Fully event-based motion estimation of neuromorphic binocular systems

Laurent Dardelet

▶ To cite this version:

Laurent Dardelet. Fully event-based motion estimation of neuromorphic binocular systems. Artificial Intelligence [cs.AI]. Sorbonne Université, 2021. English. NNT : 2021SORUS329 . tel-03554749

HAL Id: tel-03554749
https://theses.hal.science/tel-03554749v1

Submitted on 3 Feb 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

SORBONNE UNIVERSITÉ

INSTITUT DE LA VISION PARIS

THÈSE DE DOCTORAT
DE SORBONNE UNIVERSITÉ

Spécialité : Ingénierie Neuromorphique

École doctorale nº391: Sciences mécaniques, acoustique, électronique et robotique

réalisée

à l'Institut de la Vision - Équipe vision et calcul naturel

sous la direction de Sio-Hoi Ieng et Ryad B. Benosman

présentée par

Laurent Dardelet

pour obtenir le grade de :

DOCTEUR DE SORBONNE UNIVERSITÉ

Sujet de la thèse :

Fully event-based motion estimation of neuromorphic binocular systems

soutenue le 15 Décembre 2021

devant le jury composé de :

Pr.	Bernabe Linares-Barranco	Rapporteur
Dr.	Laurent Perrinet	Rapporteur
Pr.	Bruno Gas	Examinateur
Dr.	Sio-Hoi Ieng	Directeur de thèse
Pr.	Ryad B. Benosman	Co-Directeur de thèse

Fully event-based motion estimation of neuromorphic binocular systems

Abstract: Through the advances of artificial retinas, neuromorphic computation has been an increasing research field over the past 20 years. Applications in robotics, autonomous driving, medical equipment are starting to appear. However, some major issues remain, as methods used too often mimic, if not simply adapt standard frame-based techniques developed for fundamentally different type of data. These techniques often process batches of data, performing global optimization while forgetting the fundamental nature of events. As they present infinitesimal changes of the scene, we believe they should be treated as such upon computation. This thesis focuses on the case on visual odometry to develop fully event-based computation techniques, by using the entire advantages given by the neuromorphic sensors available. Using infinitesimal updates, we developed low-latency algorithms, while handling vastly different scene dynamics. By carefully analyzing events streams, we believe low latency can be achieved at low computational cost, showing once again that neuromorphic engineering is a way to reduce computer vision energy footprint.
The first part tackles the problem of on-screen tracking. Through the use of an inertial model, we developed a high-frequency tracking solution, with no prior required about the concerned shape. A second set of algorithms showcases how to compute optical flow and depth from a binocular system, and use them in an asynchronous way to compute a visual sensor ego-kinetics estimation. The third part combines the two previous one into an virtual model, to recover online pose with little to no assumption about the scene. Finally, a last part analyses more in-depth the importance of time in the event-based paradigm, and describes the development framework solution implemented to allow correct and efficient time handling.

Keywords : Neuromorphic engineering, event-based processing, tracking, visual odometry, Python language

Estimation évènementielle du déplacement de systèmes neuromorphiques binoculaires

Résumé : L'ingeniérie neuromorphique est un domaine en pleine expansion depuis une vingtaine d'années, notamment grace aux développement de rétines artificielles. Leurs applications concrètes sont en train d'apparaitre, en particulier dans les domaines de la robotique, de la contuite autonome ou encore dans le médical. Cependant, des problèmes fondamentaux persistent dans le traitement même de ces données d'un type nouveau. Les méthodes utilisées essayent trop souvent d'imiter celles adaptées aux flux vidéos conventionnels, voire même de les reprendre entièrement. Ces méthodes, qui traitent de manière synchrone des ensembles de donnees ne sont pas adaptées à la nature fondamentale du signal évènementiel. Les calculs fait sur ces données devraient, en toute logique, respecter et mettre en avant cette nature discrète des évènements. Cette thèse utilise comme support l'odométrie visuelle pour développer des techniques purement évènementielles qui tirent profit au maximum des avantages des capteurs neuromorphiques. Par l'utilisation de mises à jour asynchrones et infinitésimales, on montre qu'il est possible de procéder à des calculs préservant une latence temporelle faible, et ce avec moindre coùt de calcul, réduisant d'autant l'empeinte énergétique de tels systèmes. La premiere partie s'attachera au suivi d'objets dans des scenes, grâce à un model inertiel mis à jour de maniere évènementielle. Un deuxième ensemble d'algorithmes permettra le calcul des vitesses de déplacement d'un système de vision binoculaire, *via* le calcul du flot optique et l'estimation de profondeur des objets visibles. Une troisième partie combinera les deux précédentes au sein d'un systeme virtuel inertiel, permettant d'estimer la position dans l'espace d'un système de vision binoculaire au cours du temps. Enfin, une dernière partie s'attachera à une analyse plus poussée du paradigme neuromorphique, en traitant notamment de la place de la donnée temporelle dans ce dernier. Il sera également présenté l'environnement développé au cours de cette thèse, permettant de traiter de manière pertinente cette donnée temporelle des évènements.

Mots clés : Ingénierie neuromorphique, traitement du signal évènementiel, suivi d'objets, odométrie visuelle, language Python

Open-source software

- **PEBBLE**: Event-based framework based on code ergonomy and data accessibility.
`https://github.com/LDardelet/PEBBLE`
- **Event-Based Tracker**: Python standalone implementation of the event-based tracking algorithm described in Chapter 1.
`https://github.com/neuromorphic-paris/eb-tracker/`
- $3D$ **Data Handling** : Python environment handling $3D$ data relations inside reference frames.
`https://github.com/LDardelet/DataScripts/`

Contents

Introduction **1**

State of the art **7**
Neuromorphic sensors . . . 7
Neuromorphic computation . . . 8
Shape tracking . . . 9
Optical flow . . . 10
Depth estimation . . . 10
Ego-motion . . . 10

1 An Event-by-Event Feature Detection and Tracking Invariant to Motion Direction and Velocity **13**
1.1 Introduction . . . 13
1.2 Incremental Continuous Detection and Tracking . . . 14
1.2.1 Trackers definition, position and angle update . . . 14
1.2.2 Tracker activity . . . 15
1.2.3 Optical flow . . . 15
1.2.4 Contour velocity estimation . . . 16
1.2.5 Normalization for the tracker speed update . . . 19
1.2.6 Time constant definition . . . 22
1.2.7 Spatial descriptor and tracker lock . . . 23
1.2.8 Illustration case . . . 24
1.2.9 Algorithms . . . 25
1.3 Experiments and performances evaluation . . . 28
1.3.1 Comparison with Frame-based tracking . . . 29
1.3.2 Benchmark with the Event Camera Simulator . . . 29
1.3.3 Benchmark on event camera dataset . . . 29
1.3.4 Panoramic reconstruction . . . 30
1.3.5 Comparison with groundtruth with IMU data . . . 30
1.3.6 Trackers stability and latency . . . 33
1.4 Conclusion . . . 33

2 Stereo Matching and Visual Ego Kinetics **35**
2.1 Introduction . . . 35
2.2 Method . . . 37

2.2.1 Gradient-Based Optical flow 37
2.2.2 Event-based Stereo Matching 39
2.2.3 Kinematic from depth map and optical flow 43
2.2.4 Maps life-time and and inter-maps latencies 46
2.3 Experiments 47
2.3.1 Optical Flow benchmark 47
2.3.2 Stereo Matching Benchmark 47
2.3.3 Ego-kinetics evaluation 48
2.4 Conclusion 53

3 Binocular visual odometry 55
3.1 Introduction 55
3.2 Proposed method 57
3.2.1 Pose Estimator 57
3.2.2 Physical solver model 58
3.2.3 Viscous fluid environment 58
3.2.4 Trackers and depth as stabilization data 59
3.2.5 Physical model equations 65
3.2.6 Parameters estimation through oscillator stability 66
3.3 Preliminary results 68
3.3.1 Data Simulator 69
3.3.2 Noiseless results 69
3.3.3 Adding noise to synthetic inputs 70
3.4 Conclusion 72

4 Events Dynamics & Processing 73
4.1 Introduction 73
4.2 Time constants analysis 74
4.2.1 Time as the elementary data : an overview 74
4.2.2 From low-level to high-level time constant 77
4.2.3 Retrieving past events 81
4.2.4 Dynamic window experiment 83
4.2.5 Conclusion on events timescales 85
4.3 PEBBLE : Python Event-Based BLEnder Framework 86
4.3.1 Development frameworks overview 86
4.3.2 Describing an event 87
4.3.3 Modular conception of a framework 90
4.3.4 Time propagation 92
4.3.5 Data interaction 93
4.3.6 Conclusion on PEBBLE 95

Conclusion & Perspectives 97

List of Figures 101

Bibliography 103

Introduction

The development of computer vision is intimately related to advances made in robotics, as it is a key component to their autonomy. As one of the goal of robotics is to make our daily life easier by entrusting to machines hard and repetitive labour, robots need this ability to analyze and behave according to their surroundings, in a similar way to humans. The very first robots given an artificial vision were developed in the late 40's, by the neurobiologist William Grey Walter. These two robots, named Elmer and Elsie, shown on the right side of figure 0.1, were able to perceive their environment via a crude photoelectric cell. If their behaviour now seems primitive, this accomplishment started paving the way for artificial vision and robotics as we know today.

Figure 0.1 *Pictures of one of the first autonomous robots, Elmer and Elsie (left) developed by Willian Grey Walter compared to state-of-the-art bipedal robot Atlas, from Boston Dynamics (right).* 70 *years separate those robots, but most of robotic developments were made possible by recent computers and computer vision advances that happened in the last two decades.*

Yet, it took a long time for machines to really interact with their surroundings thanks to vision sensors. Most industrial machines used throughout the 20th century were either human-controlled, or packed with various sensors, and confined within a controlled environment dedicated for specific tasks. While analog image representation and processing saw no major development during the decades they were used, digitization unlocked rapid improvements of the field. The miniaturization of computers, the increasing resolution of digital cameras, alongside with the development of computer vision methods, allowed for large improvements in machines environment interaction - including drones, autonomous

cars, robots, ... These improvements increase machines' possibilities and put within their reach domains that were previously accessible to humans only. An example of these advances Atlas, robot developed by Boston Dynamics and shown on the right side of figure 0.1. Thanks to computer vision, this bipedal robot is able to analyze rough terrains, and perform movements one could not think possible 20 years ago.

However, the way digital images and videos are encoded and processed cannot be further away from human vision, even compared to Elmer and Elsie's primitive artificial eyes. Those were barely photoreceptors followed by a purely analogical processing. This means that Walter's systems were naturally asynchronous, in the sense that they didn't rely on some clock, with step-by-step computations as modern computer vision does. It also means that these systems had a very low latency by design. In contrast, computer vision nowadays is inherently a step-by-step process. Put aside the computation itself, the sampling of visual scenes in traditional computer vision is also governed by master clocks. This translates into a batch acquisition of the visual scene and higher latencies.

Modern imaging systems capture and display images at typical frequencies of about $50Hz$. Those frequencies are set just high enough for humans to perceive successive images as a continuous motion. In our daily lives, this capture/display paradigm is not a limitation, as it allows for simple systems, easy to use and well adapted to our perception. It made this frame-based paradigm the basis of modern computer vision. However the limitations of this paradigm become salient in more advanced use cases such as robotic vision. Constant frequency sampling can lead to both over and under-sampling in the same visual scene, generating both over-computation and motion blur. The fact that frame-based compression algorithms are able to reduce raw video streams size by an usual factor of 50, and up to 200 in extreme cases[1] is a strong hint that constant frequency sampling is not an optimal way to acquire visual information.

Despite these limitations, digital image processing has achieved tremendous and exciting results over time and progress are still being made today. The processing techniques are so successful that they cross boundaries to be applied to other fields: Artificial Neural Networks (ANN) that were initially developed and designed to process digital images [1, 2, 3], are now largely used for sound processing [4], in electrical engineering [5], or even applied to biochemistry [6]. The extension to those domains is straightforward as data is acquired according to the same fixed frequency sampling principle. This paradigm led to the development of dedicated hardware such as the Graphical Processing Unit (GPU), whose computation power increases exponentially over the last two decades. Thanks to components miniaturization, GPUs and multicore CPUs made efficient computing architectures available to everyone via computers, smartphones, or even smart-watches. However, this efficiency increase is reaching its limits.

Electronic components cannot be shrunk indefinitely as quantum effects become sources of

[1] `https://www.lensrentals.com/blog/2010/01/video-compression-explained/`

excessive computation errors. Hence, gaining more computational power by increasing the chip's density will reach a limit. An alternative way to increase the number of transistors in a chip is to adopt a 3D solution, by stacking silicon wafers. From an exponential increase in computation power, the next years should display a more linear trend. Another consequence of this critical size is the limit of components power efficiency. As shown by figure 0.2, the computational efficiency has been steadily increasing since 1946. However, as physics constraints starts to appear, this computational efficiency is expected to reach a plateau in the near future. If the energy footprint of digital devices has been on a 4% average increase per year[2], we can expect this trend to increase even more as this maximum efficiency is being reached. With this considerations in mind, we need to look for new ways to acquire and process data and in particular the visual one, to exit from this unsustainable future.

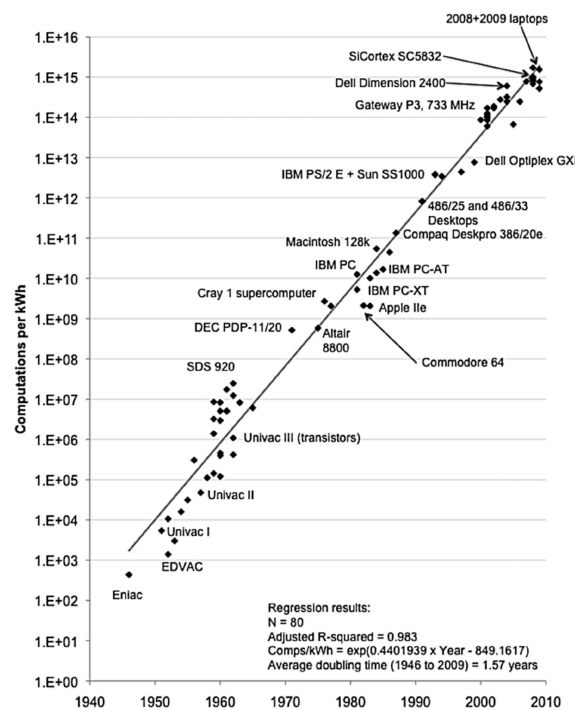

Figure 0.2 *Computational efficiency have doubled every* 1.57 *year on average between 1946 and 2009. However, a lower limit in components size is being reached, and with it this energy efficiency increase will shrink in the near future. The constant demand for additional computational power will result in an acceleration of the digital energy footprint. Graph taken from [7].*

Biological systems are known to be very energy efficient and for this reason, are often a source of inspiration for machines. Comparisons are often made between the human brain and CPUs or GPUs, and it is true that the difference in energy efficiency is staggering. However, this comparison must be taken with caution, as brains and computers are designed to perform very different tasks. If we are able to estimate the power consumption

[2]https://theshiftproject.org/wp-content/uploads/2019/03/Lean-ICT-Report_The-Shift-Project_2019.pdf

of a human brain, at about 20 watts [8], it is on the other hand very hard - if not impossible - to quantify the number of operations it performs and compare it to a computer. However, tasks such as object recognition, visual odometry, environment mapping can be performed by both systems, and can be used to compare them. For these tasks, brains are outperforming state-of-the-art algorithms. The study of biological models provides therefore great examples to create and improve artificial computational systems. This led to the field of Neuromorphic Engineering.

This domain takes inspirations from biological neurons for information acquisition and processing. Visual information processing is one of the most prominent example with the development of silicon retinas, also known as event-based vision sensors [9, 10, 11, 12]. These devices trade the fixed-frequency sampling paradigm for the event-driven one: acquisitions are not triggered by a master clock but rather by changes that occur in the visual scene. Because each pixel is asynchronous, independent and reacts directly to changes in the visual scene, the event-based vision sensor is impervious to under and oversampling problems. At the processing level, instead of working with frames, *event-based* computation, as its name implies, is triggered by events. Events are quanta of data, which can be related to spikes the biological eye sends to the brain. As the paradigm shifts from images to quanta of information, most computer vision algorithms, designed to work on large batches of dense data, become ill-fitted to process events that are inherently sparse.

Since neuromorphic sensors became available, many people have transposed existing frame-based algorithms to the event-based paradigm, focusing on the implementation itself. This can be understood, as it allows to test validated methods, just on different data. However, it is sub-optimal and misses the central fact that the output of such sensors is sparse and asynchronous. The nature of the data itself imposes to change the computation paradigm, shifting from heavy operations made on constant-size batches of data, to light-weight, elementary operations, performed on local changes. The advantages of event-triggered and light operations are numerous, as they allow for low-latency and high-speed computation. But also, keeping the sparse property of the data at the center of its processing allows to keep the energy consumption at low levels [13, 14, 15].

One of the motivations behind this thesis is to change this computation paradigm, by constantly challenging the legitimacy of classical computation techniques used in frame-based computer vision. Beyond the simple objective of adapting frame-based techniques to event-based data, we wanted to explore new ways to tackle computer vision problems. We wanted to process events in an asynchronous manner, considering local neighborhoods. The core of this thesis aimed to find a new event-based method for visual odometry.

It is often said that the development of neuromorphic engineering is limited by the lack of event-based dedicated hardware[3]. This is similar to quantum computing, for which algorithms have been known since the 1980s [17, 18], while still missing the hardware to

[3]Neuromorphic processors [16] do exist, but mainly focus on Spiking Neural Networks implementations.

implement them[4]. A computational paradigm has first to set up its theoretical grounds, before hardware development can follow up based on the extracted requirements. In the case of this work, we acknowledged this lack of hardware, by focusing on the algorithmic side of things, rather than its actual implementation. The experiments are made to produce results and proofs of concept, rather than efficiency. As event-based computation sorts out its pros and cons, we believe neuromorphic hardware development will be led by well-defined guidelines, improving the data processing efficiency, alongside with reducing the energy consumption.

We will keep time as the centerpiece of all our techniques developed in this work. The asynchronous nature of events makes their timing the most important information medium. Similar to the ordering of frames inside a conventional video stream, time orders events, and states their relations to one another. However, its reach extends far beyond just ordering data, and as such, the consideration of time for event-based data processing has been a constant matter throughout this work. The second fundamental guideline of our work is data hierarchy. We believe the neuromorphic world, in its debuts, has to focus on simple computing blocks that extract low level information from the data. Higher level building blocks will feed on this low level data to produce higher-level information. For any task, we believe it is important to first fully understand the basics of both the input and the output of an algorithm, prior to trying black-box methods, such as ANNs, or for more neuromorphic compatible networks, Spiking Neural Networks (SNNs).

Neuromorphic sensors show particular abilities - low latency, high dynamic range, power efficiency - that are especially fit for robotic applications. Robotics had to wait for the development of highly capable computers in order to gain some autonomy, but the energy consumption of these embarked computers restrict their autonomy, both in terms of energy storage and hardware requirement. Simultaneous Localization And Mapping (SLAM), which is the holy grail for autonomous robotics, at best requires state-of-the-art hardware to run, along with multiple sensors for accuracy [19]. Thus, we propose in this work to apply event-based ground-rules to motion estimation or better known as visual odometry problem, purely based on visual information captured by neuromorphic sensors. By extracting higher and higher level data from these sensors, and combining them, we show that it is possible to recover motion from asynchronous data, without the need of cost prohibitive optimization processes on accumulated batches of data.

This thesis is structured into four main chapters:

1. The first chapter introduces a fully event-based method to track features in a scene recorded by a single event-based camera. This technique features both detection of objects of interest, along with high-frequency asynchronous updated tracking. The closed-loop cycle implemented notably allows to consider time on dynamical scales, making the tracking process robust to large variations in signal dynamics.

[4] If some advances have been made recently, the domain is still decades away from a working, configurable quantum computer.

2. The second chapter will present a technique to recover the velocity of an event-based stereo-rig by using optical flow and depth information. The presented optical flow computation method allows for an improvement in the amplitude accuracy compared to existing methods, while using a fully event-based algorithm and light computation. The stereo-matching method makes use of a pair of rectified sensors, and allows for a fast and reliable disparity computation through the use of $1D$ spatial descriptors. Finally, using the previously obtained features we can estimate the instantaneous translation and rotation velocities of a stereoscopic visual system, that we referred to as *ego-kinetics*.

3. The third chapter presents a model that fuses previously obtained information : tracking, depth and velocity information. This allows to compute a visual odometry. By using an event-driven physical simulation, we show that pose can be updated at the scale of the event. This is achieved by rephrasing the classical heavy optimization computation into a incremental computation.

4. The fourth chapter will present a more in-depth analysis of why time is so important in event-based computation. We will also present a new framework. This framework, available in open-source, makes it easier to implement event-based algorithms, with easy data management and visualization.

State of the Art

Neuromorphic Sensors

Event-based vision sensors are gaining popularity within the computer vision community as they offer many advantages over conventional frame-based cameras that cannot be matched without an unreasonable increase in computational resources. Low computation requirements are achieved by reducing redundant visual data at the level of pixels. This naturally allows lower latency and precise temporal acquisition. Event-based cameras (Figure.0.3(a)) are based on an asynchronous level crossing sampling as shown in Figure.0.3(b)-(c).

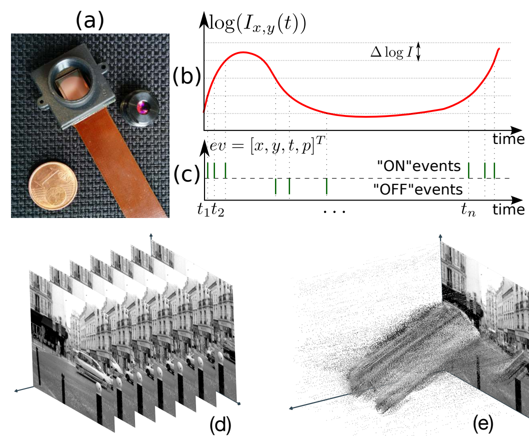

Figure 0.3 *Event-based acquisition of visual signals: (a) the ATIS Event-based camera [11], (b) principle of event-based sampling. The variations of the logarithm of the light intensity of a pixel located at $[x, y]^T$ over time, (c) asynchronous temporal contrast events generated each time the light intensity crosses a level in log(I), (d) the output of a conventional frame based cameras at some 30-60Hz vs in (e) the high temporal precision event-based output of the sensor at around 1μs.*

The pioneering work in this domain was made by Mahowald, in the 1980s [20]. this work fundamentally studies the biological eye structure, and aims to mimic its behaviour by presenting an electronic equivalent. This initial neuromorphic sensor [9] paved the way to numerous variants, with different philosophies and improving along with the technological advances.

