 (cit. on p. 18).

[25] C. Harris and M. Stephens. «A Combined Corner and Edge Detector». en. In: *Procedings of the Alvey Vision Conference 1988*. Manchester: Alvey Vision Club, 1988, pp. 23.1–23.6. DOI: 10.5244/C.2.23 (cit. on p. 19).

[26] Jianbo Shi and Tomasi. «Good features to track». en. In: *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition CVPR-94*. Seattle, WA, USA: IEEE Comput. Soc. Press, 1994, pp. 593–600. ISBN: 978-0-8186-5825-9. DOI: 10.1109/CVPR.1994.323794 (cit. on p. 19).

[27] Edward Rosten and Tom Drummond. «Machine Learning for High-Speed Corner Detection». en. In: *Computer Vision – ECCV 2006*. Ed. by Aleš Leonardis, Horst Bischof, and Axel Pinz. Vol. 3951. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 430–443. DOI: 10.1007/11744023_34 (cit. on pp. 19, 39, 96).

[28] Valentina Vasco, Arren Glover, and Chiara Bartolozzi. «Fast event-based Harris corner detection exploiting the advantages of event-driven cameras». en. In: *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. Daejeon, South Korea: IEEE, Oct. 2016, pp. 4144–4149. ISBN: 978-1-5090-3762-9. DOI: 10.1109/IROS.2016.7759610 (cit. on p. 20).

[29] Ignacio Alzugaray and Margarita Chli. «Asynchronous Corner Detection and Tracking for Event Cameras in Real-Time». In: *IEEE Robotics and Automation Letters* PP (June 2018), pp. 1–1. DOI: 10.1109/LRA.2018.2849882 (cit. on pp. 20, 95–97).

[30] Sungsik Huh and David Hyunchul Shim. «A Vision-Based Automatic Landing Method for Fixed-Wing UAVs». en. In: *Journal of Intelligent and Robotic Systems* 57.1 (Jan. 2010), pp. 217–231. ISSN: 1573-0409. DOI: 10.1007/s10846-009-9382-2 (cit. on p. 22).

[31] Martin Trittler, Thomas Rothermel, and Walter Fichter. «Autopilot for Landing Small Fixed-Wing Unmanned Aerial Vehicles with Optical Sensors». In: *Journal of Guidance, Control, and Dynamics* 39.9 (2016), pp. 2011–2021. ISSN: 0731-5090. DOI: 10.2514/1.G000261 (cit. on p. 22).

[32] Zhaoyang Wang, Dan Zhao, and Yunfeng Cao. «Visual Navigation Algorithm for Night Landing of Fixed-Wing Unmanned Aerial Vehicle». In: *Aerospace* 9.10 (2022). ISSN: 2226-4310. DOI: 10.3390/aerospace9100615 (cit. on p. 22).

[33] YanMing Fan, Meng Ding, and YunFeng Cao. «Vision algorithms for fixed-wing unmanned aerial vehicle landing system». en. In: *Science China Technological Sciences* 60.3 (Mar. 2017), pp. 434–443. ISSN: 1869-1900. DOI: 10.1007/s11431-016-0618-3 (cit. on p. 22).

[34] Xavier Clady, Charles Clercq, Sio-Hoi Ieng, Fouzhan Houseini, Marco Randazzo, Lorenzo Natale, Chiara Bartolozzi, and Ryad Benosman. «Asynchronous visual event-based time-to-contact». In: *Frontiers in Neuroscience* 8 (Feb. 2014), p. 9. ISSN: 1662-4548. DOI: 10.3389/fnins.2014.00009 (cit. on p. 22).

[35] Moritz B. Milde, Olivier J.N. Bertrand, Ryad Benosmanz, Martin Egelhaaf, and Elisabetta Chicca. «Bioinspired event-driven collision avoidance algorithm based on optic flow». In: *2015 International Conference on Event-based Control, Communication, and Signal Processing (EBCCSP)*. June 2015, pp. 1–7. DOI: 10.1109/EBCCSP.2015.7300673 (cit. on p. 22).

[36] Benedek Forrai, Takahiro Miki, Daniel Gehrig, Marco Hutter, and Davide Scaramuzza. *Event-based Agile Object Catching with a Quadrupedal Robot*. arXiv:2303.17479 [cs]. Apr. 2023. DOI: 10.48550/arXiv.2303.17479 (cit. on p. 22).