Neuromorphic sensors can exploit gradient values, optical flow, or edge orientation [21, 22, 23] specifically. However, the most commonly used sensors are luminance-based, as they generates the most versatile type of events among visual neuromorphic sensors. Most notably, we can cite the first sensors of this type, like the DVS and the ATIS [10, 11, 12, 24]. The first one implemented the events are they are used nowadays, with the sign of the luminosity change, often referred to as polarity. The second one improves that concept by adding absolute luminance measurement. Both used the Address Event Representation (AER) standard [25], still in use now.
Over the years, improvement in this technology has lead to significant noise reduction, along with an increase of the average sensor resolution. VGA sensors definition were made available a few years ago, now even shifting to megapixel-resolutions [26]. The DAVIS [27], used thoroughly throughout this work, features in its 346×260 resolution version an Inertial Measurement Unit (IMU), along with frames recording and external synchronization.
We also point out the existence of neuromorphic cochleas [28, 29, 30, 31], audio counterparts to the artificial retinas. The later specifically uses the AER representation, allowing easier fusion sensor for neuromorphic computation.

Neuromorphic computation

Various works report on neuromorphic hardware development, mostly oriented towards Spiking Neural Networks (SNNs). We can most notably cite SpiNNaker chip [32], Synapse DARPA program [5] and Loihi Intel chip [33].
[34] presents a novel neuromorphic computation paradigm using time at its core, by fundamentally changing the way data is seen. Other works show that the use of event-based processing is already fitted for embarked computation, showing its low-power consumption on standard hardware [13] or developing directly on mobile phones [35].

However, the literature shows little work on the fundamentals of event-based processing, compared to the rest of the event-based literature. Most algorithms, whether that show fully-event based methods or accumulative ones often take time constant parameters for granted, tweaking them to get the best possible outcome, with little additional analysis. HOTS [36] shows how timescales affects data processing, by creating a hierarchy of spatial descriptors that use different decay rates. [37] presents a contrast maximization method to enhance frames generation, and question the importance of time through the use of optical flow. A dynamical timescale is proposed in [38] in order to improve optical flow and Lucas-Kanade implementation results in a closed loop. [39] presents a more profound analysis of time importance in neuromorphic computing, by considering the lifetime of events, and dynamic slicing.
Apart from time considerations, event-based processing is often split into two main categories, being accumulative versus event-by-event methods [40]. The former one processes event by clustering them within a temporal window to produce a spatial-only representation so it can be fed into more traditional processing pipelines [41, 42, 43, 44]. While such approaches seem to be intuitive, they are facing two major problems as they

[5] `https://en.wikipedia.org/wiki/SyNAPSE`

are discarding the most important advantages gained from the event-based representation: the temporal accuracy and the spatial sparseness. By building frames, not only additional computation resources are necessary but also, due to the lower spatial resolution of the sensor, the achieved performances of vision algorithms from these data are often lower than from data acquired with standard high resolution conventional cameras. The second category of event-based processing focuses on the sparsity of the data. This is one methodology encouraged and adopted by several event-based neuromorphic engineering techniques which consist in locally -spatially and temporally- processing each new event [45, 46, 47].

Shape tracking

Object tracking is a cornerstone for computer vision, with applications across multiple fields. It starts namely with Lucas and Kanade's work, in 1981 [48], which laid foundations for deterministic tracking, as opposed to learning techniques. This later category has been increasingly used in many different fields, and object tracking is no exception [49].
Previous works that tackled feature detection from the output of an event-based sensor include several corner detectors that are inferred from the optical flow computation [50], or obtained by using Harris operator on locally integrated temporal frame of events [51]. The time-surface as introduced in [52] is another way to extract dynamic patterns (which also includes corners) as it allows to define a compact feature-velocity descriptor around an incoming event. In [53], the time-surface allows to build a feature that can then be used with FAST [54] detector. In [55], a FAST detector is applied to the time surface to find corners and a graph-based technique is jointly used to find the closest corner in space and time that support motions assumption to achieve tracking. An improvement to this approach is introduced in [56] with the idea of using a local descriptor built from the time surface. The use of descriptors allows for tracking via minimizing descriptor differences. A hybrid class of approaches combine frames and events to overcome the difficulty of pattern selection and tracking using events as the same scene pattern can produce different events depending on the motion direction, making the event correspondences across time very challenging. In [57], the DAVIS sensor is used at the rate of 35 fps to extract corners and other traditional image features are detected on frames by applying standard computer vision detectors, while the tracking is operated on the events captured between the frames [58]. The same idea can be also found in [59]. An expectation-maximization approach, coupled with an iterative closest point algorithm is used in [60] to keep track of corners that are detected at the beginning of the sequences from an edge map built by accumulating events over a manually selected integration time. Events are integrated over a duration deduced from the optical flow and the edge are mapped to generate unitary contours to then apply an ICP algorithm [61]. The approach proposed in [62], based on global optimization of a cost function over a cluster of events, aims to find the motion parameters that yield a homography that maps the cluster of events into an “event image" with a maximal contrast. This approach while not based on features, still deals with the data association problem. It has some overlapping properties with our approach, such as the ability to relate the sensor's motion to the data, however they differ deeply due to their synchronous/asynchronous nature.

Optical flow

The optical flow design the velocity of an object on screen. A large gap exists between frame-based and neuromorphic sensors, as the high-temporal resolution of the later allows for precise - and instantaneous - velocities in each point where events occur. On the other hand, conventional cameras often present large displacements of the objects on two successive frames, and as such, the optical flow can refer to an overall displacement, rather than an actual speed. Thus, optical flows estimation methods from event-based sensors flourished as soon as they were introduced: adaption of the classical photoconsistency constraint [63] then techniques that steers away from frames [52]. Later, more complex approaches relying on learning techniques [64, 65] up to some improved aperture-robust computation technique [66].

Depth estimation

In the case of a scene recording, depth estimation will design any method allowing to estimate the distance of an object to the sensor[6]. Monocular methods exist, actively changing the focal length of the sensor to get focus or defocus on different objects. If the method exists since more than 20 years for conventional sensors [67], it was only recently adapted to event-based sensors [68]. However, without using the focal length as baseline unit for depth estimation, a stereoscopic system is required to obtain depth. Thus, depth estimation is often done using stereo-rectified synchronized sensors, and the depth is bijective to a disparity value, being the apparent displacement of an object between two sensors.
In this case, the problem can be formulated as pattern matching, with extensive frame-based research that started in the 80s [69]. Similar method were therefor adapted to the event-based paradigm, using time-surfaces as support [70, 71, 72, 73, 74]. Other work present stereo-matching solutions through spiking neural networks [75].

Ego-motion

We can consider two approaches for ego-motion estimation, being respectively dynamical estimation, recovering speed, acceleration, . . . and visual odometry.
That first category can also be referred to as ego-kinetics estimation. Previous works in this particular vision of things are rare. Two hypothesis can explain this lack of such approaches. The first one is that the initial frame-based paradigm offers little possibilities for such first order approach, as the discrete nature of frames makes this computation imprecise when scenes are being recorded at standard frequencies. As the displacements gets too high, the instantaneous velocity makes little sense to be determined, and approaches based on displacement instead are preferred. The second reason it that IMUs can provide such first-order information, at least for rotation. In this case, sensor fusion and be deemed more easy to implement that recovering a computer-vision based velocity estimation.
Visual odometry, on the other hand, have been an active field of study for decades. It mainly divides into two categories. The first is the bundle adjustment problem, which consists in

[6] In this case, the sensor center is often considered to be its focal center.

finding the optimal pose minimizing an error on a set of known $3D$ points reprojection. The more advanced formulation of this problem is the Simultaneous Localization And Mapping (SLAM). It consists in creating a $3D$ representation of the ongoing scene, and using it for visual odometry in a closed loop. Implementations can vary, especially in terms of sensors used, being possibly monocular or multi-sensors, along with additional sensors such as LIDARs, IMUs, ... [19]
Monocular visual odometry is an old problem even in event-based vision sensing. Previous works in this field rely largely on classical SLAM approaches developed for frame-based vision sensor [76, 77, 78, 79]. These works show how robotics is still in the search of low latency and low power strategies when it comes to fast autonomous navigation. On the other hand, stereo vision odometry using a pair of event-based vision sensor seems to attract less attention. Most notably, we have accounted the live demonstrator from Prophesee [80] and the work published in [74]. Without much of surprise, these works on event-based stereo visual odometry rely on a rigidly calibrated and epipolar rectified pair of event-based sensors to ensure spatial consistency. Their precise synchronization is additionally required to ensure the temporal consistency.
The works presented in [74] tackles localization and partially 3D mapping problems in a complex minimization pipeline. Semi dense depth maps are estimated by enforcing spatial and temporal consistency, followed by a depth fusion operation. This work focuses on a event-based approach by using Time-Surfaces as a support to process data. At the stage of depth estimation, a frame based strategy is applied at a rate of 100Hz to update the stereo time-surfaces.
More advanced SLAM techniques were developed using neuromorphic hardware for data acquisition or computation [81, 82].

Chapter 1

An Event-by-Event Feature Detection and Tracking Invariant to Motion Direction and Velocity

1.1 Introduction

This chapter introduces an event-based solution for selecting and tracking features from the output of an event-based camera. It overcomes for the first time the hard issue of extracting stable features from temporal information output by the sensor while being independent from motion direction and velocity changes. This is a major issue when using these sensors as the acquired space-time features can vary drastically although the underlying spatial structure is the same. This paper is based on a pure event per event methodology that allows it to make the most of event-based cameras' native properties of low latency and power. It permits for the first time to formulate the problem of features' selection and tracking in the time domain as an inter-dependent process between detection, tracking and continuous velocity estimation. The method does not assume any priors of what a feature is.
Event based acquisition introduces a major shift in the way visual information is being processed. Unlike conventional cameras that produce absolute intensity images at a fixed rate, event cameras rely on a data driven acquisition process where independent pixels asynchronously signal light intensity changes (coined as "events") at a high temporal resolution (usually around 1μs).
The use of sparse information represented by visual events can be puzzling to a conventional computer vision scientist used to deal with snapshots, gray levels and colors. Processing visual events requires new thinking to derive a novel family of algorithms capable of operating efficiently in the time domain by reducing memory footprints and computational power requirements.
With continuously improved spatial resolutions, almost matching conventional sensors (from VGA to Mega Pixel), the event-based vision sensors' high temporal resolution allows the measurement of velocities in the focal plane with great precision. This is the most valuable and reliable source of information of these sensors.

Presently, there are two opposite philosophies in processing data acquired by event cameras. The first one is advocating for the use of optimization techniques on static images of accumulated events or batches of events defined over different duration and sizes. This approach only uses the high dynamic range properties of these sensors, the recurring motivation behind this is to reuse decades of conventional computer vision techniques [54, 83] and avoid using single events that are wrongly considered to be too noisy and carrying little information. This approach results in wasted computation resources and an increased latency. The second approach is closer to the event-based acquisition properties. It is advocating for a local space-time processing of individual events, as they occur, with the idea that each one of them is contributing by an infinitesimal information to the sensing problem to solve. This approach naturally saves power and memory footprint because it only stores what is required and computes using the minimal amount of resources only when something has changed in the scene.

1.2 Incremental Continuous Detection and Tracking

1.2.1 Trackers definition, position and angle update

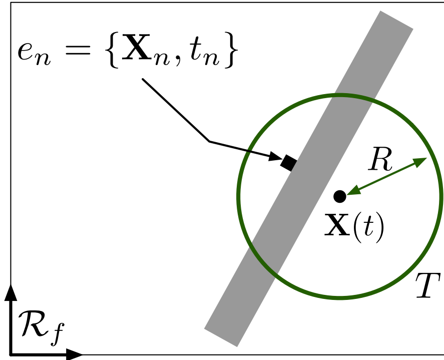

Figure 1.1 *A tracker T located at $\mathbf{X}$ is assigned an incoming event e_n at location $\boldsymbol{X_n}$ if $||\boldsymbol{X_n} - \boldsymbol{X}|| \leq R$, both expressed in $\mathcal{R}_f$ the coordinate reference of the focal plane*

We define a tracker T as a set $\{\boldsymbol{X}(t), \boldsymbol{v}(t), \theta(t), \omega(t)\}$, where $\boldsymbol{X}(t)$ is its position in the focal plane, $\boldsymbol{v}(t)$ its translation velocity, $\theta(t)$ its rotation angle w.r.t the optical axis $\boldsymbol{z}$ and $\omega(t)$ its rotation speed. An event output by the neuromorphic camera indexed by n is defined as the pair $e_n = \{\boldsymbol{X_n}, t_n\}$, with $\boldsymbol{X_n}$ its 2D spatial coordinates in the focal plane and t_n its time of occurrence. An incoming event is "assigned" to a tracker T when it is close enough to the tracker i.e. satisfying the spatial constraint: $||\boldsymbol{X_n} - \boldsymbol{X}|| \leq R$, where R is the spatial radius of the tracker (as shown in Figure 1.1).
If we assume a constant velocity motion of the tracker between consecutive event - a reasonable assumption due to the low latency and the high temporal accuracy of the event-based sensors - we can update its position $\boldsymbol{X}(t_n)$ and angle $\theta(t_n)$ at the time of the

arrival of e_n as:

$$\begin{cases} \boldsymbol{X}(t_n) & = \boldsymbol{X}(t_{n-1}) + (t_n - t_{n-1})\boldsymbol{v}(t_{n-1}) \\ \theta(t_n) & = \theta(t_{n-1}) + (t_n - t_{n-1})\omega(t_{n-1}) \end{cases} \tag{1.1}$$

This assumes that $\boldsymbol{v}$ and ω can be estimated at any time. This is partially achieved by computing the optical flow as explained later.
We define the location of that event in the tracker coordinate frame as $\boldsymbol{x_n}$, such that

$$\boldsymbol{x_n} = \varrho(-\theta(t_n))(\boldsymbol{X_n} - \boldsymbol{X(t_n)}) \tag{1.2}$$

with $\varrho(\theta)$ the $2D$ rotation matrix of angle θ.

1.2.2 Tracker activity

To enable the continuous update of a tracker T, we first define the time decay factor λ_n associated with T as:

$$\lambda_n = \exp\left(-(t_n - t_{n-1})/\tau(t_{n-1})\right). \tag{1.3}$$

where $\tau(t_n)$ is the time needed for the tracker to move by one pixel. The estimation of this dynamic time property of the tracker will be fully expressed in section 1.2.6. Times t_n and t_{n-1} correspond respectively to time of arrival of the incoming event e_n and the preceding one e_{n-1}.
We then define $\mathcal{A}$, initialized to 0 at t_0, as a measure of the activity of a tracker T:

$$\mathcal{A}(t_n) = \mathcal{A}(t_{n-1})\lambda_n + 1. \tag{1.4}$$

$\mathcal{A}$ is updated by each incoming event that is assigned to a tracker T. This form is thus a sliding average over a time period τ. Each variable defined by a similar equation is thus robust to noise, as well as reactive to variations on a timescale τ.

1.2.3 Optical flow

For an incoming event $e_n = \{\boldsymbol{x_n}, t_n\}$ assigned to T, we define the set $\mathcal{E}_n$ that contains all events that occurred in the time interval $[t_n - N_\tau(t_n), t_n]$, within a distance r to $\boldsymbol{x_n}$ (shown as a black circle in Figure.1.2) :

$$\mathcal{E}_n = \{e_k = \{\boldsymbol{x_k}, t_k\} \mid ||\boldsymbol{x_k} - \boldsymbol{x_n}|| \leq r \text{ and } t_k \in [t_n - N_\tau(t_n), t_n]\}. \tag{1.5}$$

The quantity $N_\tau(t) = N.\tau$ with τ the time needed for a unitary contour to move by one pixel, and N the number of lines considered. $N_\tau(t)$ expresses the time needed by the tracker to move by N pixels, it also defines an adaptive time window parametrized by the velocity of the tracker.
With $\mathcal{E}_n$, we estimate the local optical flow $\boldsymbol{f}_n$ (shown in blue in Figure 1.2) at $\boldsymbol{x_n}$ at time t_n as:

$$\boldsymbol{f}_n = (\boldsymbol{x_n} - \overline{\boldsymbol{x_k}})/(t_n - \overline{t_k}), \tag{1.6}$$

where $\overline{\boldsymbol{x_k}}$ and $\overline{t_k}$ are the spatial position and the timestamp averaged over the set $\mathcal{E}_n$.

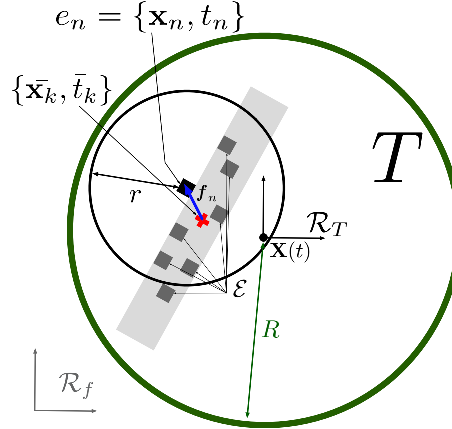

Figure 1.2 *The spatial coordinates of an incoming event e_n are now expressed in the tracker T coordinates system $\mathcal{R}_T$ at location $\boldsymbol{x}_n$. The optical flow $\boldsymbol{f}_n$ is computed as the difference between $\boldsymbol{x}_n$ and the mean positions of previous events happening in the temporal window $[t_n - N_\tau(t_n), t_n]$ contained in the set $\mathcal{E}_n$*

Remark1:

The computation of the optical flow as a difference between the mean spatio-temporal position of $\mathcal{E}_n$ and the current event works reliably provided one considers small spatial neighborhoods r (usually set between 2-4 pixels). The reliable direction of the estimate flow is the one orthogonal to local edges due to ambiguities related to the aperture problem. We will show in the following sections that local average flow allows the tracker to track local edges by introducing a metric that quantifies the aperture problem. We intentionally choose this simple form of flow instead of more accurate incremental flow computations techniques such as the canonical local plane fitting regularization used in [84, 52]. The main reason being it is computationally less prohibitive than most of the existing event-based optical flow algorithms and still proves to converge to an accurate flow estimation with the continuous update of the tracking procedure.

Remark2:

In the tracker's coordinate frame, all events generated by the same edge will follow a narrow spatial distribution that outlines the edge once the tracker velocity converges to the structure's one as shown in [85, 86]. The estimated flow, in the tracker's coordinate frame being the relative velocity between the tracker and the structure, will converge to $\mathbf{0}$ when the estimation is correct [87].

1.2.4 Contour velocity estimation

A 3 degree of freedom (2 for the translation and 1 for the rotation) tracker cannot be properly estimated from the optical flow alone as the flow provides only two translational components. The rotational component can be calculated from an estimator that integrates the optical flow over the time period τ, based on the motion equation expressed in the

tracker reference frame. We define $\boldsymbol{v_c} = (v_{c_x}, v_{c_y})^T$ and ω_c respectively the translation and rotation velocities of the contour in $\mathcal{R}_T$. These are the velocities we want to estimate and nullify by correcting the tracker velocities $\boldsymbol{v}$ and ω.

For any incoming event $e_n = \{\boldsymbol{x_n}, t_n\}$, and defining $\boldsymbol{x_n} = (x_n, y_n)^T$, the instantaneous velocity on the object at the event location is

$$\boldsymbol{v_n} = \boldsymbol{v}_c + \omega_c \boldsymbol{o_n}, \tag{1.7}$$

where $\boldsymbol{o_n} = (-y_n, x_n)^T$, the resulting vector in the focal plane of the cross product of $\boldsymbol{z} = (0,0,1)^T$ and $(\boldsymbol{x_n}^T, 0)^T$.

We then define the unit vector normal to that edge as $\boldsymbol{u_n} = (c_n, s_n)^T$. Hence, the optical flow $\boldsymbol{f_n} = (f_{n_x}, f_{n_y})^T$, that is the local velocity projected on $\boldsymbol{u_n}$ is equal to:

$$\boldsymbol{f_n} = (\boldsymbol{v_n}^T.\boldsymbol{u_n})\boldsymbol{u_n}. \tag{1.8}$$

We define the following quantities that will be used to estimate $\boldsymbol{v_c}$ and ω_c. They are updated in a similar way as the activity $\mathcal{A}$, as explained in section 1.2.2 :

$$\begin{cases} \Sigma_{\overline{f}_x}(t_n) & = \Sigma_{\overline{f}_x}(t_{n-1})\lambda_n + f_{n_x} \\ \Sigma_{\overline{f}_y}(t_n) & = \Sigma_{\overline{f}_y}(t_{n-1})\lambda_n + f_{n_y} \\ \Sigma_{\overline{f}_\omega}(t_n) & = \Sigma_{\overline{f}_\omega}(t_{n-1})\lambda_n + \boldsymbol{f_n}^T.\boldsymbol{o_n} \end{cases} \tag{1.9}$$

and we regroup these terms as well as the contour velocities in two vectors :

$$\boldsymbol{\Sigma} \equiv \begin{pmatrix} \Sigma_{\overline{f}_x} \\ \Sigma_{\overline{f}_y} \\ \Sigma_{\overline{f}_\omega} \end{pmatrix}, \boldsymbol{V_c} \equiv \begin{pmatrix} v_{c_x} \\ v_{c_y} \\ \omega_c \end{pmatrix} \tag{1.10}$$

The following lines provide calculations details, leading to compact form of equation 1.16.

$$\begin{aligned} \boldsymbol{f_n} &= \left((\boldsymbol{v}_c + \omega_c \boldsymbol{o_n}).(c_n, s_n)^T\right)(c_n, s_n)^T \\ &= \begin{pmatrix} c_n^2(v_{c_x} - y_n\omega_c) + c_n s_n(v_{c_y} + x_n\omega_c) \\ c_n s_n(v_{c_x} - y_n\omega_c) + s_n^2(v_{c_y} + x_n\omega_c) \end{pmatrix} \\ \boldsymbol{f_n}.\boldsymbol{o_n} &= (v_{c_x} - y_n\omega_c)(x_n c_n s_n - y_n c_n^2) \\ &\quad + (v_{c_y} + x_n\omega_c)(x_n s_n^2 - y_n c_n s_n) \end{aligned} \tag{1.11}$$

In matrix form, it gives us

$$\begin{aligned} \boldsymbol{f_n} &= \begin{pmatrix} c_n^2 & c_n s_n & x_n c_n s_n - y_n c_n^2 \\ c_n s_n & s_n^2 & x_n s_n^2 - y_n c_n s_n \end{pmatrix} \boldsymbol{V_c} \\ \boldsymbol{f_n}.\boldsymbol{o_n} &= \begin{pmatrix} x_n c_n s_n - y_n c_n^2 \\ x_n s_n^2 - y_n c_n s_n \\ x_n^2 s_n^2 + y_n^2 c_n^2 - 2x_n y_n c_n s_n \end{pmatrix}^T .\boldsymbol{V_c} \end{aligned} \tag{1.12}$$

We define the matrix M such that:

$$M \equiv \begin{pmatrix} r_{cc} & r_{cs} & r_{xcs} - r_{ycc} \\ r_{cs} & r_{ss} & r_{xss} - r_{ycs} \\ r_{xcs} - r_{ycc} & r_{xss} - r_{ycs} & r_{xxss} + r_{yycc} - 2r_{xycs} \end{pmatrix}, \tag{1.13}$$

with $r_{\alpha\beta uv}(t_n) = r_{\alpha\beta uv}(t_{n-1})\lambda_n + \alpha_n\beta_n u_n v_n$, for $\alpha, \beta \in \{x, y\}$ and $u, v \in \{c, s\}$.
This matrix holds purely spatial information about the contour, this information being composed of the events locations $\boldsymbol{x}_n$ along with the local contour directions $\boldsymbol{u}_n$. From this definition of M, equation 1.12 can be combined into :

$$\begin{pmatrix} \boldsymbol{f_n} \\ \boldsymbol{f_n}.\boldsymbol{o_n} \end{pmatrix} = \left(M(t_n) - M(t_{n-1})\lambda_n\right) \boldsymbol{V_c}. \tag{1.14}$$

And with definition given in equation 1.10, we obtain:

$$\begin{aligned} \boldsymbol{\Sigma}(t_n) - \boldsymbol{\Sigma}(t_{n-1})\lambda_n &= \left(M(t_n) - M(t_{n-1})\lambda_n\right) \boldsymbol{V_c} \\ \Leftrightarrow \boldsymbol{\Sigma}(t_n) &= M(t_n)\boldsymbol{V_c} + \lambda_n \left(\boldsymbol{\Sigma}(t_{n-1}) - M(t_{n-1})\boldsymbol{V_c}\right) \end{aligned}. \tag{1.15}$$

As $\boldsymbol{\Sigma}(t = 0) = \boldsymbol{0}$ and $M(t = 0) = 0$, by recurrence, for all t_n, we have the main relation:

$$\boldsymbol{\Sigma} = M\boldsymbol{V_c} \tag{1.16}$$

Hence, $\boldsymbol{V_c}$ is calculated by inverting M:

$$\boldsymbol{V_c} = M^{-1}\boldsymbol{\Sigma}. \tag{1.17}$$

Both M and $\boldsymbol{\Sigma}$ are updated in a purely-event-based fashion. $\boldsymbol{V_c}$ contains the contour velocities in $\mathcal{R}_T$ that we want to nullify, as that means a correct tracking of the contour (illustration by Figure 1.3).

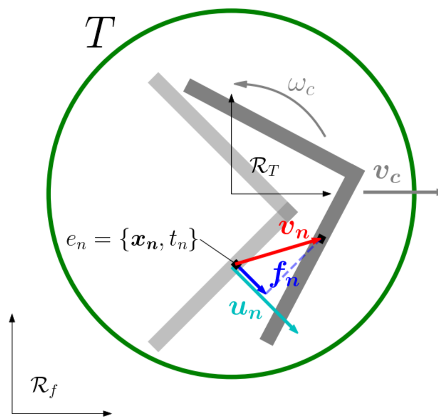

Figure 1.3 *Description of the contour velocities $\boldsymbol{v_c}$ and ω_c estimation. We update the vector $\boldsymbol{\Sigma}$ and the matrix M defined in equations 1.10 and 1.13 using event location $\boldsymbol{x_n}$, its orthogonal vector $\boldsymbol{o_n}$, optical flow $\boldsymbol{f_n}$ and unit vector $\boldsymbol{u_n} = \boldsymbol{f_n}/||\boldsymbol{f_n}||$. All those vectors are expressed in $\mathcal{R}_T$.*

1.2.5 Normalization for the tracker speed update

The estimated contour velocities $\boldsymbol{V_c}$ can be expressed as the opposites of errors vectors $\boldsymbol{\epsilon_v}$ and ϵ_ω between the tracker velocities $\boldsymbol{v}$ and ω, and the contour velocities $\boldsymbol{v_{c,\mathcal{R}_f}}$ and $\omega_{c,\mathcal{R}_f}$ in the local frame $\mathcal{R}_f$:

$$\begin{cases} -\boldsymbol{v}_c &= \boldsymbol{\epsilon_v} \equiv \boldsymbol{v} - \boldsymbol{v}_{c,\mathcal{R}_f} \\ -\omega_\epsilon &= \epsilon_\omega \equiv \omega - \omega_{c,\mathcal{R}_f} \end{cases}. \tag{1.18}$$

We aim for an exponential dampening of that velocity error, leading to a match between the tracker and the contour velocities:

$$\begin{cases} \frac{\partial \boldsymbol{v}}{\partial t} &= -\frac{\boldsymbol{\epsilon_v}}{\tau} = \frac{\boldsymbol{v_c}}{\tau} \\ \frac{\partial \omega}{\partial t} &= -\frac{\epsilon_\omega}{\tau} = \frac{\omega_c}{\tau} \end{cases}. \tag{1.19}$$

Let us define $\overline{\tau}$ the average time interval between two successive events, if $\overline{n}$ is the average number of events per second affecting that tracker, then :

$$\overline{n} = 1/\overline{\tau}. \tag{1.20}$$

Finally, we define $\boldsymbol{\delta v}_n$ and $\delta\omega_n$ the translation and rotation corrections applied to the tracker for the n-th event:

$$\begin{cases} \boldsymbol{\delta v}_n &= \boldsymbol{v}(t_n) - \boldsymbol{v}(t_{n-1}) \\ \delta\omega_n &= \omega(t_n) - \omega(t_{n-1}) \end{cases}. \tag{1.21}$$

On average, $t_{n-1} = t_n - \overline{\tau}$, thus by differentiating these equations and using equations 1.19, we obtain

$$\begin{cases} \boldsymbol{\delta v}_n &= \frac{\partial \boldsymbol{v}}{\partial t}(t_n)\overline{\tau} = \frac{\boldsymbol{v_c}\overline{\tau}}{\tau} \\ \delta\omega_n &= \frac{\partial \omega}{\partial t}(t_n)\overline{\tau} = \frac{\omega_c\overline{\tau}}{\tau} \end{cases}. \tag{1.22}$$

Let us consider now the evolution of the activity $\mathcal{A}$ defined in equation 1.4 between t and $t + \delta t$, with the constraint

$$\overline{\tau} \ll \delta t \ll \tau. \tag{1.23}$$

This constraint can be achieved, as $\overline{\tau}$ is the average time between two events, and τ is the time during which the tracker moves of one pixel : during τ, many events will occur, typically one for each pixel of the observed shape.
The exponential decay of the activity ensures that it reaches a steady state after some time. We approximate that the evolution of the activity over a restricted duration such as δt can be written as

$$\mathcal{A}(t + \delta t) \simeq \mathcal{A}(t)\rho(\delta t) + \Delta(\delta t) \tag{1.24}$$

with ρ being the decay term given by

$$\begin{aligned} &\rho(\delta t) = e^{-\frac{\delta t}{\tau}} \text{with } \delta t \ll \tau \text{ giving} \\ &\rho(\delta t) \simeq (1 - \frac{\delta t}{\tau}) \end{aligned} \tag{1.25}$$

and Δ the sum of unit increases for each event that occurred between t and $t + \delta t$, given by

$$\begin{aligned}\Delta(\delta t) &= \sum_{k, t \leq t_k \leq t+\delta t} 1, \text{ with } \overline{\tau} \ll \delta t \text{ giving} \\ &\simeq \delta t \overline{n} \\ \Delta(\delta t) &\simeq \frac{\delta t}{\overline{\tau}}\end{aligned} \tag{1.26}$$

As activity reaches a steady state, we have

$$\begin{aligned}\mathcal{A}(t) &= \mathcal{A}(t + \delta t) \\ \mathcal{A}(t) &\simeq \mathcal{A}(t)(1 - \frac{\delta t}{\tau}) + \frac{\delta t}{\overline{\tau}} \\ \mathcal{A}(t)\frac{\delta t}{\tau} &\simeq \frac{\delta t}{\overline{\tau}} \\ \mathcal{A}(t) &\simeq \frac{\tau}{\overline{\tau}}\end{aligned} \tag{1.27}$$

Finally, injecting this result in 1.22, we obtain

$$\begin{cases} \boldsymbol{\delta v}_n &= \frac{\boldsymbol{v_c}}{\mathcal{A}(t_n)} \\ \delta\omega_n &= \frac{\omega_c}{\mathcal{A}(t_n)} \end{cases}. \tag{1.28}$$

This is the speed correction used in our implementation, allowing for smooth speed update during the convergence process. The activity $\mathcal{A}$ acts as a normalization value for all quantities updated event-by-event in this work.
Figure 1.4 illustrates how the velocities correction mechanism operates under the hood of a tracker.

a) Aperture metric

The local and incremental approach in this work is prone to the same aperture problem as the optical flow it relies on. To solve this problem, we introduce a metric that measures the quality of a tracked structure i.e. if it allows to estimate the correct velocity: a structure such as a single or a bundle of parallel lines is typically a bad candidate while intersecting lines or edges with finite curvature are good ones.
For clarity purposes, let us assume two local edges are captured by the tracker (the same analysis can be extended to n edges) and they are approximated as line segments with respective direction angles θ_1 and θ_2. We define $\Delta\theta$ as the difference of these angles that we constrain to take value in $[0, \frac{\pi}{2}]$ (this does not remove any generality to the problem since we can always choose the smallest intersecting angle of the two lines for $\Delta\theta$). The edges define a good structure to track if $|\Delta\theta|$ is sufficiently large (i.e. lines less parallel). According to this hypothesis, the metric we need must be a monotonic function of $\Delta\theta$.
Let us express the unit direction vectors of the two line segments in their complex form $e^{i\theta_1}$ and $e^{i\theta_2}$. If we want to constrain the metric function to take value in $[0, 1]$, we

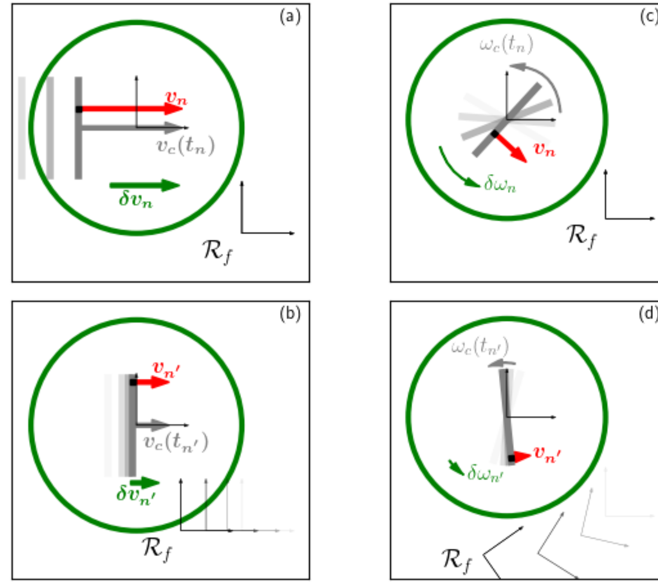

Figure 1.4 *Convergence process of a tracker onto a simple feature. We separated translation - $(a), (b)$ - and rotation - $(c), (d)$ - for clarity purposes. In practise, both are resolved in parallel. (a) and (c) : the trackers starts moving as the features appears. Each event is associated with a local velocity $\boldsymbol{v_n}$ in red (cf. Figure 1.3) and allows to estimate the contour velocities in translation $\boldsymbol{v_c}$ and in rotation ω_c in grey. The corrections applied $\boldsymbol{\delta v_n}$ and $\delta\omega_n$ follow equation 1.28. In both translation and rotation cases, when the tracker speed matches the contour velocities - (b) and (d) - , the contour appears immobile in the tracker reference frame.*

can increase the vectors arguments by a factor of 2. Let $\boldsymbol{r_1}$ and $\boldsymbol{r_2}$ be these new unit vectors:

$$\boldsymbol{r_1} = e^{i2\theta_1} \text{ and } \boldsymbol{r_2} = e^{i2\theta_2}. \tag{1.29}$$

Assuming $\theta_2 \geq \theta_1$, we define the two quantities: $\overline{\theta} = \theta_1 + \theta_2$ and $\Delta\theta = \theta_2 - \theta_1$. Then the averaged resultant of the two vectors is:

$$\begin{aligned}\boldsymbol{\mu} &= \frac{e^{i(\overline{\theta}-\Delta\theta)} + e^{i(\overline{\theta}+\Delta\theta)}}{2} = \frac{e^{i\overline{\theta}}(e^{-i\Delta\theta} + e^{i\Delta\theta})}{2} \\ &= \cos{(\Delta\theta)} e^{i\overline{\theta}}.\end{aligned} \tag{1.30}$$

From here, we can see that we can build the metric function based on the cosine in the amplitude of $\boldsymbol{\mu}$. It is monotonic for $\Delta\theta \in [0, \pi/2]$, decreasing from 1 to 0. Thus, if the tracker captures two parallel edges, $\Delta\theta = 0$ and the function is maximum and if the two edges are orthogonal, $\Delta\theta = \frac{\pi}{2}$ and the function reaches its minimum of 0.

To build the metric function which will also be updated for each incoming event assigned to a tracker, we proceed as follows:

- from the set $\mathcal{E}_n$ defined in section 1.2.3, build the offset vector $\boldsymbol{\delta x_k} = \boldsymbol{x_k} - \overline{\boldsymbol{x_k}}$ and increase its direction angle by 2 to produce $\widetilde{\boldsymbol{\delta x_k}}$.

  These vectors are averaged over $\mathcal{E}_n$ into a vector giving the principal orientation of the local edge:

$$\overline{\boldsymbol{r}}_{\boldsymbol{n}} = \frac{1}{N} \sum_{k \in \mathcal{E}_n} \widetilde{\boldsymbol{\delta x_k}}, \tag{1.31}$$

- then we compute the temporal decaying sum of the normalized vectors $\overline{\boldsymbol{r}}_{\boldsymbol{n}}$ and divide it by the tracker activity $\mathcal{A}(t_n)$:

$$\begin{cases} \boldsymbol{M}(t) & = \boldsymbol{M}(t_{n-1})\lambda_n + \frac{\overline{\boldsymbol{r}}_{\boldsymbol{n}}}{||\overline{\boldsymbol{r}}_{\boldsymbol{n}}||} \\ \boldsymbol{\mu}(t_n) & = \frac{\boldsymbol{M}(t_n)}{\mathcal{A}(t_n)} \end{cases} . \tag{1.32}$$

  The chosen metric is then $||\boldsymbol{\mu}(t)||$ which takes value in $[0, 1]$.

Structures subject to aperture ambiguities is filtered out by setting a threshold on $||\boldsymbol{\mu}||$ as good candidates to track yield small metric values.

b) Tracker Stabilization

Trackers are by default initialized with a null velocity. As events are registered, a tracker's motion parameters (velocity and position) are updated continuously, leading to a rapid convergence time. However, this continuous update might still report non null velocity errors and leaves us uncertain about the steady state of the trackers. We implement an additional estimator to lift this uncertainty based on the averaged norm of $\boldsymbol{x_n} - \overline{\boldsymbol{x_k}}$, as used in computing the optical flow in 1.6. As the tracking converges, the flow norm should converge to 0. We then build the ν function as follows:

$$\begin{cases} N(t_n) & = N(t_{n-1})\lambda_n + ||\boldsymbol{x_n} - \overline{\boldsymbol{x_k}}|| \\ \nu(t_n) & = \frac{N(t_n)}{\mathcal{A}(t_n)} \end{cases} \tag{1.33}$$

A threshold is set on ν (experimentally fixed to 0.5 pixel) below which we can state that a tracker has been stabilized. This value can be changed, but is not critical, as experiments have shown that as long as we stay within this order of magnitude, the described estimator is able to discriminate between a stabilization running and a converged state.

1.2.6 Time constant definition

So far, the mechanism of a tracker is built around the decay mechanism allowing it to be updated upon reception of an event and giving the tracker the ability to self-adjust to the scene dynamic. This decaying property is summarized by the time constant τ in the exponential function, hence the estimation of τ is one (if not the most), of the most important tasks to achieve.
To estimate τ from the events, we are first considering the average squared velocity of points over the tracker area:

$$\overline{||\boldsymbol{v_n}||^2} = ||\boldsymbol{v}||^2 + 2\omega \boldsymbol{v}.\overline{\boldsymbol{o_n}} + \omega^2 \overline{||\boldsymbol{o_n}||^2}, \tag{1.34}$$

where the averaged terms $< \boldsymbol{o_n} >$ and $< ||\boldsymbol{o_n}||^2 >$ are updated according to the mechanism:

$$\begin{cases} \boldsymbol{O}(t_n) & = \boldsymbol{O}(t_{n-1})\lambda_n + \boldsymbol{o_n} \\ R_{sq}(t_n) & = R_{sq}(t_{n-1})\lambda_n + ||\boldsymbol{o_n}||^2 \\ \overline{\boldsymbol{o_n}} & = \frac{O(t_n)}{\mathcal{A}_n} \\ \overline{||\boldsymbol{o_n}||^2} & = \frac{R_{sq}(t_n)}{\mathcal{A}_n} \end{cases} \tag{1.35}$$

Finally, we can define τ as

$$\tau(t_n) = \frac{1}{\sqrt{\overline{||\boldsymbol{v_n}||^2}}} \tag{1.36}$$

1.2.7 Spatial descriptor and tracker lock

For the entire tracking process, a rolling set of events is maintained from the events affecting the tracker. At time t, we only keep events that appeared between $t - N\tau(t)$ and t. Events from set $\mathcal{E}_n$ are extracted from this spatial descriptor, stored as a $2 \times m$ matrix.

$$\boldsymbol{d} = \begin{bmatrix} \boldsymbol{x_1}^T \\ \vdots \\ \boldsymbol{x_k}^T \\ \vdots \\ \boldsymbol{x_m}^T \end{bmatrix} \tag{1.37}$$

This local descriptor is calculated in a continuous space i.e. the coordinates are real values instead of integers. This allows for a much smoother spatial representation of the object, often enhancing its visual appearance from the raw events, and can thus be used in following algorithms based on that tracking, such as feature matching, tracking recovery or object recognition.

a) Locking the feature

As the spatial descriptor so far is a rolling set, even with a tracker that has converged onto a trackable feature, a drift can still occur. To avoid that, we can *lock* the tracker. With the tracker T going into the *locked* status at time t_l, we lock that spatial descriptor, and the new events are compared to the events stored in d at time t_l. This shape is then the reference shape, and the tracking process can remain stable for much longer times.

b) Disengaging the feature

One of the challenges of the approach is caused by sudden deceleration of tracked objects. As less events are produced during a deceleration the correction processes must be robust to rapid changes in dynamics. The solution lays in monitoring the activity $\mathcal{A}$ of the *locked* tracker. The idea being that below a certain threshold, one must *disengage* the tracker. We roll back its position $\boldsymbol{X}$ and angle θ to the last known values where a viable optical flow has been computed, and stop updating the position relative to the speed as described

in 1.1. At the same time, we decay the speed itself, as only a rapid deceleration can induce such a behaviour :

$$\boldsymbol{v}(t_n) = \boldsymbol{v}(t_{n-1})\lambda_n \tag{1.38}$$

In most cases, this process provides sufficient time for the object to move again on the focal plane, and the tracking to resume, once the activity $\mathcal{A}$ is above a predefined threshold.

1.2.8 Illustration case

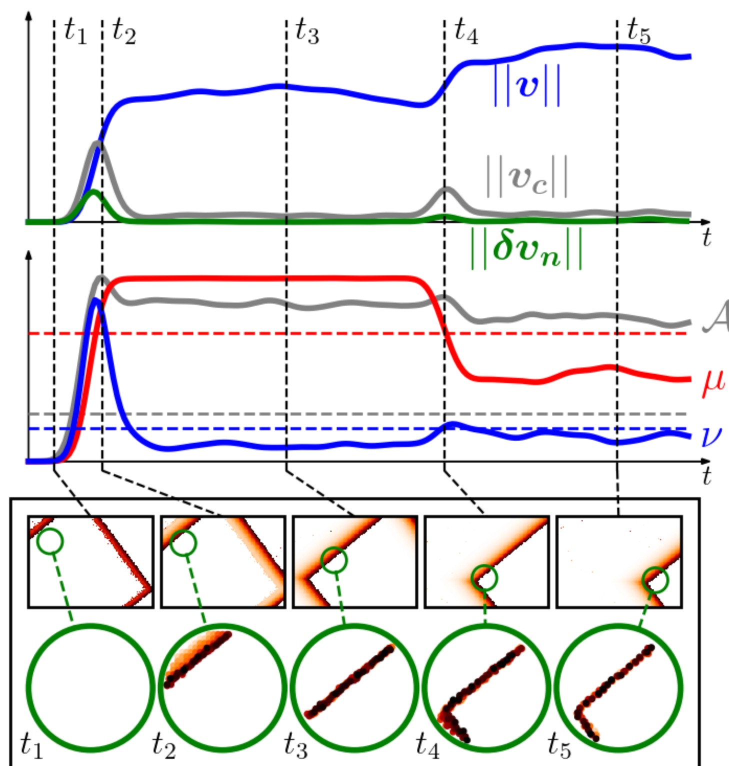

Figure 1.5 *Evolution of the main variables during the tracker convergence on a corner in translation, described in detail in section 1.2.8. Top graphs shows the velocity variable $\boldsymbol{v}$ of the tracker, the estimated contour velocity $\boldsymbol{v_c}$ w.r.t the tracker, and the updates for each event $\boldsymbol{\delta v_n}$. For clarity purposes, we use the norm for each of these variables. The second graph shows the evolution of the activity $\mathcal{A}$, the aperture metric $\mu = ||\boldsymbol{\mu}||$ and the convergence metric ν. Square pictures show the spatio-temporal context of the scene, while green circles contain the projected events, i.e the events from the tracker perspective. Key moments are highlighted with timestamps t_1 to t_5.*

The main parameters of a tracker are highlighted in a practical case displayed in Figure 1.5. This real-data example shows a square moving in a pure translation movement at a constant speed of about $1000 px.s^{-1}$. We show the temporal evolution of the six main

parameters of a tracker. First, the norms of the tracker velocity $||\boldsymbol{v}||$, the norm of the contour velocity w.r.t the tracker $||\boldsymbol{v_c}||$ and the norm of the velocity variations for each event $||\boldsymbol{\delta v_n}||$. In a second graph are shown the activity $\mathcal{A}$, the aperture metric $\mu = ||\boldsymbol{\mu}||$ and the convergence metric ν.
The sequence is broken into five key moments, each reflecting a different important phase of the process. At $t = t_1$, no event have occurred in the tracker region of interest (ROI) since its initialization. All tracker's parameters are either null or undefined, and the tracker is still. For $t = t_2$, the convergence process starts after the activity reaches a predefined threshold. The tracker then starts to cancel out any contour velocity. This results in a increase of the convergence metric μ. The aperture metric already shows that the structure captured by the tracker is a straight line, as it reaches a value close to 1. For clarity purposes, we deactivated the mechanism to reject tracking based on the aperture metric.
At $t = t_3$, the tracker reaches a steady state. All values remain almost constant, the tracker knows it has converged (ν is low) but the aperture issue remains. Only the orthogonal component of the velocity is known to be reliable, and the tracker is free to move along that line[1]. During this phase, the contour velocity $\boldsymbol{v_c}$ - and thus the correction applied $\boldsymbol{\delta v_n}$ - are almost null.
Around $t = t_4$, a corner appears within the tracker ROI. Immediately, the aperture metric drops. The other corner edge feeds new velocity information to the tracker, resulting in an increase of the contour velocity $\boldsymbol{v_c}$ as well as the convergence metric ν. The tracker compensates this velocity to cancel out this remaining velocity. One can notice a small increase of the activity. This is due to an increase of the number of events assigned to the tracker when the corner appears, while the velocity - and thus the time constant- does not immediately change.
Finally, at $t = t_5$, once the correction has been applied, the tracker speed matches the corner speed, and the features appears immobile from the tracker perspective (in the tracker reference system $\mathcal{R}_T$). Both the aperture and the convergence metrics are below their predefined threshold, and the shape is *locked* as described in section 1.2.7.a. This increases even more the stability of the tracking. As the tracker velocity increases, its time constant τ lowers, slightly lowering the activity, and improving the time response of the whole system to match the scene dynamics.
The tracker has been intentionally initialized far from its target feature resulting in a $100ms$ convergence time. When trackers were initialized closer to the target for the very same example, the convergence time was on average less than $10ms$.