[37] Nitin J. Sanket, Chethan M. Parameshwara, Chahat Deep Singh, Ashwin V. Kuruttukulam, Cornelia Fermüller, Davide Scaramuzza, and Yiannis Aloimonos. *EVDodgeNet: Deep Dynamic Obstacle Dodging with Event Cameras*. 2020 (cit. on pp. 22, 23).

[38] Davide Falanga, Kevin Kleber, and Davide Scaramuzza. «Dynamic obstacle avoidance for quadrotors with event cameras». In: *Science Robotics* 5.40 (2020), eaaz9712. DOI: 10.1126/scirobotics.aaz9712 (cit. on pp. 22, 32, 94).

[39] Amogh Joshi, Sourav Sanyal, and Kaushik Roy. *Real-Time Neuromorphic Navigation: Integrating Event-Based Vision and Physics-Driven Planning on a Parrot Bebop2 Quadrotor*. 2024 (cit. on p. 22).

[40] Elias Mueggler, Nathan Baumli, Flavio Fontana, and Davide Scaramuzza. «Towards evasive maneuvers with quadrotors using dynamic vision sensors». In: *2015 European Conference on Mobile Robots (ECMR)*. 2015, pp. 1–8. DOI: 10.1109/ECMR.2015.7324048 (cit. on p. 22).

[41] Juan Pablo Rodríguez-Gómez, Raul Tapia, Maria del Mar Guzmán Garcia, Jose Ramiro Martínez-de Dios, and Anibal Ollero. «Free as a Bird: Event-Based Dynamic Sense-and-Avoid for Ornithopter Robot Flight». In: *IEEE Robotics and Automation Letters* 7.2 (2022), pp. 5413–5420. DOI: 10.1109/LRA.2022.3153904 (cit. on pp. 22, 23).

[42] Bas J. Pijnacker Hordijk, Kirk Y. W. Scheper, and Guido C. H. E. de Croon. «Vertical landing for micro air vehicles using event-based optical flow». In: *Journal of Field Robotics* 35.1 (Nov. 2017), pp. 69–90. ISSN: 1556-4967. DOI: 10.1002/rob.21764 (cit. on pp. 23, 94).

[43] F. Paredes-Vallés, J. J. Hagenaars, J. Dupeyroux, S. Stroobants, Y. Xu, and G. C. H. E. de Croon. «Fully neuromorphic vision and control for autonomous drone flight». In: *Science Robotics* 9.90 (2024), eadi0591. DOI: 10.1126/scirobotics.adi0591 (cit. on p. 23).

[44] Daniel Gehrig and Davide Scaramuzza. «Low-latency automotive vision with event cameras». In: *Nature* 629 (2024), pp. 1034–1040. DOI: 10.1038/s41586-024-07409-w (cit. on p. 23).

[45] Guang Chen, Hu Cao, Jorg Conradt, Huajin Tang, Florian Rohrbein, and Alois Knoll. «Event-Based Neuromorphic Vision for Autonomous Driving: A Paradigm Shift for Bio-Inspired Visual Sensing and Perception». In: *IEEE Signal Processing Magazine* 37.4 (2020), pp. 34–49. DOI: 10.1109/MSP.2020.2985815 (cit. on p. 23).

[46] Vincent Brebion, Julien Moreau, and Franck Davoine. «Real-Time Optical Flow for Vehicular Perception With Low- and High-Resolution Event Cameras». en. In: *IEEE Transactions on Intelligent Transportation Systems* 23.9 (Sept. 2022), pp. 15066–15078. ISSN: 1524-9050, 1558-0016. DOI: 10.1109/TITS.2021.3136358 (cit. on p. 31).

[47] Anton Mitrokhin, Cornelia Fermuller, Chethan Parameshwara, and Yiannis Aloimonos. «Event-based Moving Object Detection and Tracking». In: *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. arXiv:1803.04523 [cs]. Oct. 2018, pp. 1–9. DOI: 10.1109/IROS.2018.8593805 (cit. on p. 32).

[48] Inivation AG. *Event Accumulation*. Mar. 2024 (cit. on p. 32).

[49] Antoine Beyeler, Jean-Christophe Zufferey, and Dario Floreano. «Vision-based control of near-obstacle flight». en. In: *Autonomous Robots* 27.3 (Oct. 2009), pp. 201–219. ISSN: 0929-5593, 1573-7527. DOI: 10.1007/s10514-009-9139-6 (cit. on p. 48).