1.2.9 Algorithms

The method presented so far is implemented as summarized by the algorithm detailed in this section. We can decompose the tracking into three blocks. All tracker variables previously described are now indexed by i:

[1] In most cases, the tracker has a null velocity component in the direction of the line. In our case, this induces a negative velocity along the y axis, and the tracker will move towards the bottom of the focal plane

Algorithm 1 Main Algorithm

Require: N_T trackers initialized in the focal plane.
1: **for** each $e_n = \{\boldsymbol{X_n}, t_n\}$ **do**
2:     **for** Tracker $T_i, i \in [1, N_T]$ **do**
3:         Update T_i according to algorithm 2
4:         **if** $||\boldsymbol{X_n} - \boldsymbol{X_i}|| < R$ **then**
5:             Compute $\boldsymbol{x_n}$ according to eq. 1.2
6:             Compute velocity error according algorithm 3

Algorithm 2 Update for each event

Require: Tracker T_i, event $e_n = \{\boldsymbol{X_n}, t_n\}$.
1: Compute time elapsed since last event Δt_n
2: Update tracker position :
3: $\boldsymbol{X_i}(t_n) \leftarrow \boldsymbol{X_i}(t_{n-1}) + (t_n - t_{n-1})\boldsymbol{v_i}(t_{n-1})$
4: $\theta_i(t_n) \leftarrow \theta_i(t_{n-1}) + (t_n - t_{n-1})\omega_i(t_{n-1})$.
5: **if** $\boldsymbol{X_i}$ is out of screen **then**
6:     Reset tracker T_i
7:     $T_i \leftarrow$ *Idle*
8:     **return**
9: Compute Exponential decay:
10: $\lambda_i(t_n) \leftarrow \exp\left(-\frac{t_n - t_{n-1}}{\tau_i(t_{n-1})}\right)$
11: Update tracker monitoring variables with $\lambda_i(t_n)$:
12: $\mathcal{A}_i(t_n) \leftarrow \lambda_i(t_n)\mathcal{A}_i(t_{n-1})$.
13: **if** $\mathcal{A}_i(t_n) < \mathcal{A}_{reset}$ **then**
14:     Reset tracker T_i
15:     $T_i \leftarrow$ *Idle*
16:     **return**
17: **else if** $\mathcal{A}_i(t_n) < \mathcal{A}_{disengage}$ **then**
18:     $T_i \leftarrow$ *Disengaged*
19:     Reset position and angle to last valid values
20: **else if** $T_i =$ *Disengaged* & $\mathcal{A}_i(t_n) \geq \mathcal{A}_{disengage}$ **then**
21:     $T_i \leftarrow$ *Locked*

We define for each tracker a set of important states that are used in the implementation[2]. Those states are:

- *Idle*: a tracker is considered *Idle* as long as the amount of events assigned to the tracker is under the experimentally predefined threshold.
- *Stabilizing*: it is considered *Stabilized* when computation is being performed but the convergence estimator is still too high. This means that the tracker still needs time to catch up with the target feature.

[2]The algorithm implementation is accessible on `https://github.com/neuromorphic-paris/eb-tracker/`.

Algorithm 3 Update for events affecting the tracker

Require: Tracker T_i, event $e_n = \{\boldsymbol{X_n}, t_n\}$.
1: $\mathcal{A}_i(t_n) \leftarrow \mathcal{A}_i(t_{n-1}) + 1$
2: **if** $T_i = Idle$ & $\mathcal{A}_i(t_n) > \mathcal{A}_{Start}$ **then**
3: $T_i \leftarrow Stabilizing$
4: Compute local flow, cf equation 1.6
5: Update dynamical estimator, cf equations 1.9 to 1.13
6: Compute contour velocity, cf equation 1.17
7: Compute speed correction, cf equation 1.28
8: Update aperture estimator, cf equation 1.32
9: Update convergence estimator, cf equation 1.33
10: Update tracker state, cf Figure 1.6
11: Update time constant, cf equation 1.36

- *Converged*: a tracker has *Converged* once it has stabilized and matched the target feature.
- *Locked*: this state maintains the tracker on tracking a structure for which the algorithm detects no aperture problem i.e. it is valid only when *Aperture_issue* is "false".
- *Aperture_issue*: a tracker has *Aperture_issue*="true" if it detects an aperture problem from the aperture metric. This status prevents the tracker from being *Locked* to the target even if it is in a *Converged* state.
- *Disengaged*: A tracker has *disengaged*="true" when it is in a *locked* state but its activity is too low. The position is no longer updated using the speed, and the speed is decayed, until new events appear again.

Those states are shown in Figure 1.6, with their corresponding relations and the parameters required to update those states.
Table 1.1 shows the experimentally predefined values for each state threshold that were used in all experiments. Note that some of these values may depend linearly on the radius R of the tracker. This radius must especially be changed accordingly to the sensor resolution.

Name	Value	Proportional to R
R	10	Yes
$\mathcal{A}_{start}$	40	Yes
$\mathcal{A}_{disengage}$	10	Yes
$\mathcal{A}_{reset}$	4	Yes
$\kappa_{aperture}$	0.55	No
κ_{conv}	0.6	No

Table 1.1 *Experimental values for the hyperparameters presented. Most of them are either constant across all experiment, or scale linearly with the radius R.*

At this stage, the proposed algorithm is able to track structures that are not restricted to corners or simple edges and can be somehow complex as shows the sample given by Figure 1.7. These features were extracted during a run of our algorithm on a recording of a city environment. This unconstrained structure selection allows to have a wide range of of features that are crucial to higher level tasks.

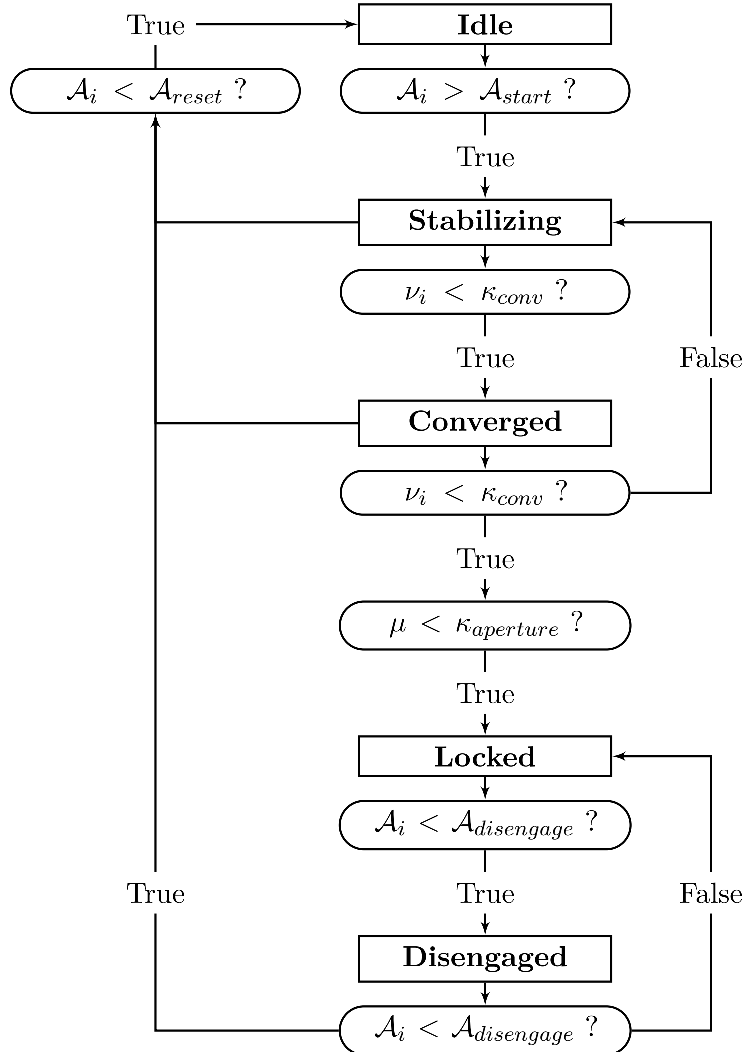

Figure 1.6 *Relations between the different features' states. We show here the key conditions for the evolution of a tracker, from its initialization in an Idle state to its reset. Conditions leaving the tracker state unchanged are not shown here for clarity purposes.*

1.3 Experiments and performances evaluation

We compare the algorithm performances to the state-of-the-art. As shown in section 1.2, the tracking and detection mechanisms are interlaced such that features emerge in an online manner. This is a purposely designed property because users do not have to assume complex priors on the features to track. We tested the algorithm on sequences generated by the event-based camera simulator, alongside with the provided dataset, acquired by event-based sensors in natural environments [88]. To assess the algorithm performances, the provided frames are used and a classical frame-based tracking algorithm is initialized when a tracker is *locked*. The same dataset provides ground-truth from an Inertial Measurement Unit (IMU), allowing us to assess directly the camera's ego-motion inferred from the trackers estimated velocities. The data is acquired

using a DAVIS 240C, with a resolution of 240 × 180 pixels [57]. Preliminary tests have also been performed with a 640 × 480 VGA ATIS [89]. Similar results have been found using both sensors relying on the same experimental set of parameters. The algorithm is bench-marked using a python implementation. We initialized 63 trackers dispatched on a 9 × 7 grid, with 25 pixels diameter ROI.

1.3.1 Comparison with Frame-based tracking

When frames are available, we can can compare the event-based tracker with a frame-based algorithm. For this purpose, we have used the frame-based Channel and Spatial Reliability Tracking (CSRT), an OpenCV implementation of [90] as the baseline. Each event-based tracker, on locking, is compared to a CSRT tracker initialized at the corresponding location on the temporally closest frame. We compare the two trackings in stages where the event-based tracker is in *locked* status.
The CSRT results are visually checked for the entire duration of the benchmark. Event if the CSRT is considered as one of the state-of-the-art frame-based tracking algorithm, it does occasionally fail. In order to have the most accurate benchmark assessment for the proposed algorithm, we manually removed the failing CSRT trackers scenarios although the presented event-based tracker has been performing correctly.
For certain highly textured scenes the frame-based tracker was also unable to provide any ground-truth data due to excessive displacement between two consecutive frames and also because of motion blur. However, this method allowed to automatize the benchmark and to avoid a manual labeling of considered scenes.

1.3.2 Benchmark with the Event Camera Simulator

We used data generated with the Event Camera Simulator [88] for which the ground-truth can be inferred from a known camera motion and scene depth. Simple movements mixing translation and axial rotation have been simulated, along with the corresponding frames. For each tracker *locked*, we determine the $3D$ location of the object and re-project its location on the focal plane. We generated two sequences. The first one uses the shapes poster from the Event-Camera dataset, while the second one uses the texture of a book cover. The event-based and frame=based algorithms use similar window sizes: a $25px$ ROI for both the event-based method and the CSRT trackers. The trackers accuracy and life-span are reported in Table 1.2.

Sequence	Valid Tracks		Asserted Error (px.)		Asserted Lifespan (s)	
	this method	CSRT	this method	CSRT	this method	CSRT
shapes (short)	93.0%	76.3%	1.40	2.29	1.06	1.06
shapes (long)	89.9%	92.8%	1.36	1.90	1.57	1.71
book (short)	92.6%	60.1%	1.28	2.02	0.68	0.60
book (long)	91.5%	81.4%	1.05	2.52	1.02	1.09

Table 1.2 *Comparison between the frame-based tracking and the event-based tracker using simulated data. CSRT trackers are removed once the event-based tracker has lost the target due to the disappearance of the target from the scene.*

1.3.3 Benchmark on event camera dataset

The performance comparison is now applied the same sequences of the dataset that were reported in previous work in [56]. Since no ground-truth is available, the CSRT trackers were used as a

ground-truth for comparison. We report in Table 1.3 the results obtained for six sequences. A track is considered *valid* if its mean error to the ground-truth is less than 5 pixels. We also display the resulting mean error of those valid tracks and their average lifespan. Most of the scene's objects come in and out of the field of view of the camera, tracker that go out of the focal plane are considered lost. This explains the average lifespan of the tracker trackers. Re-entry strategy are beyond the scope of the paper as they imply storing previously seen trackers. Results show lifespans of several seconds with accurate tracking.

Stream	Duration	Valid Tracks	Error [px]	Lifespan
shapes_rotation	30s	71.9%	1.90	0.60s
shapes_translation	30s	93.7%	1.66	0.84s
shapes_6dof	30s	86.8%	1.90	0.61s
poster_rotation	11s	69.1%	1.85	0.69s
poster_translation	14s	68.1%	2.12	0.88s
poster_6dof	20s	62.1%	2.23	0.49s

Table 1.3 *Results on GT comparison with CSRT trackers in OpenCV library.*

1.3.4 Panoramic reconstruction

Using a set of trackers we will now compute a panoramic reconstruction of an outdoors scene using a recorded sequence of 6.6s. Each *green* rectangle represents a tracker used for that reconstruction. We display on that panoramic view the most representative set of trackers that were *locked* and used for that reconstructions (*blue dots*) with some of them highlighted in colored circles for clarity purposes, and shown in more details below the panoramic view.

This selection shows the wide variety of features the trackers can lock on. Below those features are displayed the average reconstruction error ϵ and the duration Δ_t during which the tracker stayed *locked* on its feature. We notice that the longer duration of tracking reported concerned mainly highly ”complex” features. The top right feature is displayed here to show the limitation of the algorithm under textured and noised recordings : the trackers can sometimes *lock* onto ill-defined shapes, resulting in poor tracking duration. However, in these cases, the convergence estimator previously described and the tracker activity monitoring are able to cancel this tracker, noticing that it does not “behave" as it should.

For this online robust reconstruction, we used 471 trackers, with an average tracking time of 0.62s, and an average reconstruction error of 3.3 pixels considering all trackers. For the non outliers the median standard deviation of the reconstruction error is 0.72 pixels. This means that 72.7% of the trackers that were selected could successfully track features with an error of less than 1 pixels. Of those 471 trackers, 27.3% outliers reached the reconstruction error threshold of 20 pixels. In most of the cases, this was due to "blurred" straight lines that can sometimes report no aperture issue when small rapid saccadic motions occur.

1.3.5 Comparison with groundtruth with IMU data

The Event-Camera Dataset [88] provides IMU and camera pose data, serving as ground-truth measurements. This information is particularly relevant for the 3 degree of fx§reedom ego-motion

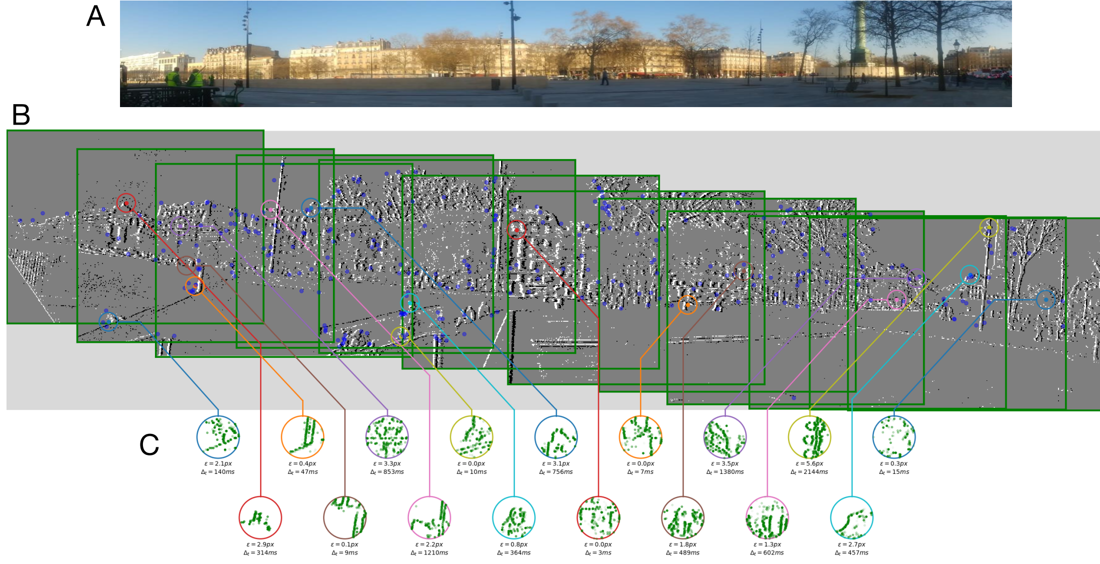

Figure 1.7 *A Panorama reconstruction of a natural outdoors scene using a hand-held event-based camera (shown in A using a conventional frame-camera for clarity purposes). In B, the constructed panorama from events emphasizing a subset of the features used for clarity. Some highlighted features are represented with colored circles to show the diversity of what can be tracked (shown in C) with their average reconstruction error ϵ and the duration Δ_t during which a tracker stayed locked on its feature. Green squares shown in B, show some of the focal plane locations used to construct the panorama.*

of the camera. Assuming $\boldsymbol{M} = [X, Y, Z]^T$, $\boldsymbol{m} = [x, y, z]^T$ being a 3D point and its projection into the focal plane in the camera coordinate frame, the instantaneous velocities are given by:

$$\begin{cases} \dot{\boldsymbol{M}} & = \boldsymbol{\omega} \times \boldsymbol{M} + \boldsymbol{t}, \\ \dot{\boldsymbol{m}} & = \frac{d(\frac{f}{Z}\boldsymbol{M})}{dt} = \frac{f}{Z}\dot{\boldsymbol{M}} - \frac{\dot{Z}}{Z}\boldsymbol{m} \end{cases} \tag{1.39}$$

where $\boldsymbol{\omega}$ and $\boldsymbol{t}$ are respectively the angular and the linear velocities of the camera. Since the velocity in the z axis of $\boldsymbol{m}$ is null, this is equivalent to:

$$\begin{pmatrix} \dot{x} \\ \dot{y} \end{pmatrix} = \begin{pmatrix} -\frac{xy}{f} & f + \frac{x^2}{f} & -y & \frac{f}{Z} & 0 & \frac{x}{Z} \\ -\frac{y^2}{f} - f & \frac{xy}{f} & x & 0 & \frac{f}{Z} & -\frac{y}{Z} \end{pmatrix} \begin{pmatrix} \omega_x \\ \omega_y \\ \omega_z \\ t_x \\ t_y \\ t_z \end{pmatrix} \tag{1.40}$$

This flow is related to the optical flow calculated in the focal plane via the upper left 2×2 submatrix k, taken from the intrinsic matrix of the camera:

$$k^{-1} \begin{pmatrix} \dot{u} \\ \dot{v} \end{pmatrix} = \begin{pmatrix} \dot{x} \\ \dot{y} \end{pmatrix}. \tag{1.41}$$

Equation 1.40 can be solved for $\boldsymbol{\omega}$ knowing only the calibration parameters and if the linear translation in the ego-motion is null. We are focusing on that particular case because to solve for the complete 3D velocity, one needs to provide a set of $\boldsymbol{M}$ and their projection $\boldsymbol{m}$ and this is not available in the Event-based Camera Dataset. We show the estimated ego-motion results for the sequence `shapes_rotation` in Figure 1.8. The axis have been normalized, assessing for x and y rotations a geometric constant of 220 pixels.rad^{-1}. For the z-axis, the normalization constant found was 0.98, showing the good match between this ego-motion computation and the ground-truth provided.

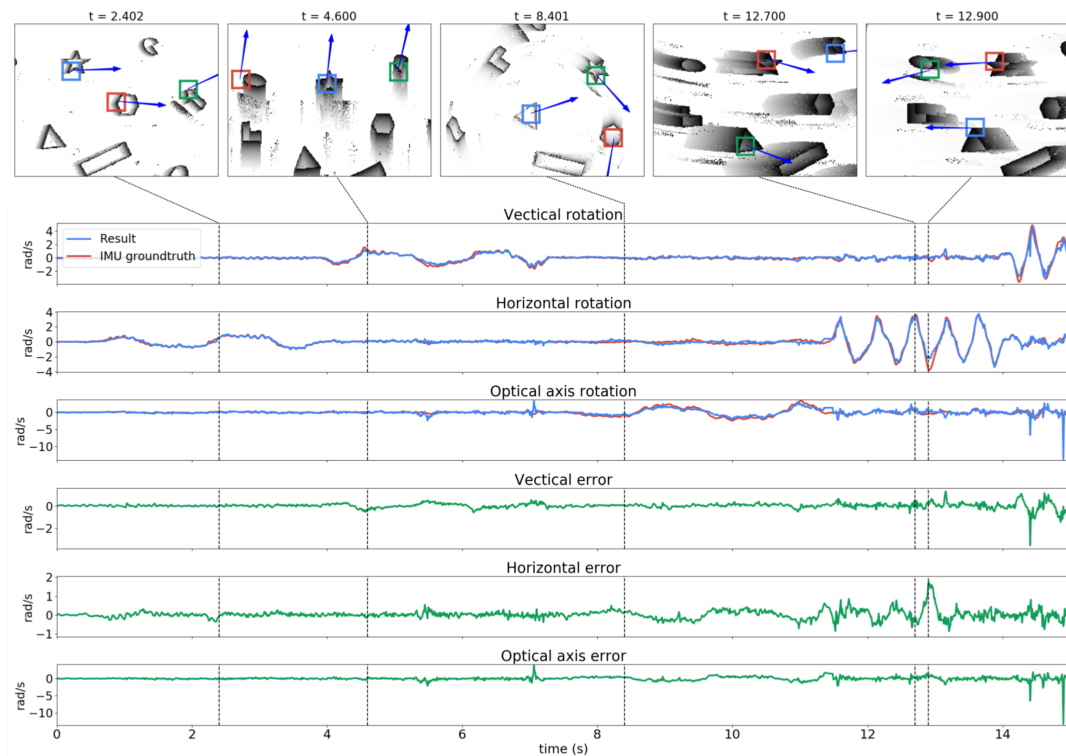

Figure 1.8 *Comparison of tracking results for shapes_rotation camera's estimated angular velocity. Trackers velocities are averaged to extract the position and speed of an initially centered point, assuming the objects on the scene are static and only the camera moves. Red lines: Gyroscopic data from the IMU provided with the DAVIS dataset [88]. Blue lines: Estimated rotational speed for each axis of the gyroscope. Green lines: Error between the reconstructed rotation and the IMU ground-truth. Top graph: Vertical rotation of the camera, producing an horizontal speed of the objects. Mean error :* 0.04rad/s*. Center graph: Horizontal rotation of the camera, producing a vertical speed of the objects. Mean error :* 0.005rad/s*. Bottom graph: Rotation of the camera along the optical axis. Mean error :* 0.01rad/s*. Top images: Snapshots from the corresponding scenes, with a restricted sample of the tracked features and their respective velocities orientations.*

1.3.6 Trackers stability and latency

By design, the tracker time constant τ is auto-adjusted during the tracking w.r.t. the scene dynamics and it is the only parameter that needs explicit initialization as the inverse of a minimal speed, experimentally chosen as about 5 pixels.s^{-1}. The inertial behavior of a tracker is implemented to ensure a smooth and a robust to noise tracking mechanism. A tracker loses its targets because it is getting out of the camera's field of view, rather than because of drifts caused by a non robust tracking approach.

The compromise to the tracker stability is the delay caused by the inertia. This is a non desired effect for a low latency vision sensor or for any vision sensor's in general. However, the velocity based tracking algorithm we introduce has an interesting property that we can observe experimentally: on average, a tracker converges to the correct location within an radius of 20 to 30 pixels, when a structure enters the receptive field of the tracker.

This typical constant offset for the tracker to converge is an important asset as the time to convergence is a decreasing function of the speed i.e. the faster the apparent motion, the faster the tracker converges, hence the lower the latency of the algorithm.

1.4 Conclusion

This work introduced an event-based method to select and track simultaneously local spatial structures from a stream of events output form an event-based camera in real-time. The presented method relies on a velocity based tracking approach to converges towards selected features from which the true velocity can be estimated. More important, it is designed to be resilient to the aperture problem. The algorithm is also designed in a way to reduce the number of parameters to be set by users: mainly a decay coefficient that represents a dimensionless inertia parameter. The method is expected to work even better for higher spatial resolutions, as higher spatial resolution combined with the current sub-millisecond temporal precision implies that velocity can be estimated more accurately.

The method is showing robustness in a wide range of conditions. It can detect and track a large variety of spatial structures at different velocities. Tracking benchmarks have shown that it is accurate while being updated event by event, at the native rate driven by the scene thus allowing to reduce resources requirements. This has been made possible by estimating reliable velocities for each tracked feature. Most importantly, using several projection velocities for initialization allowed to have very little dependency on the tuning parameters.

From a larger perspective, this paper introduces a new canonical general scheme for computation using event-based cameras that allows for an independence from motion direction and velocity changes. The introduction of a simultaneous use of space and precise timing information brought by each incoming event with a combined regularization scheme using the local activities of events proved to be efficient for local computations. The same scheme can be applied to a wide variety of problems where the same approach could be applied efficiently.

Chapter 2

Stereo Matching and Visual Ego Kinetics

2.1 Introduction

In this chapter, we propose to tackle the odometry problem from a totally new perspective which focuses on recovering the event-based vision sensors' "ego-kinetic" as the physical quantities directly estimated are relative to the sensors velocities. We are motivated by an approach where the processing is inducing little latency w.r.t. to the fast perception of the event-based sensors. The stereo vision is chosen to provide instantaneous depth estimation as opposed to monocular visual odometry where the sensor has to move significantly before depth can be reliably estimated. This approach therefor also encompasses the monocular sensing used to solve the same problem at a longer time scale. In the case of a stereo configuration, depth is instantaneously estimated, ensuring a minimal latency in the processing pipeline by building encoding with time-surfaces different level of abstraction in visual information.
To maintain a low latency processing pipeline, we are relying on a hierarchical structure of computation blocks. Three different block, presented independently, respectively compute the optical flow, objects depth from stereo-matching and ego-kinetics. The later makes use of the two first block extracted data, however other depth estimation or optical flow methods could be used as inputs. However, the fully event-based formulation of each of those block allows to gain access to each component with a low latency, reduced computational cost, while achieving state-of-the-art results.

The optical flow computation presented in this chapter performs the gradient of a time surface, formulated in a fully event-based fashion. Gradient operations are thus minimized, and a second step is added after data retrieval for outliers removal. This ensure both a low-latency process, along with strong robustness to data noise.

Depth estimation from event-based stereo data is the preferred approach as it provide instantaneous depth information. Its only requirement is a prior geometrical calibration of each sensor and their precise synchronization. From the calibration, an epipolar rectification is also applied so each row of pixels in a sensor is coinciding with an epipolar line. The core idea in depth estimation from event-based stereo is not fundamentally different from classical vision stereo. It is a two step problem: first the matching of events along the epipolar lines follows by the depth estimation from the stereo matched pair disparities. This is also equivalent to the triangulation operation if the

focal length of the lens used in the sensor is known. The main difference in event-based stereo techniques is the absence of luminance information. The matching support has to be built in order to use spatial and temporal consistencies to match two events. To bypass the absence of image, we build epipolar time-surfaces as it is explained in section 2.2.2.

Up to our knowledge, there is in the literature no reported work on direct instantaneous velocity estimation from passive vision sensors. Event-based vision sensors, with their microsecond latency, are however the perfect devices to allow such a task. This has been shown in several works from the field of event-based vision sensor, to recover sensor's rotational motions. In short, the rotational motion estimation consists in estimating the angular velocity. These works originally aimed to generate panoramic view from the events [91, 92] while in [93], it is a preliminary step toward a contrast maximization framework to produce motion blur free event-images.
This work is the first of his kind in focusing on the direct and instantaneous estimation of the sensor's velocity which encompasses both translation and angular velocities. Therefore, instead of referring to ego-motion estimation, we are rather estimating sensor's ego-kinematic. We describe as such our approach as an Ego-Kinetics Estimation (EKE).

2.2 Method

The strategy to maintain low latency all along the processing is to rely the an extensive use of Time-surfaces for an event-by-event processing strategy. To reach this objective, we develop a modular architecture based on independent building blocks that process events as input and then output also event with a higher level of abstraction. For example, the optical flow building block takes a raw event from a sensor as input and output the corresponding 2D velocity vector. The same goes for stereo-matching block that however requires events from both vision sensors and that outputs for each of them a depth event. The last standalone block is the EKE and is at this point of processing specific to each sensor. It requires flow and depth events to estimate the sensor's velocity event. Fig.2.1 provides an illustration summarizing the event-based end-to-end processing pipeline.

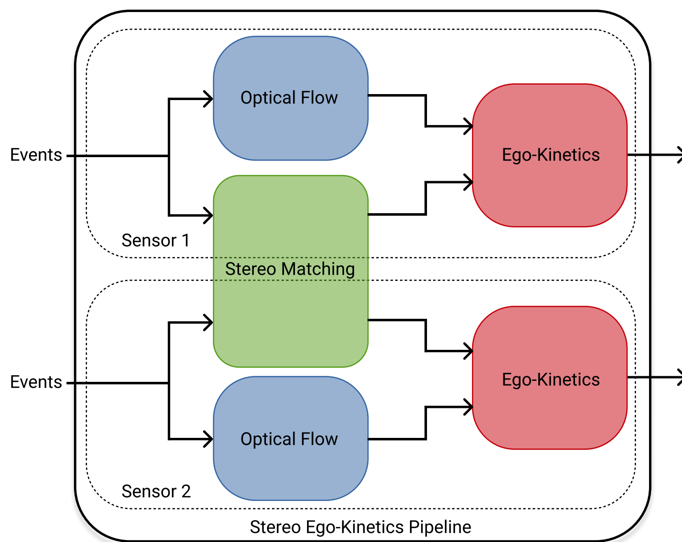

Figure 2.1 *Overview of the entire pipeline described in this work. Each block is a standalone computation block, detailed and experimented on. The EKE block requires an amplitude-precise optical flow, as well as 3D data of the scene. Each block in developed in a true event-based manner, using the full potential of neuromorphic sensors.*

2.2.1 Gradient-Based Optical flow

The optical flow block is a key component in the velocity estimation as the measured flow is the direct projection of the sensor's velocity into the focal plane. A low latency event-based optical flow algorithm is required. Therefore, plane fitting techniques such as [52] are favored. Aperture robust variant such as [66] could be the ideal choice if it was not adding latency with the multi-scale approach to correct the estimated flow.
As shown in [52], the plane fitting technique is equivalent in estimating the gradient of the time-surface T, that stores at location (x_n, y_n) the time t_n of occurrence of the event e_n:

$$\begin{array}{rcl} T : \mathbb{N}^2 & \rightarrow & \mathbb{R} \\ (x_n, y_n)^T & \mapsto & T(x_n, y_n) = t_n \end{array} . \tag{2.1}$$

The optical flow for each event is obtained by the inversion of the norm of the gradient given by the fitted plane parameters. As the minimization scheme used in the plane fitting does not have a

outlier rejection mechanism, we propose an improved event-based optical flow computation based on a direct computation of the gradient of T, followed by a non linear outlier rejection mechanism to remove spurious events. Incoming sections provide details on the algorithm and results from experiments will show the performance of the flow compared to the plane fitting algorithm.

a) Time and Gradient Maps

As we are intensively relying on the use of the concept of time-surfaces, the gradient of T is computed with a locale finite difference approach of T, at each event e_n and results are stored into two matrices ΔT_x:

$$\begin{aligned} \Delta T_x(x_n-1,y_n) &= t_n - T(x_n-1,y_n) \\ \Delta T_x(x_n,y_n) &= T(x_n+1,y_n) - t_n \end{aligned} \tag{2.2}$$

and ΔT_y:

$$\begin{aligned} \Delta T_y(x_n,y_n-1) &= t_n - T(x_n,y_n-1) \\ \Delta T_y(x_n,y_n) &= T(x_n,y_n+1) - t_n. \end{aligned} \tag{2.3}$$

We also used build two intermediary matrices to accelerate the locale data update and retrieval. T_x to store the neighbors' times in the x-axis:

$$\begin{aligned} T_x(x_n-1,y_n) &= T(x_n-1,y_n) \\ T_x(x_n,y_n) &= T(x_n+1,y_n), \end{aligned} \tag{2.4}$$

and T_y, the neighbors' times in the y-axis:

$$\begin{aligned} T_y(x_n,y_n-1) &= T(x_n,y_n-1) \\ T_y(x_n,y_n) &= T(x_n,y_n+1). \end{aligned} \tag{2.5}$$

This allows to build a complete representation of the temporal gradient on both axes, and to achieve gradient computation only once for each event.

b) Outliers suppression

From the previously defined "time-surfaces" ΔTx, and ΔT_y, we have a coarse event-based optical flow. In this section, we add to the pipeline an outlier removal mechanism to improve the flow estimation.

Let us define the parameters R and τ for the size of the spatio-temporal neighborhood centered at the current event e_n. We define the set S_x such as:

$$S_x(n) = \left\{ \Delta T_x(x,y), \left| \begin{array}{l} |x-x_n| \le R, \\ |y-y_n| \le R \text{ and} \\ t_n \ge T_x(x,y) \ge t_n - \tau \end{array} \right. \right\}, \tag{2.6}$$

and from $S_x(n)$, let us define $\tilde{\delta}_x(n)$ as its median value. Any element $\delta_k \in S_x(n)$ is discarded if it satisfies:

$$|\delta_k - \tilde{\delta}_x(n)| > \rho \sqrt{\sum_{k=1}^{N} (\delta_k - \tilde{\delta}_x(n))^2}, \tag{2.7}$$

where $N = card(S_x(n))$. This outlier suppression is iterated with the recomputation of $\tilde{\delta}_x(n)$ and N, until no element in $S_x(n)$ satisfies eq.2.7.

We proceed in a similar way for the y-axis to get $\tilde{\delta}_y$. Finally, the optical flow at (x_n, y_n) is given by:

$$\boldsymbol{f}_n = \left(\frac{\tilde{\delta}_x(n)}{\tilde{\delta}_x(n)^2+\tilde{\delta}_y(n)^2}, \quad \frac{\tilde{\delta}_y(n)}{\tilde{\delta}_x(n)^2+\tilde{\delta}_y(n)^2}\right)^T \tag{2.8}$$

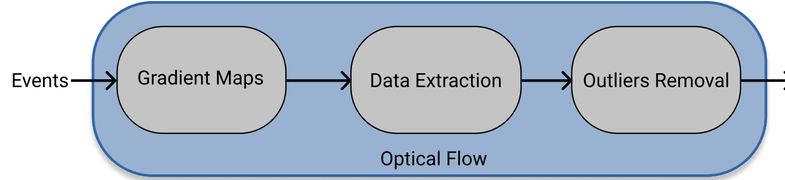

Figure 2.2 *Overview of the optical flow pipeline used within this work. The gradient maps, detailed in Section 2.2.1.a, allows to store the necessary data, and perform the time differentiation computation only once.*

The standalone optical flow block is summed up in Figure 2.2, with the three main steps described in this section. A standalone benchmark of this optical flow method is reported in section 2.3.1, comparing it to the more traditional plane fitting method.

2.2.2 Event-based Stereo Matching

The stereo matching requires as input, the synchronized streams of events from the two sensors and outputs the depth estimation for each of them. It is the only part of the pipeline where the two streams are dependant. Once the depth is known, each sensor can again be handled independently and the depth information is fed into the EKE block at the same rate as the optical flow block. This is ensured again by maintaining a time-surface that updates and stores the estimated depth (when it can be) for each event. In this section, we are building several form of epipolar time-surfaces with each of them storing different information needed for the stereo matching task.

a) Epipolar Time Map

The epipolar time-surface serves the same purpose as a generic time-surface which is to keep a short history of the depth/disparity information inferred from the stereo sensor system that reflects the scene dynamics. It assumes the epipolar-rectified hypothesis of the sensors and each new event e_n is updating the neighbors locations at column x_n, from row $y_n - R_y$ to $y_n + R_y$. R_y is the radius for which an event is considered to have a impact to its neighbors.
More formally, we first define

$$I_n = \left\{ (x, y) \middle| \begin{array}{l} x = x_n \\ |y - y_n| \leq R_y \end{array} \right\}, \tag{2.9}$$

the set of pixels within the neighborhood of (x_n, y_n), mentioned previously. This set can be visualized in Figure 2.3, highlighted in green.
A time-surface storing at the neighbor pixels defined by I_n, the time t_n is updated to provide an easy to access event history. The update of this time-surface is [1] is defined as:

$$\forall (x, y) \in I_n, M_T(x, y, n) = t_n, \tag{2.10}$$

[1] All other values remain unchanged

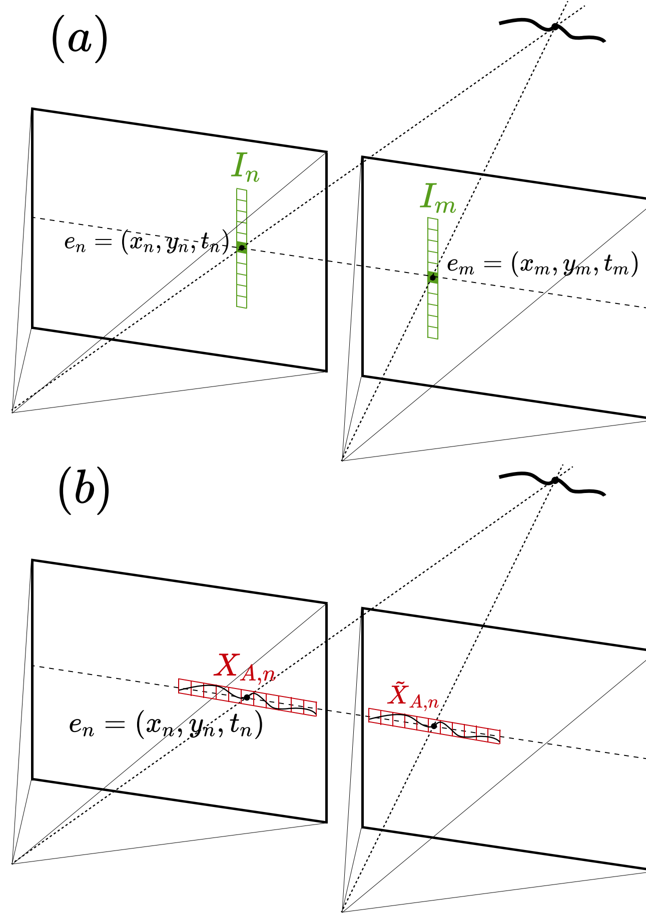

Figure 2.3 (a) : *Given an event e_n, the set I_n defined in eq. 2.9 collects all pixel of interest to build the local time-surface necessary to build the spatio-temporal descriptor time-surfaces for each sensor.* (b) *Row vector spatio-temporal descriptors centered around (x_n, y_n), $X_{A,n}$ and $\hat{X}_{A,n}$. Rows are coinciding with epipolar lines due to rectified sensors.*

A decay function λ is designed to favor the temporally close neighbors over too old ones at location (x_n, y_n), at time t_n (corresponding to an event e_n), according to:

$$\begin{aligned} &\forall (x,y) \in I_n, \\ &\lambda(x,y,t_n) = \exp\left(-\frac{t_n - M_T(x,y)}{\tau}\right) \end{aligned} \tag{2.11}$$

for all (x, y) such that $x = x_n$ and $|y - y_n| \leq R_y$. The factor τ should reflect the scene dynamic, but with no prior from the scene, it is set to some empirical value. This decay rate's purpose is to maintain the scene dynamic in the three other intermediate maps we used for the depth computation. These maps are explained in the next subsections.

b) Activity map

The activity surface is the function defined as:

$$\begin{aligned} & && \forall (x,y) \in I_n, \\ &M_{A,n}(x,y) && = \lambda(x,y,t_n).M_{A,n-1}(x,y) + 1, \end{aligned} \tag{2.12}$$

and A_0 is initialized to zero at time t=0. Again, all pixels $(x, y) \in I_n$ of $M_{\mathcal{A},n}$ are updated. That activity map has a double purpose. It first serves as support for spatial descriptors of the scene as it counts the number of events affecting each pixel w.r.t the definition in equation 2.9. It also serves as a normalization factor for the other maps defined further. Given any map $M_{X,n}$ used in this work, we define its normalization by $M_{A,n}$ as:

$$\bar{M}_{X,n}(x, y) = \frac{M_{X,n}(x, y)}{M_{\mathcal{A},n}(x, y)} \tag{2.13}$$

c) Spatial distribution maps

These maps provide a local spatial "description" around each event at (x_n, y_n). To this end, a spatial neighborhood is integrated into the map $M_{S,n}$, with the following update rule:

$$\begin{aligned} & \forall (x, y) \in I_n, \\ M_{S,n}(x, y) & = \lambda(x, y, t_n).M_{S,n-1}(x, y) + (y_n - y), \end{aligned} \tag{2.14}$$

for all i such that $|y_n - y_i| \leq R_y$, where R_y is a constant chosen empirically. There is no constraint on the x-axis i.e. all event above the row y_n are taken into account.

The second map integrates the spatial distribution variance, also weighted by the decay function λ:

$$\begin{aligned} & \forall (x, y) \in I_n, \\ M_{\sigma,n}(x, y) & = \lambda(x, y, t_n).M_{\sigma,n-1}(x, y) + y_n^2 - \bar{M}_{S,n}(x, y)^2 \end{aligned} \tag{2.15}$$

These maps are updated continuously via the decaying function λ that maintains the temporal dynamic of the scene. The Fig.2.4 shows an example of the three maps generated and updated event-y-event for the MVSEC Indoor_flying sequence.

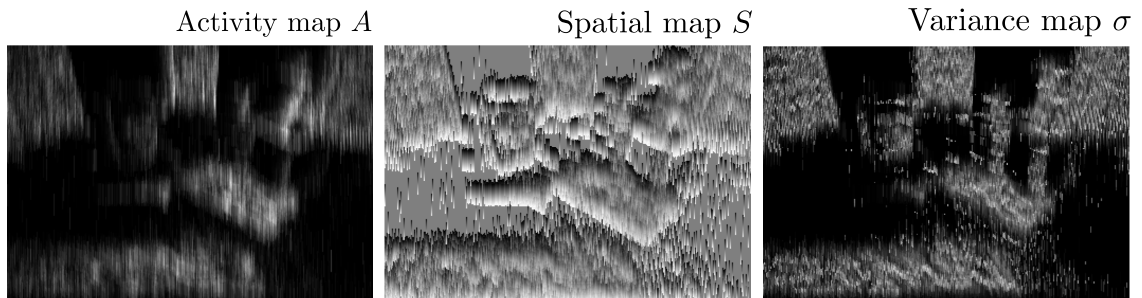

Figure 2.4 *The spatio-temporal maps are updated for each event, describing different scene spatial information.*

d) A 1-D spatio-temporal descriptor for stereo-matching

For an event e_n, the activity map and the spatial distribution maps defined above are used to extract vectors of length $2 \times R_x + 1$, on row y_n and centered at x_n:

$$\begin{aligned} \boldsymbol{X}_{A,n} &= \begin{pmatrix} M_{\mathcal{A},n}(x_n - Rx, y_n) \\ \dots \\ M_{\mathcal{A},n}(x_n + R_x, y_n) \end{pmatrix} \\ \boldsymbol{X}_{S,n} &= \begin{pmatrix} \bar{M}_{S,n}(x_n - Rx, y_n) \\ \dots \\ \bar{M}_{S,n}(x_n + R_x, y_n) \end{pmatrix} \\ \boldsymbol{X}_{\sigma,n} &= \begin{pmatrix} \bar{M}_{\sigma,n}(x_n - Rx, y_n) \\ \dots \\ \bar{M}_{\sigma,n}(x_n + R_x, y_n) \end{pmatrix} \end{aligned} \tag{2.16}$$

As the sensors are assumed to be rectified, those vectors extracted from row of pixels form a 1-D descriptor that are used for the stereo matching procedure. Let us assume that we have two vectors $\boldsymbol{X}^1$ and $\boldsymbol{X}^2$ of the same nature or modality, from sensor 1 and sensor 2. Their similarity is measured with the cosine similarity:

$$C(\boldsymbol{X}^1, \boldsymbol{X}^2) = \frac{\boldsymbol{X}^{1T}\boldsymbol{X}^2}{||\boldsymbol{X}^1||.||\boldsymbol{X}^2||}. \tag{2.17}$$

For the three modalities we are computing the following cosine similarities:

$$\begin{aligned} s_A &= C(\boldsymbol{X}_A^1, \boldsymbol{X}_A^2) \\ s_S &= \tfrac{1+C(\boldsymbol{X}_S^1, \boldsymbol{X}_S^2)}{2} \\ S_\sigma &= C(\boldsymbol{X}_\sigma^1, \boldsymbol{X}_\sigma^2), \end{aligned} \tag{2.18}$$

which gives the global similarity score $s = s_A \times s_S \times s_\sigma$. s_S is slightly different as S_n can be negative. We re-centered and re-scaled it into the interval $[0, 1]$.