[50] Jean-Christophe Zufferey, Adam Klaptocz, Antoine Beyeler, Jean-Daniel Nicoud, and Dario Floreano. «A 10-gram Microflyer for Vision-based Indoor Navigation». en. In: () (cit. on p. 48).

[51] Simon Zingg, Davide Scaramuzza, Stephan Weiss, and Roland Siegwart. «MAV navigation through indoor corridors using optical flow». en. In: *2010 IEEE International Conference on Robotics and Automation*. Anchorage, AK: IEEE, May 2010, pp. 3361–3368. ISBN: 978-1-4244-5038-1. DOI: 10.1109/ROBOT.2010.5509777 (cit. on pp. 48, 50, 51).

[52] Farid Kendoul, Isabelle Fantoni, and Kenzo Nonami. «Optic flow-based vision system for autonomous 3D localization and control of small aerial vehicles». In: *Robotics and Autonomous Systems* 57.6 (June 2009), pp. 591–602. ISSN: 0921-8890. DOI: 10.1016/j.robot.2009.02.001 (cit. on p. 48).

[53] K. Bruce, J.R. Kelly, and L. Person Jr. «NASA B737 flight test results of the Total Energy Control System». en. In: *Astrodynamics Conference*. Williamsburg,VA,U.S.A.: American Institute of Aeronautics and Astronautics, Aug. 1986. DOI: 10.2514/6.1986-2143 (cit. on p. 60).

[54] PX4 Autopilot. *Controller Diagrams* (cit. on p. 63).

[55] Jawad N. Yasin, Sherif A. S. Mohamed, Mohammad-hashem Haghbayan, Jukka Heikkonen, Hannu Tenhunen, Muhammad Mehboob Yasin, and Juha Plosila. «Night vision obstacle detection and avoidance based on Bio-Inspired Vision Sensors». In: *2020 IEEE SENSORS*. ISSN: 2168-9229. Oct. 2020, pp. 1–4. DOI: 10.1109/SENSORS47125.2020.9278914 (cit. on p. 64).

[56] Jeffrey Delmerico, Titus Cieslewski, Henri Rebecq, Matthias Faessler, and Davide Scaramuzza. «Are We Ready for Autonomous Drone Racing? The UZH-FPV Drone Racing Dataset». en. In: *2019 International Conference on Robotics and Automation (ICRA)*. Montreal, QC, Canada: IEEE, May 2019, pp. 6713–6719. ISBN: 978-1-5386-6027-0. DOI: 10.1109/ICRA.2019.8793887 (cit. on p. 70).

[57] Valentin Wüest, Enrico Ajanic, Matthias Müller, and Dario Floreano. «Accurate Vision-based Flight with Fixed-Wing Drones». In: *2022 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2022, pp. 12344–12351. DOI: 10.1109/IROS47612.2022.9981921 (cit. on p. 82).

[58] Mathias Gehrig, Manasi Muglikar, and Davide Scaramuzza. «Dense Continuous-Time Optical Flow From Event Cameras». In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 46.7 (July 2024). Conference Name: IEEE Transactions on Pattern Analysis and Machine Intelligence, pp. 4736–4746. ISSN: 1939-3539. DOI: 10.1109/TPAMI.2024.3361671 (cit. on p. 93).

[59] Mathias Gehrig, Mario Millhäusler, Daniel Gehrig, and Davide Scaramuzza. *E-RAFT: Dense Optical Flow from Event Cameras*. arXiv:2108.10552 [cs]. Oct. 2021. DOI: 10.48550/arXiv.2108.10552 (cit. on p. 93).

[60] Zhexiong Wan, Yuchao Dai, and Yuxin Mao. «Learning Dense and Continuous Optical Flow from an Event Camera». In: *IEEE Transactions on Image Processing* 31 (2022). arXiv:2211.09078 [cs], pp. 7237–7251. ISSN: 1057-7149, 1941-0042. DOI: 10.1109/TIP.2022.3220938 (cit. on p. 93).

[61] Javier Cuadrado, Ulysse Rançon, Benoit R. Cottereau, Francisco Barranco, and Timothée Masquelier. «Optical flow estimation from event-based cameras and spiking neural networks». English. In: *Frontiers in Neuroscience* 17 (May 2023). Publisher: Frontiers. ISSN: 1662-453X. DOI: 10.3389/fnins.2023.1160034 (cit. on p. 93).