The spatial dependence for each similarity is omitted for readability reason, however each vector $\boldsymbol{X}^1, \boldsymbol{X}^2, ...$ are computed at locations of events to be compared. Fig.2.4 shows a sample of the A, S and σ maps at some time t for the indoor_flying sequence of the MVSEC dataset and the 1-D descriptor resulting from the $\boldsymbol{X}_$ vectors.

e) Key points selection

The stereo-matching leads straightforwardly to the depth estimation, however it does not starts as soon as the first event arrive, but rather we have implemented a "key-points" selection operation. This consists in creating candidates for matching when events occur at locations where the activity map A_n is above a threshold th_a. This operation ensures that events to match are less likely to be triggered by noise. This key-points selection is performed for both sensors. To avoid over-selection/detection, events that are not from a local maximum of the activity map are not considered as key-points.

The matching procedure for events at key-points location is working as follows: if an event e_n occurs at the first sensor, the candidates to test in the second sensor are events that occurs close to the key-points position on the corresponding epipolar line. The best match is the first event,

hence closer in time, leading to a high enough similarity score s. At this stage, a disparity map D, of the same size of the sensors, is updated to store the estimated disparity at pixel (x_n, y_n):

$$\begin{aligned} D_n : \mathbb{N}^2 \rightarrow & \quad \mathbb{N} \\ (x_n, y_n) \mapsto & \quad D_n(x_n, y_n) = x_n - x_i, \end{aligned} \tag{2.19}$$

where (x_i, y_i), with $|y_n - y_i| < R_y$, is the position of the matched location in the second camera.

f) Disparity propagation

Matching events with the key-points is the first step for building the disparity maps (one for each sensor). To densify the disparity maps, we propagate the disparity information around these key-points to match events located at non key-points locations. This propagation is reasonable as even for the event-based sensors, depth information should continuous (e.g. along edges). The disparity propagation is achieved as follows :

- for an event e_n of sensor 1, gather from its disparity map D, all available disparities d_i within a radius of 1 from (x_n, y_n). This form the $\mathcal{N} = \{d_i\}$,
- build the 1-D descriptors at $(x_n + d_i, y_n)$ for $d_i \in \mathcal{N}$ with the maps of sensor 2,
- Calculate the similarity scores s as defined in section 2.2.2.d at pixel (x_n, y_n) and pixels $(x_n + d_i, y_n)$,
- Select disparity d_n at x_n, y_n, defined by $d_n = x_n - x_i$ and maximizing s,
- Store disparity $D_n(x_n, y_n) = d_n$.

g) Winner-take-all filtering

The last step for the disparity computation is a local winner-take-all, to increase even further the stability of the system. Over a spatio-temporal patch of radius R and time-window τ around pixel (x_n, y_n), we retrieve and count the unique disparities values stored in D_n.

$$\mathcal{S}_n = \left\{ D_n(x, y), \begin{array}{l} |x - \tilde{x_n}| \leq R \\ |y - \tilde{y_n}| \leq R \\ \mathcal{M}_{T,n}(x, y) \geq t_n - \tau \end{array} \right\} \tag{2.20}$$

As the physical $3D$ object is continuous, the disparity associated will be continuous, and even locally constant in most cases. Thus, we propagate a disparity $\overline{d_n}$ that is the most occurring disparity within the set $\mathcal{S}_n$. The benefits found with this method greatly overweight the issues it may pose in edge cases, such as obstructions.
Figure 2.5 sums up the entire stereo-matching pipeline as described in this section. Additionally, a study of this standalone block is reported in section 2.3.2, using an indoor controlled scene, with a known ground-truth to assess its precision.

2.2.3 Kinematic from depth map and optical flow

The third and final block we present is an event-based kinematic estimator. Contrary to the two previous block, it relies also on optical flow and disparity values related to the incoming event. As such, in this section, we assume that this additional data is known for the event e_n.
In the scenario of a static scene within which the stereo-rig is moving, the apparent motion is translated into optical flow in each sensor into the focal plane. Let us assume that a 3D point

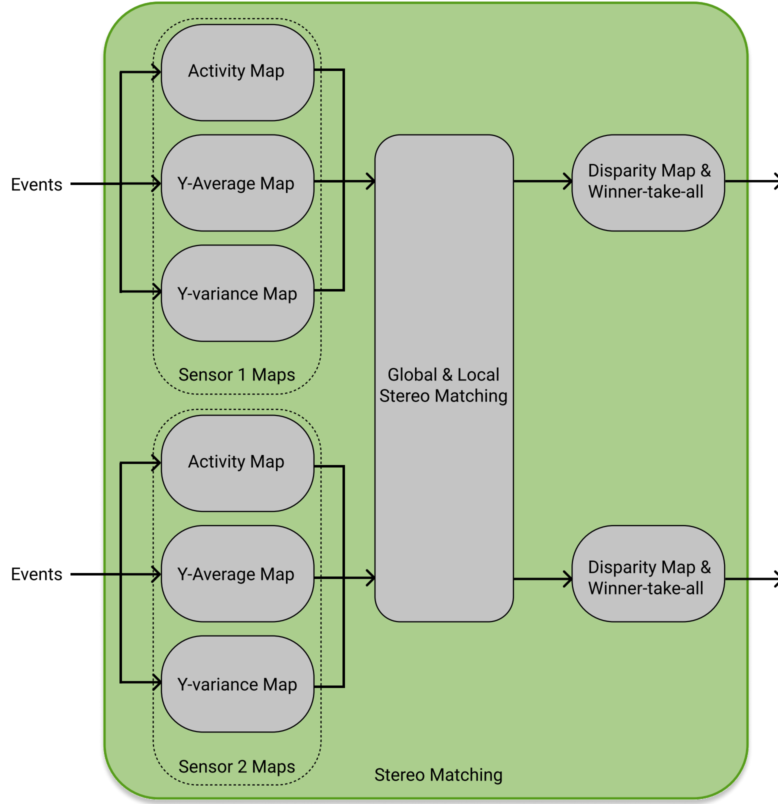

Figure 2.5 *Detailed pipeline of the stereo-matching algorithm.*

$\mathbf{M} = (X, Y, Z)^T$ moving w.r.t. the sensors and $\mathbf{m} = (X', Y', Z')^T$, its projection into a sensor's focal plane, then their respective velocities are defined as:

$$\begin{aligned} \dot{\mathbf{M}} &= \boldsymbol{\omega} \times \mathbf{M} + \mathbf{t} \\ \dot{\mathbf{m}} &= \tfrac{f}{Z}\dot{\mathbf{M}} - \tfrac{\dot{Z}}{Z}\mathbf{m} \end{aligned}, \tag{2.21}$$

where $\boldsymbol{\omega}$ and $\mathbf{t}$ are respectively the angular and translation velocities, and f is the sensor's focal length. The real expected optical flow $\tilde{\boldsymbol{v}}$ is related to $\dot{\mathbf{m}}$ via the 2×2 sub-matrix K, extracted from the two first rows and columns of the sensor's intrinsic matrix:

$$\begin{aligned} \tilde{\boldsymbol{v}} &= K \begin{pmatrix} \dot{X}' \\ \dot{Y}' \end{pmatrix} \\ &= \underbrace{K \begin{pmatrix} -\frac{X'Y'}{f} & \frac{f^2+X'^2}{f} & -Y' & \frac{f}{Z} & 0 & \frac{X'}{Z} \\ -\frac{f^2+Y'^2}{f} & \frac{X'Y'}{f} & X' & 0 & \frac{f}{Z} & -\frac{Y'}{Z} \end{pmatrix}}_{P} \boldsymbol{\Omega} \end{aligned} \tag{2.22}$$

where $\boldsymbol{\Omega} = \begin{pmatrix} \omega_X & \omega_Y & \omega_Z & t_X & t_Y & t_Z \end{pmatrix}^T$.

It is more a common practice to estimate the optical flow from 3D structures using eq.2.22. If we are providing the left-hand side of the equation and the 3D map of the scene, we can recover sensor's velocities $\boldsymbol{\omega}$ and $\mathbf{t}$ from the same equation if enough independent equations are provided i.e. since each pair of $(\mathbf{M}, \dot{\mathbf{m}})$ provides two independent scalar equations and we have six unknowns

to estimate, thus at least three 3D points and their optical flow are required. Each point should thus satisfy

$$\tilde{\boldsymbol{v}}_{\boldsymbol{i}} = P_i \boldsymbol{\Omega} \tag{2.23}$$

Up to estimation errors, $\tilde{\boldsymbol{v}}_{\boldsymbol{i}}$ should coincide with the optical flow v_i estimated from the independently computed optical flow map. There is no prior requirement on the optical flow technique to use, however, to preserve the event-based representation, it is natural to favor techniques like [66]. We can solve eq.2.23 for $\boldsymbol{\Omega}$ with an event-based form of the least square minimization of the cost function :

$$\begin{aligned} \mathcal{E}(t_n) &= \sum_{i \le n} w_i(t_n) ||\tilde{\boldsymbol{v}}_{\boldsymbol{i}} - \boldsymbol{v}_{\boldsymbol{i}}||^2 \\ &= \sum_{i \le n} w_i(t_n) ||P_i \boldsymbol{\Omega} - \boldsymbol{v}_{\boldsymbol{i}}||^2 \end{aligned} \tag{2.24}$$

where $w_i(t_n) = e^{-\frac{t_n - t_i}{\tau}}$. $\mathcal{E}$ is minimized when its gradient w.r.t. $\boldsymbol{\Omega}$ vanishes i.e. when:

$$\begin{aligned} &\sum_{i \le n} w_i(t_n) P_i^T (P_i \boldsymbol{\Omega} - \boldsymbol{v}_{\boldsymbol{i}}) = 0 \\ \Leftrightarrow &\underbrace{\sum_{i \le n} w_i(t_n) P_i^T P_i}_{A(t_n)} \boldsymbol{\Omega} = \underbrace{\sum_{i \le n} w_i(t_n) P_i^T \boldsymbol{v}_{\boldsymbol{i}}}_{\boldsymbol{\Sigma}(t_n)} \end{aligned} \tag{2.25}$$

hence $\boldsymbol{\Omega}$ is given by inverting the 6×6 matrix $A(t_n)$:

$$\boldsymbol{\Omega}(t_n) = A^{-1}(t_n) \boldsymbol{\Sigma}(t_n). \tag{2.26}$$

All optical flow estimation techniques suffer to some extend from the aperture problem due of the finite field of view of the sensor. Because of that, the normal (to the contour) component of the flow is generally the most, if not the only, reliable information that can be estimated. Thus, without excluding any event-based optical flow estimation technique found in the literature, we assume to use only that normal component of the flow to recover the real velocity of the sensor. To this end, let us define $\boldsymbol{n}_{\boldsymbol{i}}$, the unit normal to the contours that can be built from the events and the normal flow $\tilde{f}_i$ as:

$$\begin{aligned} \tilde{f}_i &\equiv \boldsymbol{n}_{\boldsymbol{i}}^T \tilde{\boldsymbol{v}}_{\boldsymbol{i}} \\ &= \boldsymbol{n}_{\boldsymbol{i}}^T P_i \boldsymbol{\Omega} \end{aligned} \tag{2.27}$$

By substituting the theoretical optical flow $\boldsymbol{v}_{\boldsymbol{i}}$ with its normal component f_i, eq. 2.24 restricted to its normal component becomes:

$$\mathcal{E}(t_n) = \sum_{i \le n} w_i(t_n) \left(\tilde{f}_i - f_i\right)^2, \tag{2.28}$$

which is least-square minimized if:

$$\underbrace{\sum_{i \le n} w_i(t_n) P_i^T \boldsymbol{n}_{\boldsymbol{i}} \boldsymbol{n}_{\boldsymbol{i}}^T P_i}_{A(t_n)} \boldsymbol{\Omega} = \underbrace{\sum_{i \le n} w_i(t_n) f_i P_i^T \boldsymbol{n}_{\boldsymbol{i}}}_{\boldsymbol{\Sigma}(t_n)} \tag{2.29}$$

As P_i is a 2×6 matrix, $P_i^T \boldsymbol{n}_i \boldsymbol{n}_i^T P_i$ is not full rank. A more accurate optical flow will provide more constraints to solve for $\boldsymbol{\Omega}$ with the same amount of data. Since events are sequential, the

minimization process converges faster if we are not restricting to the normal flow. However, we focus on eq. 2.28 to ensure that whatever the flow estimation technique is used upstream, assuming the normal component is accurate enough, the pose can be estimated.

The proposed approach differs from works found in literature where they focus on sensor's poses estimation. As we are dealing with event-based camera, we have mainly access to the first order information from the scene. As such, our approach aims to estimate the "ego-kinematics" rather than the traditional ego-motion of the sensor. The core steps of this method are schematized in Figure 2.6.

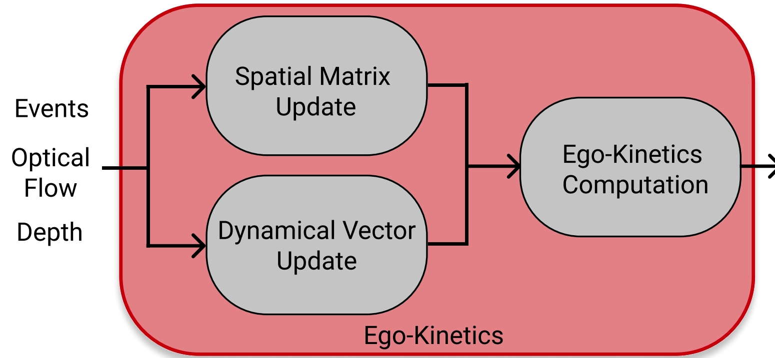

Figure 2.6 *The ego-kinetics computation consists in three steps. The two first are updates of the spatial matrix A and the dynamical vector $\mathbf{\Sigma}$, using the optical flow $\boldsymbol{f_n}$ and the depth D_n associated with an event e_n. The last step is the rotation and translation velocities estimation, given by $\mathbf{\Omega}$.*

Experiments on this process are shown in section 2.3.3, using the previously described blocks a primary inputs.

2.2.4 Maps life-time and and inter-maps latencies

All the maps are built from dynamic inputs and updated in an continuous manner. The persistence of information in each map is handled by an exponential function λ, where the decay rate τ is representing the life-span of the information. How to let the processing pipeline to estimate and regulate itself this decay rate is a key problem to this work as this will impact all the processing chain performances. As chained maps are integrating higher and higher level information, the life-span of the meaningful information at a given time is defined differently. For each map, the assumed decay rate at the beginning is provided by the previous maps that is is directly fed from: for example, life time of a new piece of information for the flow is given by the raw events themselves, while for the kinematic estimation map, the decay rate is inferred from the result output by the flow.

At the lowest level of the event data, we define the decay rate τ as a function of a density parameter d, the ratio of pixels that have emitted an event. It is straightforward to see that for a fixed d, τ increases if the scene apparent motion is slow and decreases if it is fast. This is a crude first estimation of events life-time. In practice the auto-regulation of τ is local and computed as

$$\tau = \tau_0 \frac{\log \frac{1}{1-1/A_n}}{\log \frac{1}{1-1/(\mathrm{d}S)}}, \tag{2.30}$$

where A_n is the activity function introduced by eq.2.12, τ_0 is a default decay parameter and S is the area in pixels of the sensor.

From the optical flow maps, the decay rate is getting more precise and is locally related to the computed flow. If $\mathbf{f_n}$ is the flow map, then $\tau = \tau_f = \frac{1}{||\mathbf{f_n}||}$.

In our case of study, the higher level map is the one estimating the kinematic and $\tau = \tau_k = \frac{1}{||\mathbf{v}||}$, where $\mathbf{v}$ is the velocity vector output by last building block of our pipeline.

This decay rate is at first hand fixed as a global value and applied to each map. However, it is reasonable to assume that each maps has it own decay rate and the decay rates from one lower level map to a higher one are likely related. We are investigating such relationship in order to define a simple parametrization of each map. A deeper analysis of timescale importance in event-based computation will be presented in the first part of the last chapter. This first analysis was the basis of this work, as it highlighted the importance of well-defined timescales, and how valuable dynamic ones can be to the performance of our algorithms. Results concerning this specific parameter will be given in the following sections.

2.3 Experiments

2.3.1 Optical Flow benchmark

We report in this first experimental section results on the optical flow. To assess its validity, we compare it to the more traditional plan fitting method[52], known to be reliable in optical flow orientation. To that end, we use a simple, controlled sequence of a translating square, moving at a constant speed of $1000px/s$, and recorded with a Q-VGA neuromorphic sensor of Prophesee. We build the ground-truth by recording the corners location, and computing the theoretical optical flow for each side of the square, taking into account the aperture issue induced. The scene is shown in Figure 2.7.

Both methods share a similar set of parameters : a spatial neighbourhood radius R, a maximum temporal depth τ, and a minumum number of events used N_{min}. thus, we set the same values for those parameters, to ensure a proper comparison is made. We use the endpoint norm, and the orientation error, to compare both methods. The results are given in Table 2.1.

We observe that under the same constraints, the gradient-based optical flow produces more flows datapoints then the plane fitting method and the estimate vectors are more accurate in amplitude while slightly worse in orientation. As we have developed in section 2.2.3, the more accurate the flow amplitude $\tilde{f}_i$, the better the camera's velocity estimation since the cost function in eq. 1.19 is to be minimized w.r.t. $\tilde{f}_i$.

2.3.2 Stereo Matching Benchmark

The second block, described in Section 2.2.2, is evaluated with a controlled indoor scene, for which the ground-truth is know. We place two planar objects, distant of respectively $1.15m$ and $2.50m$ from the stereo rig center. The scene is recorded with a pair of DAVIS 346 sensors from *IniLabs*. Those sensors are calibrated and rectified, such that at any given time, the relation between depth and disparity can be computed. A spatio-temporal view of this scene is given in Figure 2.8(a).

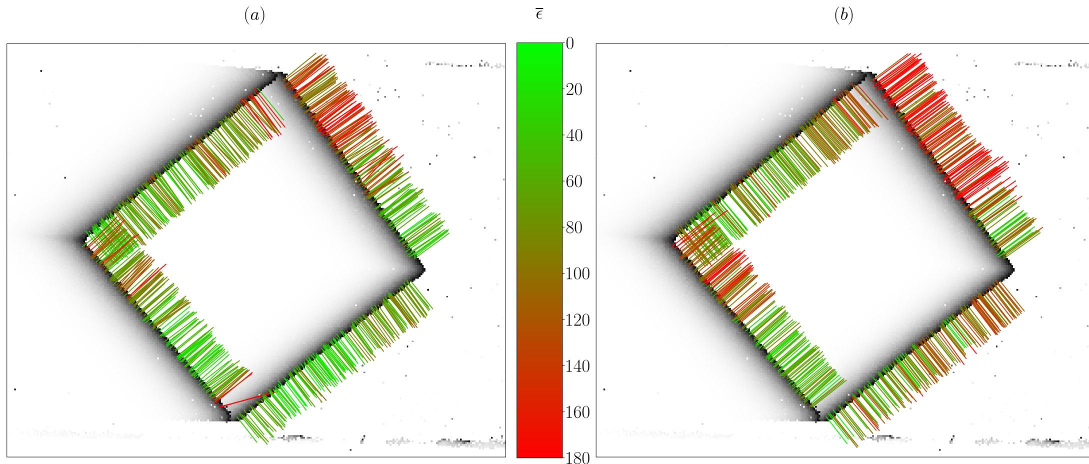

Figure 2.7 *Spatio-temporal context of the translating square used for the optical flow method benchmark. Optical flow samples are added, color-coded with the norm of the endpoint error. (a): Our gradient flow method, with an average endpoint error of* $56.2px/s$*. (a): The traditional plan fitting flow method, with an average endpoint error of* $83.9px/s$*. The corners have been purposely removed from the comparison, as the plan fitting method cannot properly describe the optical flow in such cases.*

		Plan Fitting	Gradient Flow
# Flows produced		1608	1719
Avg. Endpoint Error	$\bar{\epsilon}$	$83.9px/s$	$\mathbf{56.2px/s}$
	$\sigma_{epsilon}$	$65.2px/s$	$\mathbf{36.0px/s}$
Avg. Relative Endpoint Error	$\bar{\epsilon}$	11.8%	**7.9%**
	$\sigma_{epsilon}$	8.9%	**4.9%**
Avg. Angular Error	$\bar{\epsilon}$	**0.52°**	0.57°
	$\sigma_{epsilon}$	**1.01°**	2.44°

Table 2.1 *Comparison results between the groundtruth optical flow, and two optical flow method. We report the number of optical flow produced, as each method has its own rejection rules, as well as errors and standard deviations for the endpoint error (px/s), the endpoint error relative to the expected norm (%) and the angular error (deg).*

We build the ground-truth and compare it with the outcome of our stereo-matching algorithm. Those results can be visualized in Figure 2.8(b). The $3D$ errors are reported in Table 2.2. We notice that 98% show errors of less than $5cm$, assessing for the quality of the matching on this scene.

2.3.3 Ego-kinetics evaluation

We built the complex system made of two event-based stereo streams as schematized in fig 2.1. We daisy-chained parallel low level feature maps to create higher and higher level building blocks such as event-based disparity maps in a asynchronous and time-continuous manner. This provides

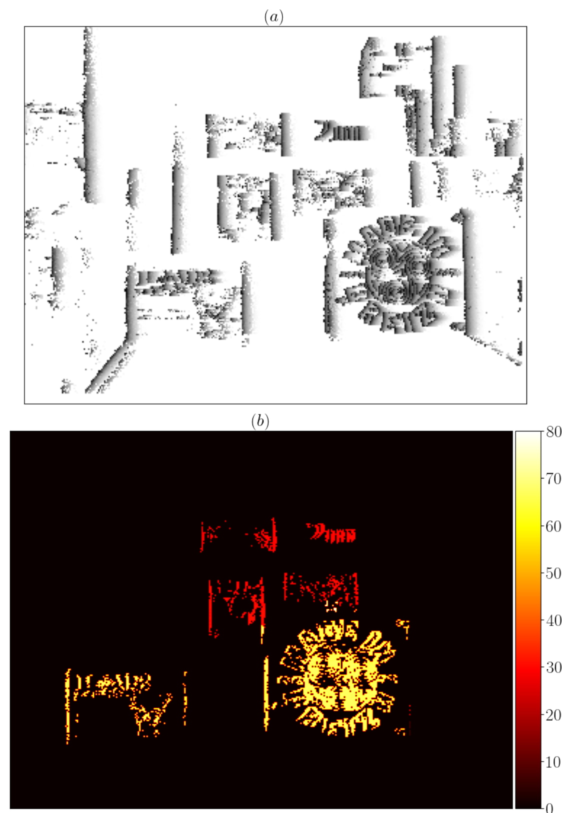

Figure 2.8 *Results of the stereo-matching block on an indoor controlled scene. (a): Spatio-temporal context of the scene. We distinguish, two successive planes, with different drawings allowing for a large diversity of shapes to be matched. (b): Visual result of the matching algorithm. We observe two well-defined plans, with little disparity error. We only consider the ground-truth of the two planes, as these we the only points the distance could be accurately defined.*

Maximum Error	Ratio of Points Matched
$1cm$	97.5%
$5cm$	98.0%
$10cm$	99.6%

Table 2.2 *Ratio of points found relative to the error to the ground-truth.*

a continuous monitoring of the scene depth and it feeds 3D information to the cameras kinematic block that, once combined with the optical flow maps, allows for direct estimation of cameras' velocities according to section 2.2.3.
The evaluation of the pipeline is achieved with three sequences of event-based data: the ESIM sequence is a synthetic data generated from an event camera simulator; the Doll and Walking sequences are real scenes acquired by us with a synchronized stereo-rig.

a) ESIM simulator

The simulator is used to provide a quick preliminary assessment of the algorithm behavior. The scenario is simple with the stereo-rig being rotated about the optical axis of the left camera, facing a plane with geometric shapes. The kinematic estimation is applied to the left camera as shown in Figure 2.9:

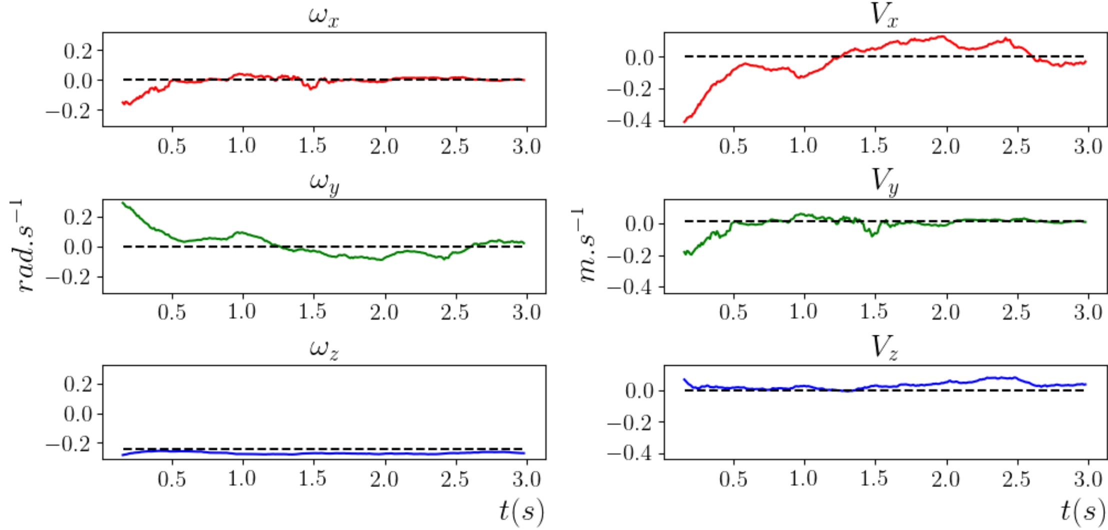

Figure 2.9 *Velocities estimation on a sequence simulated on ESIM. The ground-truth - in dashed -, shows a constant angular velocity about the z-axis, of* $-0.3rad/s$ *and a negligible y-translation of* $2.10^{-2}m.s^{-1}$.

The sharp increase of the velocity happens at the beginning of the motion as the processing chain requires delay to achieve correct estimate. Table 2.3 is summarizing the estimation errors for three scales of decay rate deduced from the flow map. The larger the scale, the slower the system converge into the correct solution. The best results are obtained in this case for the shortest decay rate of τ.

	ϵ_{ω_x}	ϵ_{ω_y}	ϵ_{ω_z}	ϵ_{v_x}	ϵ_{v_y}	ϵ_{v_z}
τ	**2.09**	**6.52**	3.90	**9.26**	**2.83**	4.55
4τ	12.85	8.52	3.86	12.17	18.52	3.18
10τ	3.38	10.21	**3.80**	14.74	5.25	**2.77**
τ_0	9.03	6.72	4.16	9.34	13.38	4.79

Table 2.3 *Root mean square errors for each component of the velocities. The three first rows show the estimation errors obtained from the adaptive decay rate strategy for different scale of* τ *inferred from the flow maps. Last row shows the result obtained with a global and fixed decay rate* tau_0. *Errors are expressed either in* $10^{-2}rad.s^{-1}$ *for angular velocity, and* $10^{-2}m.s^{-1}$ *for translation velocity.*

b) Doll sequence

The sequence is an indoor scene where the stereo-rig is recording while manually moved in front of a static wooden doll put on a table. The embedded IMU is providing the angular velocity and translation acceleration measurements that serve as ground-truth to assess the estimation accuracy. As we do not have the translation velocity, we are only providing evaluation for the angular velocity. The impact of setting a global and fixed decay constant or an adaptive decay constant is also outlined by the left and right column respectively. The global decay constant is chosen according to an experimental prior, while the adaptive decay is driven by the decay deduced from lower level maps. Figure 2.10(a-b) show the angular velocities overlaid with the imu measurements serving as ground truth and Table 2.4 is summarizing the velocity estimation performances for three scale of the adaptive decay constant and for the fixed and global decay constant. Errors are decreasing as the decay increases and at ten times the initial τ, the mean errors are comparable to the globally set decay constant $\tau_0 = 80ms$.

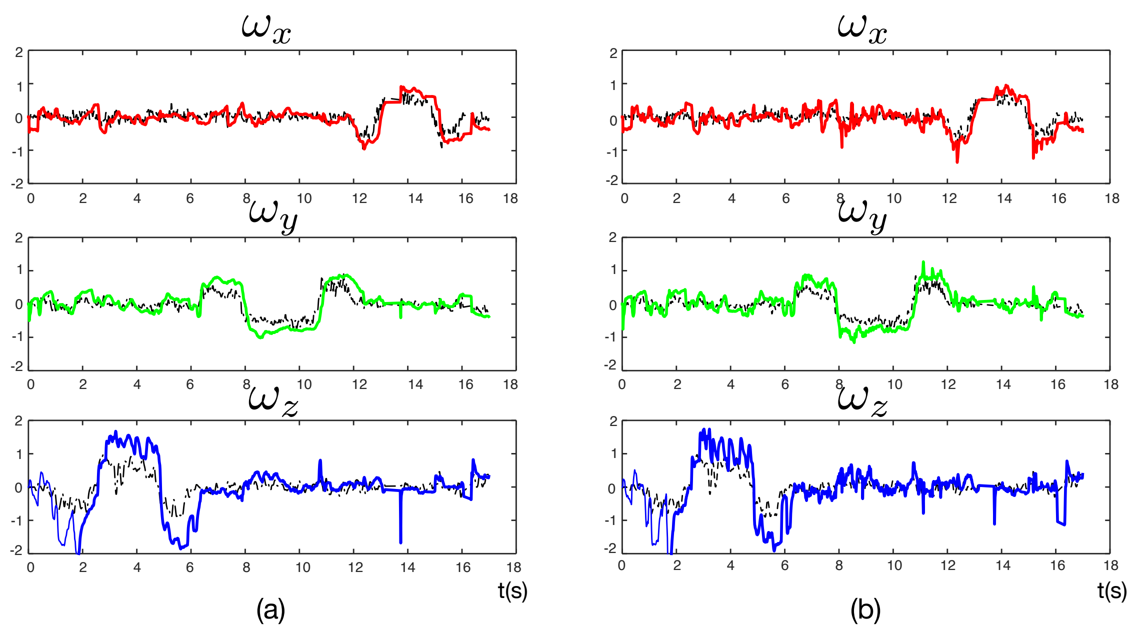

Figure 2.10 *(a) Angular velocities estimation on the Doll sequence with constant timescale $\tau = 80ms$. (b) Angular estimation for the same sequence with an adaptive timescale.*

	τ	4τ	10τ	τ_0
$\epsilon_x(\%)$	14.56	12.46	11.04	10.35
$\epsilon_y(\%)$	14.91	13.86	12.41	12.49
$\epsilon_z(\%)$	18.13	16.85	17.98	20.47

Table 2.4 *Relative (to the swing of the ground-truth signal for each axis) angular velocity errors for 3 adaptive decay constant scaled w.r.t. the τ, calculated from the lower level maps and for a experimentaly global decay constant τ_0.*

c) Walking sequences

These sequences tested for the ego kinematics estimation are sequences of $4.5s$ and $86.3s$ long in duration. They are referred to as *walk_a* and *walk_b* respectively. *walk_a* is acquired indoor, while *walk_b* is a sequence acquired indoor and outdoor. Both are evaluated for the angular velocity, as no ground-truth was available for the translation part of the sensors kinetics. Figure 2.11 reports those rotation velocities comparisons, and the match between the ground-truth and the recovered motion. Images of those scenes are displayed in Figure 2.12, showing events and depth visualization of indoor and outdoor sections.

	walk_a	*walk_b*
Duration	4.5s	86.3s
mean angular velocity error	0.285	0.298
mean ϵ_x (rad/sec)	0.093	0.115
mean ϵ_y (rad/sec)	0.188	0.182
mean ϵ_z (rad/sec)	0.144	0.149
Global angular error	$0.82rad$	$8.3rad$

Table 2.5 *Angular velocity errors for two hand-held sequences, in indoor and outdoor environments. We present the rotation speed error in norm and for each axis of the sensor. We compute the rotation as the integration over time of the velocity error. If not specified otherwise, the values are expressed in* $rad.s^{-1}$.

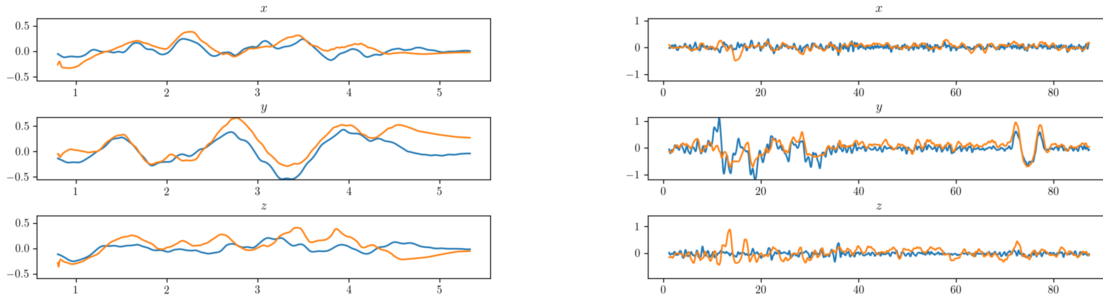

Figure 2.11 *Comparison of the recovered rotation velocity by our algorithm with the ground-truth values given by the sensors' IMU. Detailed statistics about those sequences are reported in Table 2.5. Left : Walking sequence* (a) *of* $4.5s$ *in an indoor environment. Right :* $86.3s$ *long sequence* (b) *with indoor and outdoor sections.*

We can however have a coarse estimation of the translation component from the known displacement in the scene. This measurements are made at the beginning and the end of the sequence, giving us the overall motion that occurred. In both cases, the distances in indoor environment have shown good coherence, and estimated speeds are matching, in order of magnitude, the walking pace at which the recordings have been done. Those results are shown in Table 2.6. The translation velocity for *walk_b* failed for an acceptable comparison after the $8m$ meters of the sequence. We hypothesized that outdoor scene flow's spanned a little range of disparity. Hence while angular velocity can be fairly estimated, the translation part cannot be.

Sequence	(a)	(b)
Estimated Distance	$2.45m$	$8.1m$
Measured Distance	$3m$	$8m$
Average Speed	$2.4km.h^{-1}$	$3.6km.h^{-1}$

Table 2.6 *Comparison for the displacement and average speed for the two recorded scenes. The accuracy in estimated distance shows that 3D displacements can be inferred from optical flow and depth.*

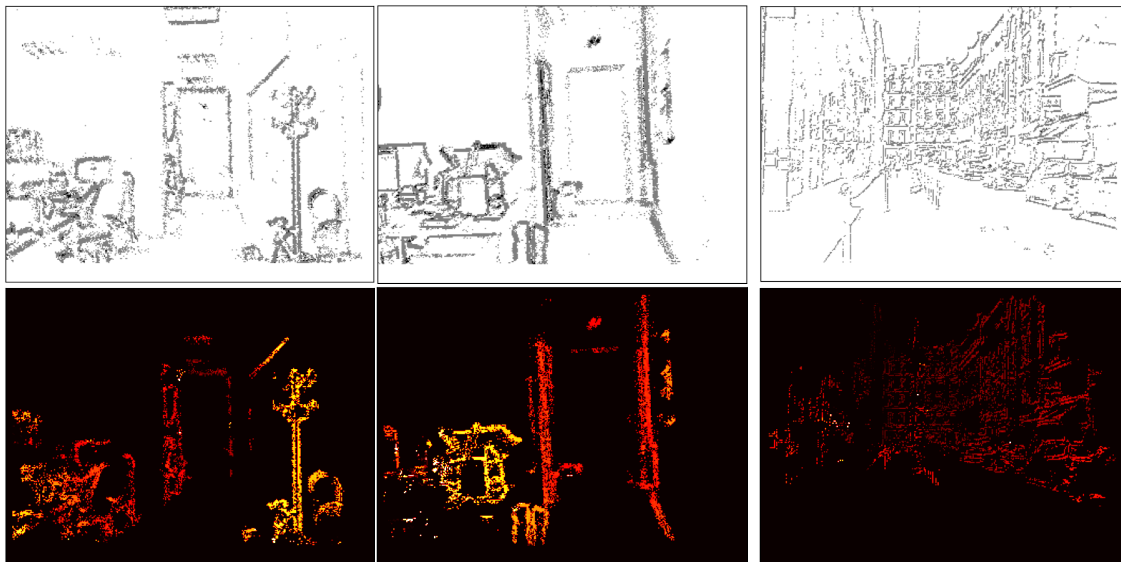

Figure 2.12 *Snapshots of the indoor-only sequence* (a) *(left), and mixed sequence* (b) *with indoor (center) and outdoor (right) sections. We show reconstructed frames from events (top row) and depth visualization (bottom row).*

2.4 Conclusion

The three described algorithms, respectively on optical flow, depth estimation from disparity and ego-kinetics estimation make full use of the event-based paradigm. By computing asynchronously the data from any type of neuromorphic retina, they show with little prior on the data how low latency can be achieve, with little computational cost. Indeed, none of this three method had to rely on large optimization processes, working on assembled sets of events. The use of dynamical timescales, detailed in section 4.2, allows to discard even more parameters for ours model, while keeping versatile abilities.

The algorithms were benchmarked on scenes of different natures, with similar results, showing excellent quality for both the optical flow and the stereo-matching, and an encouraging trend for the ego-kinetics estimation. However, as previously stated, the limitations of ground-truth recordings made some of these benchmarks limited, and more in-depth work is necessary the push the model to its limits, while assessing its validity in more diverse situations.

Chapter 3

Binocular visual odometry

3.1 Introduction

The goal of this thesis was to tackle the problem of visual odometry, while implementing fully event-based methods. Previous chapter focused on building a multi-maps from low to higher level abstraction of visual information. This hierarchical approach aims to maintain a low latency and time-continuous processing of the events by chaining optical flow to 3D depth information and finally to the velocity estimation for a stereo-rig. These are, as we announced in the previous chapters, the building blocks aimed to solved visual odometry problem from a global first order approach, and ultimately to solve the Simultaneous Localization And Mapping (SLAM) problem. It is the ability for a system to recover its location while simultaneously reconstructing a 3D representation of its environment. Both processes occur in parallel, in a closed-loop format. It is thus extremely challenging, as a local failure of one of those two processes generates self-amplified errors over time.
It is today considered the grail of robotics, as it is a primal need for a robot to interact with its surroundings. However, most methods that can be found in the literature use multiple sensors apart from visual input : LIDARs for depth estimation, IMUs to estimate kinetic information, and use extensive computations to fuse all those inputs in order to recover a visual odometry. If [94] proposes a solution without IMU measurement, this solution induces a computationally-heavy RANSAC process to estimate equivalent inertial measurements. The overall computation times for each step - about $100ms$ are strong hints that low latency can only be achieved by changing the paradigm used.

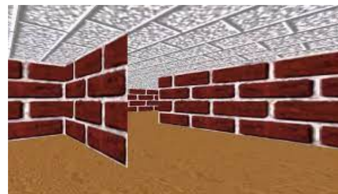

Figure 3.1 *Windows 95 maze screensaver view. With nothing more than the visual input of a single screen, a human can accurately estimate its position within the maze. This guided us to try to develop a visual odometry algorithm with no additional data than the visual inputs of a stereo rig.*

A simple thought experiment tells us that nothing more than a single visual sensor is needed

for this purpose. Let us take a seemingly insignificant example to demonstrate that. In August 1995, Microsoft released its latest operating system, and in this Windows 95 OS was included a screensaver that became iconic. It represented a $3D$ view of the insides of a maze, being travelled randomly, with the graphics quality available at the time.
This screensaver, however basic, is a good hint that the pose of a system can be recovered only from visual input, given the correct rules of data continuity. If this assertion has been shown valid throughout SLAM development in recent years [95], this simple example can talk to everybody. It encourages us to research how we naturally process visual inputs to solve this problem.

We have to acknowledge from the start that human visual processing involves much higher visual abstraction, for example analyzing macroscopic structures such as walls, paths, ... that simple visual processing can only process as shapes. The associated constraints, such as walls being solid objects that hide objects behind, or having most objects lying on the ground, are many clues that help us processing this scene, and that cannot easily be replicated with simple heuristics. Those clues allow humans to understand the former maze, while only monocular vision is available. More complex scenes, involving unusual environments, can make it much more difficult for humans to locate themselves, as everyday landmarks are missing, and those clues are missing.

Yet, those clues are not an absolute necessity for visual odometry. By using depth information thanks to binocular systems, the scene can be much more easily abstracted into a $3D$ map. This greatly reduces the computational load required to recover visual odometry, with little to no priors about scene, and a much lower latency.

One of the challenges in standard monocular visual odometry is the addition of new 3D points, mandatory to the motion estimation. 3D structures can be reliably triangulated only if the stereo basis is sufficiently large i.e. the displacement of the camera is sufficiently large between the two instants. This naturally requires additional time, therefore the global latency of the processing pipeline increases. Since our objective is to keep the latency minimal, the use of a stereo vision sensor is necessary as it provides instantaneous 3D measurements according to the method presented in Chapter 2. With that in mind, this chapter presents an event-based solution for visual odometry, using a binocular vision system. It is directly streamlined with Chapters 1 and 2, as it feeds on the direct results of both building blocks. The idea, once again, was to use a continuous, event-based inertial system, to estimate the pose, formalized here as a mechanical problem.

3.2 Proposed method

3.2.1 Pose Estimator

a) From solution optimization to continuous mechanical solving

Batch optimization processes are widely used, robust methods, that present unique and theoretically optimal solutions to often complex problems. By aggregating data over time, it is possible to infer the optimal solution to a highly non linear problem, often at the cost of latency and energy consumption. It was only natural for computer scientists and robotics engineers to use this type of algorithm for visual odometry. Moreover, GPUs usage optimizes this type of computation, making online visual odometry or SLAM possible. However, as we have already shown many times, this optimization paradigm hardly fits the neuromorphic realm. Events are elementary bits of data, and the only solution to apply these methods is to aggregate them into spatio-temporal volumes. It often is done by generating frames, and doing what has already being done with conventional cameras. It implies losing the benefits from both neuromorphic and conventional sensors, and thus seems doomed to underperform.

From a physics point of view, the error minimized in those processes can be seen as energy. In our case, the link between the problem to solve - i.e the pose of a visual system - and the data itself is straightforward. The cameras are a physical system, on which constraints are applied : continuous speed and position, rigidity between both cameras of the stereo rig, and continuity of the observed scene. Thus, instead of finding the minimum-energy state by doing batch optimization, we propose to develop an equivalent virtual system reproducing the constraints dictated by visual inputs. The gradient descent processes that would be performed on the energy hypersurface is thus translated into a virtual model, on which we apply forces and torque on an event-by-event basis. Using such a system has many advantages, among which a continuity of the solution, the simplicity of the implementation, but also the small cost for each update, resulting in lower expected latency. But this kind of solver is also a consequence of the nature of events themselves.

b) Events as infinitesimal time steps

A dynamical physics emulator is driven by a continuous loop, allowing constraints to act upon the simulated system as time goes on. A critical setting for such a dynamical system is the simulation time step or time discretization value δt. That value has to be small compared to any time constant related to the simulated physics in the engine, for non linear phenomenons to be accurately simulated through approximated series. However, setting this value too low implies a large amount of needless computations to be performed, making the solver slow and under-efficient. Methods exist to estimate that value δt during runtime, however those can sometimes be computationally costly [96].

In our case, the data fed to this solver is triggered by changes, and as a consequence is naturally bounded to the dynamics of the system. In case of an almost still stereo-rig, few events are produced, and the time between two events is large. In such a case, a simulation would not require a small time step to accurately simulate the ongoing motion. Oppositely, in case of a fast moving object, more steps per time unit are needed to recreate the conditions of the simulation. We hypothesize that the time between two successive events is ideal as the time step for such a dynamical approach to visual odometry. We thus chose to use those events as discretization time steps for our mechanical solver, whose characteristics we will develop in the following sections.