[62] Daniel R. Kepple, Daewon Lee, Colin Prepsius, Volkan Isler, Il Memming Park, and Daniel D. Lee. «Jointly Learning Visual Motion and Confidence from Local Patches in Event Cameras». en. In: *Computer Vision – ECCV 2020*. Ed. by Andrea Vedaldi, Horst Bischof, Thomas Brox, and Jan-Michael Frahm. Vol. 12351. Series Title: Lecture Notes in Computer Science. Cham:

Springer International Publishing, 2020, pp. 500–516. ISBN: 978-3-030-58538-9. DOI: 10.1007/978-3-030-58539-6_30 (cit. on p. 93).

[63] Chankyu Lee, Adarsh Kumar Kosta, Alex Zihao Zhu, Kenneth Chaney, Kostas Daniilidis, and Kaushik Roy. «Spike-FlowNet: Event-Based Optical Flow Estimation with Energy-Efficient Hybrid Neural Networks». en. In: *Computer Vision – ECCV 2020*. Ed. by Andrea Vedaldi, Horst Bischof, Thomas Brox, and Jan-Michael Frahm. Cham: Springer International Publishing, 2020, pp. 366–382. ISBN: 978-3-030-58526-6. DOI: 10.1007/978-3-030-58526-6_22 (cit. on p. 93).

[64] Chengxi Ye, Anton Mitrokhin, Cornelia Fermüller, James A. Yorke, and Yiannis Aloimonos. *Unsupervised Learning of Dense Optical Flow, Depth and Egomotion from Sparse Event Data*. arXiv:1809.08625 [cs]. Feb. 2019. DOI: 10.48550/arXiv.1809.08625 (cit. on p. 93).

[65] Jun Nagata, Yusuke Sekikawa, and Yoshimitsu Aoki. «Optical Flow Estimation by Matching Time Surface with Event-Based Cameras». en. In: *Sensors* 21.4 (Jan. 2021). Number: 4 Publisher: Multidisciplinary Digital Publishing Institute, p. 1150. ISSN: 1424-8220. DOI: 10.3390/s21041150 (cit. on p. 95).

[66] Shintaro Shiba, Yoshimitsu Aoki, and Guillermo Gallego. «Fast Event-based Optical Flow Estimation by Triplet Matching». In: *IEEE Signal Processing Letters* 29 (2022). arXiv:2212.12218 [cs, eess], pp. 2712–2716. ISSN: 1070-9908, 1558-2361. DOI: 10.1109/LSP.2023.3234800 (cit. on pp. 95, 96).

[67] Bodo Rueckauer and Tobi Delbruck. «Evaluation of Event-Based Algorithms for Optical Flow with Ground-Truth from Inertial Measurement Sensor». English. In: *Frontiers in Neuroscience* 10 (Apr. 2016). Publisher: Frontiers. ISSN: 1662-453X. DOI: 10.3389/fnins.2016.00176 (cit. on p. 96).

[68] Till Kroeger, Radu Timofte, Dengxin Dai, and Luc Van Gool. *Fast Optical Flow using Dense Inverse Search*. arXiv:1603.03590 [cs]. Mar. 2016. DOI: 10.48550/arXiv.1603.03590 (cit. on p. 97).

Acknowledgements

I would like to express my deepest gratitude to all those who have supported me throughout the course of my thesis. Without their guidance, encouragement, and expertise, this work would not have been possible.

First and foremost, I am profoundly grateful to Professor Dario Floreano, head of the LIS Lab at EPFL, for offering me the opportunity to conduct my research as a visiting student. His visionary insights and invaluable feedback have been instrumental in shaping this thesis, and I deeply appreciate his willingness to share his opinions and knowledge with me.

A special thank you goes to my internal supervisor at Politecnico di Torino, Professor Marcello Chiaberge, for having accepted me as a student and for his support to assist me.

My heartfelt thanks to Charbel Toumieh, one of my supervisor at EPFL, for his guidance, encouragement, and technical support throughout this project. Most of the work presented in this thesis would not have been possible without his knowledge and expertise.

Lastly, my deepest gratitude goes to Simon Jeger, who has been my primary supervisor at EPFL. He has been an invaluable mentor, assistant and most importantly, a friend. He helped me in every step of the way, from the initial idea to the final submission. He showed incredible patience and dedication, always pushing me to do better, even when I thought I couldn't.

I would also like to thank all the colleagues and friends I met during my time at EPFL. The collaborative and stimulating environment of the LIS Lab helped me grow both as a researcher and as a person. Finally, to my family and friends, your unwavering support throughout this journey has meant the world to me, and I am forever grateful.

Thank you all for helping me reach this important milestone.