3.2.2 Physical solver model

We describe in this section the model used in our solver. Preliminary results will be presented in section 3.3. We use the previous works, described in Chapters 1 and 2, as primary visual inputs. Other visual inputs should be used in a more advanced work, especially trying to add loop closure or *keypoints* recognition. However, we focus here only on the pose estimation of the system, and decided to rely on the only data we were able to produce through fully event-based methods developed and detailed in the corresponding chapters.

We assume a number of sensors $\mathcal{S}$, rigidly linked to the rig $\mathcal{R}$. We also define the location and orientation vectors of each sensor $\mathcal{S}$ as:

- $\boldsymbol{X}_{\boldsymbol{\mathcal{S}}}$ is the $3D$ location of sensor, relative to its starting point.
- $\boldsymbol{\theta}_{\boldsymbol{\mathcal{S}}}$ is the axis-angle representation of the sensor rotation.

Both are defined relative to a frame initialized when data first arrives to the system. We denote their respective time derivatives $\dot{\boldsymbol{X}}_{\boldsymbol{\mathcal{S}}}$, $\dot{\boldsymbol{\theta}}_{\boldsymbol{\mathcal{S}}}$ and accelerations $\ddot{\boldsymbol{X}}_{\boldsymbol{\mathcal{S}}}$ and $\ddot{\boldsymbol{\theta}}_{\boldsymbol{\mathcal{S}}}$. Notice that we don't specify the sensor here. In the later development, the equations are applied for any of the sensors of the visual system. The rig $\mathcal{R}$ motion parameters are updated, by summing the different constraints later developed, taking into account the offset of each sensor w.r.t the rig center. Also, all time dependencies will be omitted to alleviate the notation load. If need be, a vector can be specifically defined in the static world reference frame $\mathcal{W}$ or in a sensor $\mathcal{S}$ frame.

3.2.3 Viscous fluid environment

As seen in Chapter 2, a purely stereoscopic visual input already allows for a precise, event-by-event ego-kinetics estimation. This data processing can be stable enough to be integrated over, giving a good estimate of the location of the camera over several seconds-long sequences, with no additional data. Thus, we use this information as one primary input source for our virtual solver.

To incorporate this velocity estimate, we emulate a viscous fluid environment in which is embedded our simulated stereo-rig. This environment velocity is set through this previously estimated ego-kinetics, and acts upon our virtual model. It can thus impact the general trend of the sensor's velocity, while not forcing both systems velocities to match. The benefits of using a viscous dampening are numerous. First of all, using such a dampening allows for a first order correction of the velocity. This automatically averages the visual ego-kinetics estimate, canceling out noise, while reaching the desired velocity over a controlled time scale. Second, a viscous interaction between the stereo-rig and the surrounding environment dissipates energy. That is a main issue with this kind of solver, as energy input from mechanical constraints can make the system diverge if not dissipated fast enough.

Let us define $\tilde{\boldsymbol{v}}_{\boldsymbol{\mathcal{S}}}$ - respectively $\tilde{\boldsymbol{\omega}}_{\boldsymbol{\mathcal{S}}}$ - the sensor velocities from an external estimator, like the one developed in Chapter 2.

The translation constraint $\boldsymbol{F}_{\boldsymbol{\mathcal{S}}}$ and rotation constraint $\boldsymbol{\Gamma}_{\boldsymbol{\mathcal{S}}}$ generated by fluid friction on sensor $\mathcal{S}$ is then

$$\begin{aligned}\boldsymbol{F}_{\boldsymbol{\mathcal{S}}} &= \mu\left(\tilde{\boldsymbol{v}}_{\boldsymbol{\mathcal{S}}} - \boldsymbol{v}_{\boldsymbol{\mathcal{S}}}\right)\\ \boldsymbol{\Gamma}_{\boldsymbol{\mathcal{S}}} &= \nu\left(\tilde{\boldsymbol{\omega}}_{\boldsymbol{\mathcal{S}}} - \boldsymbol{\omega}_{\boldsymbol{\mathcal{S}}}\right)\end{aligned} \tag{3.1}$$

with μ and ν two dampening constants whose values will be estimated later in section 3.2.6. In case of multiple sensors, we average those constraints, for the system to use a maximum amount of data while keeping constraints amplitudes independent of the number of sensors involved.

3.2.4 Trackers and depth as stabilization data

The second visual input are trackers, used to stabilize the estimated location in space. Indeed, using only the velocity with the viscous model as described before would lead in the best case scenario to valid visual odometry delayed in time, but most likely in a continuously increasing drift of the virtual rig pose, as error in velocity gets integrated over time. In other methods, this type of visual input is often referred to as *keypoints*, as they describe important parts of the scene that the algorithms refers to in order to recover the pose of the system. Those *keypoints* are also often used for loop closure, as one can hope to match them and confirm the location of the system after some time. However, this later part has been left aside, and we only focus on the local tracking of those points.
More information can be used, apart from on-screen trackers. Working with binocular visual input allows to recover depth information. By coupling it to tracking information, those *keypoints* can be estimated as points in $3D$ space, rather than lines of sight.

a) Springs binding trackers to stereo rig model

The question then becomes how to emulate those constraints to accurately reproduce this tracker reprojection and estimated depth error. Such errors can be represented by a spring, in which case the least squared error is mathematically equivalent to finding the minimum-energy state of the spring. Such springs models have already been used in event-based processing, with a similar philosophy [97].
Most bundle-adjustment algorithms minimize a reprojection error, which is a two-dimensional constraint. It means that only the line of sight is considered, for the current projected point of a $3D$ object to match the expected $2D$ point on-screen location. This can be seen as a rotation spring, generating a torque aligning two intersecting lines. However, several drawbacks appear from this initial approach.
First of all, this technique does not take depth into account. It means that less data is used by the solver, inducing a waste of information. Also, such a torque would be very efficient for the rotational component, but very inefficient for the translation component of the pose we want to recover. If the translation component is expected to be harder to find - rotation-only pose estimators are easier to implement, and more efficient in computation speed - this central torque model makes it even harder. Thus, we have to create a *keypoints* constraint model which takes depth into account, and can act directly on all motion parameters.

Let us assume each sensor $\mathcal{S}$ presents a set of trackers $T \in \mathcal{T}_{\mathcal{S}}$, that can appear or disappear over time. Each tracker is described by a set of coordinates on the sensor focal plane (x_T, y_T), and an estimated depth of the tracked object D_T. Again, we omit the time notation, as we focus first on the instantaneous physical parameters of the *keypoints* and their effects on the system.

b) *Keypoint* initialization

With (x, y) being a screen location and D a depth, we define H the transformation from the triplet (x, y, D) to the $3D$ location $\boldsymbol{X}$.

$$\begin{aligned} H : \mathbb{R}^2 \times \mathbb{R}^+ &\to \mathbb{R}^3 \\ (x, y, D) &\mapsto \boldsymbol{X} = H(x, y, D) \end{aligned} \tag{3.2}$$

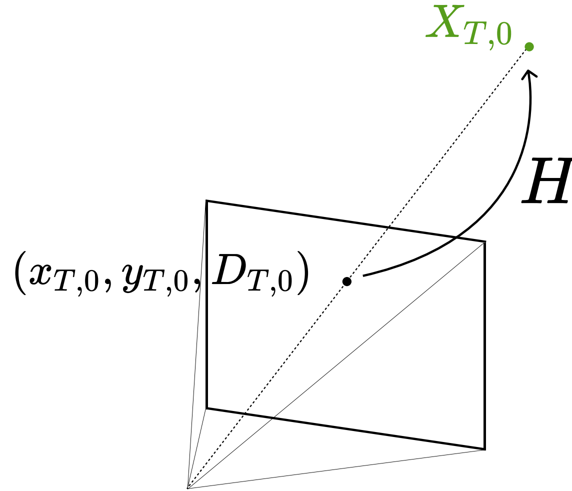

Figure 3.2 *Visualization of a keypoint initialization from a tracker T. The tracker is started at location $(x_{T,0}, y_{T,0})$ and with an estimated depth $D_{T,0}$. It is mapped to a 3D location $\boldsymbol{X_{T,0}}$ with H. Those keypoints will be used to stabilize motion estimation in space.*

Let us assume a tracker being initialized at location $(x_{T,0}, y_{T,0})$, and the depth $D_{T,0}$ at this location given through a prior algorithm, such as the one described in section 2.2.2. This initializes a *keypoint* in space the stereo-rig will refer to, at location $\boldsymbol{X_{T,0}} = H(x_{T,0}, y_{T,0}, D_{T,0})$. We assume that all objects in the scene remain still during sensors motion. Thus all *keypoints* $\boldsymbol{X_{T,0}}$ are static in frame $\mathcal{W}$. This initialization can be visualized in Figure 3.2.

c) Trackers constraint principle

Now that *keypoints* have been initialized, we use them as stabilization anchors for our system, through simulated springs links. Let us first focus on a single *keypoint* constraint. Like previously, we note $\boldsymbol{X_T}$ the estimated location of a tracked point in space. We define $\boldsymbol{P_T}$ the orthogonal projection of the initial keypoint $\boldsymbol{X_{T,0}}$ on the line of sight of sensor $\mathcal{S}$ going through $\boldsymbol{X_T}$. All corrections created by one tracker will rest in the plane formed by the three points $\boldsymbol{X_T}$, $\boldsymbol{P_T}$ and $\boldsymbol{X_{T,0}}$, which can be visualized in Figure 3.3.

From these points, we define $\delta_\perp$ the projection distance, and $\delta_{/\!/}$ the distance between $\boldsymbol{P_T}$ and $\boldsymbol{X_T}$:

$$\begin{aligned} \delta_\perp &\equiv ||\boldsymbol{P_T} - \boldsymbol{X_{T,0}}|| \\ \delta_{/\!/} &\equiv ||\boldsymbol{P_T} - \boldsymbol{X_T}|| \end{aligned} \tag{3.3}$$

We also define $\boldsymbol{u}_\perp$ the orthogonal projection unit vector, and $\boldsymbol{u}_{/\!/}$ the unit vector for this line of sight:

$$\begin{aligned} \boldsymbol{u}_\perp &\equiv \frac{\boldsymbol{X_{T,0}} - \boldsymbol{P_T}}{\delta_\perp} \\ \boldsymbol{u}_{/\!/} &\equiv \frac{\boldsymbol{P_T} - \boldsymbol{X_T}}{\delta_{/\!/}} \end{aligned} \tag{3.4}$$

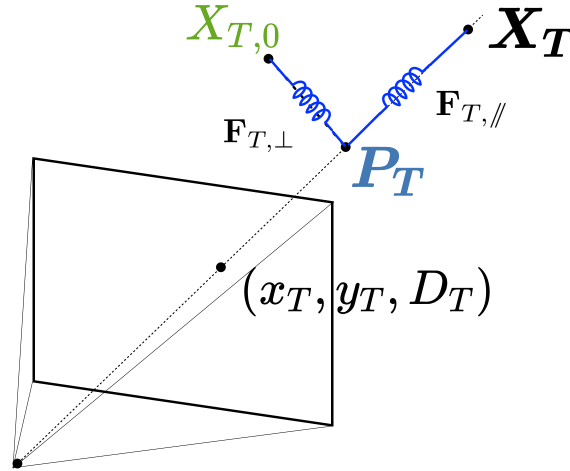

Figure 3.3 *Visualization of the anchors' 3D points $\boldsymbol{X_T}$, $\boldsymbol{P_T}$ and $\boldsymbol{X_{T,0}}$, along with the two correction contribution from this particular keypoints $\boldsymbol{F_{T,\perp}}$ and $\boldsymbol{F_{T,/\!/}}$*

Each tracker T will generate two different forces $\boldsymbol{F_{T,\perp}}$ and $\boldsymbol{F_{T,/\!/}}$, with directions $\boldsymbol{u}_\perp$ and $\boldsymbol{u}_{/\!/}$, and application points $\boldsymbol{P_T}$ and $\boldsymbol{X_T}$ respectively. Their general expression is given by:

$$\boldsymbol{F} \equiv \kappa\delta\boldsymbol{u} \tag{3.5}$$

κ being the constant of the spring, which value will be discussed in section 3.2.6. Distinguishing those two contributions has two advantages. The first one is that both contributions have different origins. $\boldsymbol{F_{T,\perp}}$ will be mostly impacted by the reprojection error onto the focal plane, while $\boldsymbol{F_{T,/\!/}}$ will increase with the estimated object's depth error. As such, we can set different repel constants for each contribution, and the trust the system can place in the respective input values. Also, in many cases, tracking an object and estimating its depth are two different processes our system relies on. It is reasonable to assume that, for the same *keypoint*, one can perform well while the other fails. This separation allows to have only partial contributions to some extent. For example it is possible to rely purely on the reprojection error if the depth estimation fails. In this case, we keep only the orthogonal contribution $\boldsymbol{F_{T,\perp}}$, by locally setting $\kappa_{/\!/} = 0$. The transformation H must be modified accordingly, to produce only the line of sight $\boldsymbol{u}_{/\!/}$, which does not need the depth to be recovered.
We note that as long as the repel force is linear and repel constants are the same, the two orthogonal springs are mathematically equivalent to a single spring joining points $\boldsymbol{X_T}$ and $\boldsymbol{X_{T,0}}$.

d) Springs force expression

The usual expression for a physical spring given in equation 3.5 have already been used in related works [98]. The force grows linearly with length, and the spring stores a quadratic-form potential energy.

$$\begin{aligned} f(\delta) &= \kappa\delta \\ E(\delta) &= \frac{1}{2}\kappa\delta^2 \end{aligned} \tag{3.6}$$

κ being the repel constant of the spring.

In our case, using a standard spring force expression could be harmful to the system stability, when we consider tracking or depth estimation outliers. As explained in the beginning of this chapter, the goal is to stay as much as possible in the minimum-energy state, for visual odometry to remain correct. As no conventional optimization is performed on the system, getting out of the global energy minimum could result in getting stuck in a local one. This would result in wrong results, an unpredictable behaviour and a probably unrecoverable situation.

As usual spring energy expression is not bounded, each tracker can theoretically feed an infinite amount of energy to the system. Removing an obviously failing tracker can be done, resulting in a large variation of the energy of the system, but also of the forces applied. The later the removal of an outlier, the greater the resulting variations.

To counter that, we implement a trust factor for each tracker, that we define as $w(\delta) = e^{-\delta^2/L^2}$. To picture its impact, let us consider a spring force implementing this trust factor as such:

$$f(\delta) = \kappa \delta e^{-\delta^2/L^2} \tag{3.7}$$

While we keep the spring constant κ, we add a parameter L which is the standard deviation allowed for *keypoints*. This deviation L is a distance in $3D$ units, and thus can be physically understood. It expresses the typical length over which a *keypoint* validity can be doubted, and thus should reach its maximum energy state. The continuous and differentiable properties of this expression allows for smooth and continuous forces on the system, helping to maintain its stability. The comparison of the force and energy for the standard and modified spring expression can be visualized in Figure 3.4.

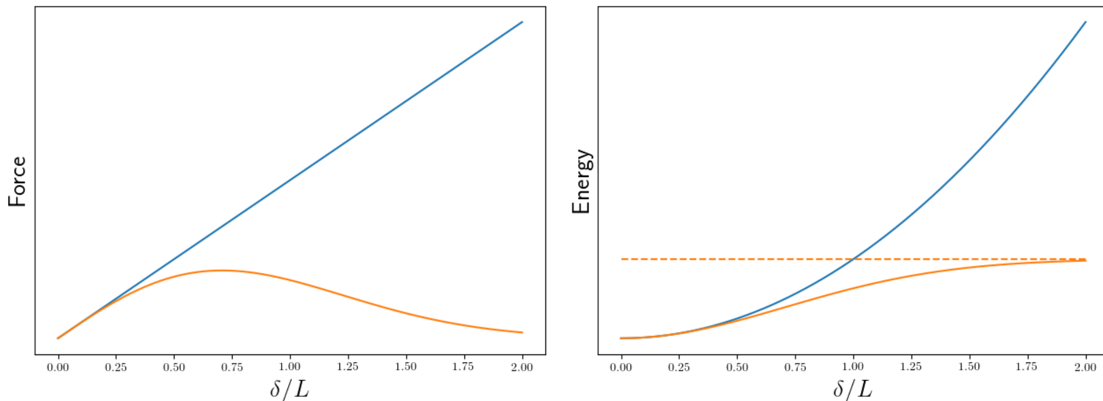

Figure 3.4 *Comparison of forces (left) and energies (right) for two springs repel forces expressions. The standard one - in blue - in unbounded in energy, and thus can lead to system instability. The proposed one - in orange - adds an exponential dampening, which constrains the energy to a certain range.*

With the proposed force expression 3.7, the potential energy for each spring is then given by

$$E(\delta) = \frac{1}{2}\kappa L^2 \left(1 - e^{-\delta^2/L^2}\right) \tag{3.8}$$

bounding the potential energy to the range $[0, \kappa L^2/2]$[1]. Notice that while $\delta << L$, the forces have similar behaviour, and most of the properties of springs dynamics remain valid[2]. However, when $\delta >> L$, the force becomes null. This implies that the failing *keypoint* can be removed, resulting in restricted potential energy reduction and force variation. This allows adding or removing trackers in an online fashion while leaving the system in a stable state.
Another advantage of using separate orthogonal and parallel contributions in this case is that the trust factor we implement can apply differently to a *keypoint* for reprojection and depth errors. The two contributions trusts are gauged separately, inducing a maximum data usage at all times. In this case, the two separate springs are not equivalent anymore to a single spring joining points $\boldsymbol{X_T}$ and $\boldsymbol{X_{T,0}}$ like before.

As L describes a real-world distance, it is reasonable to set a similar value for both perpendicular and parallel forces described above, keeping the number of parameters low. It is set manually, depending on the estimated allowed $3D$ error for each point.

e) System Equivalent Constraint

With the model described so far, the natural idea to aggregate these contributions $\boldsymbol{F_T}$ for each tracker is to sum them all. If the sum force can be done as such, a torque can be extracted by using the corresponding application points $\boldsymbol{X_T}$ and $\boldsymbol{X_{T,0}}$. However, this leads to unexpected, or even counter-intuitive behaviour of the system. Indeed, let us imagine two systems $\mathcal{R}_1$ and $\mathcal{R}_2$, each perfectly solved with respectively $N_1 > N_2$ trackers, feeding noiseless information to our solvers. Summing the contributions would lead to system 1 converging faster than system 2, even though both receive perfect information to solve the pose. The resulting amplitude of *keypoints* contributions have to be independent of the tracker number, as that number should improve correction quality, and not constraints amplitudes.

Thus, we need to average those contributions, and find the optimal correction that best fits the information we are given. However, simply averaging those contributions and applying them to the rig also hardly makes sense. Examples can be handmade where the average force - which creates translation - is null, while a translation is needed. Inversely, zero torque and non-zero average force can be extracted when the necessary displacement is purely rotational. We picture such cases in Figure 3.5, in $2D$ cases for clarity. We describe in this section the solution implemented to properly aggregate those contributions into a *system equivalent constraint* applied to the stereo rig, using the trust factor described in the previous section.

Let us describe the rig - i.e the physical rigid body holding 1 or more sensors - and the lines of sight carrying all application points $\boldsymbol{X_T}$ and $\boldsymbol{P_T}$ as an entire, locally rigid body[3]. Thus, all *keypoints* contributions $\boldsymbol{F_T}$ are constraints applied onto a single rigid body. This means that any point $\boldsymbol{X}$ of our rigid body perceives equivalent translation constraint $\boldsymbol{F}(\boldsymbol{X})$ and rotation constraint $\boldsymbol{\Gamma}(\boldsymbol{X})$ such that:

[1] We set a null potential energy for an infinitely small spring.

[2] The motion of an oscillator with this repel force expression is not a perfect sine wave anymore. However, the system remains oscillatory, and an equivalent frequency can be extracted from numerical simulations. For extreme amplitudes of $\delta_0 = L$, a 45% frequency change is observed. The order of magnitude of frequencies in oscillatory regimes stays the same with this modified repel force expression compared to a standard harmonic oscillator.

[3] Lines of sight change over time as trackers move on screen. However, at any time t, those line of sight can be perceived as infinitely rigid links, fixed w.r.t the rig.

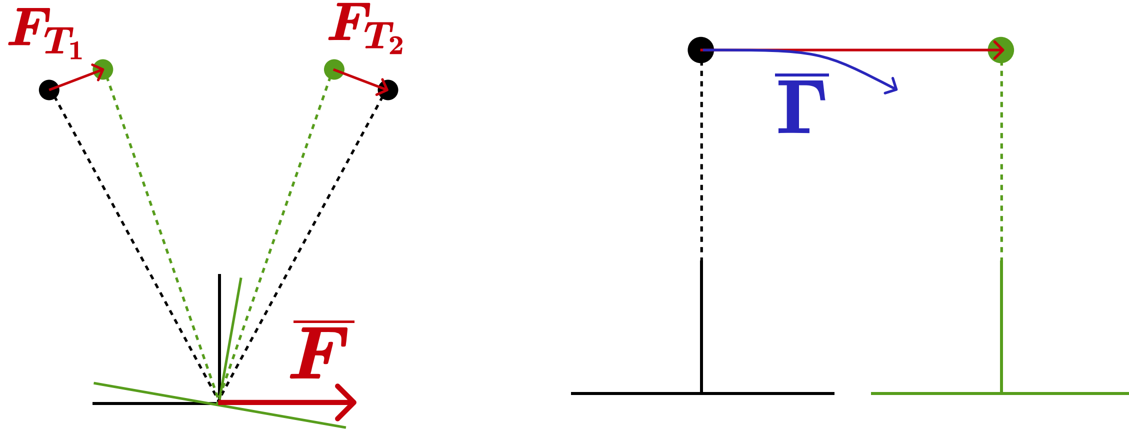

Figure 3.5 *Examples where averaging individual keypoints contributions lead to wrong corrections. The current estimated pose is pictured in black, while the theoretical pose is in green. They are voluntarily oversimplified for clarity. Left : The average contribution of two trackers result in a non-zero average force, while only rotation - and thus only torque - was needed. Right : A single keypoint contribution leads to rotation, while pure translation was required.*

$$\begin{aligned} \boldsymbol{F}(\boldsymbol{X}) &= \boldsymbol{F_0} + (\boldsymbol{X} - \boldsymbol{X_0}) \times \boldsymbol{\Gamma_0} \\ \boldsymbol{\Gamma}(\boldsymbol{X}) &= \boldsymbol{\Gamma_0} \end{aligned} \tag{3.9}$$

with $\boldsymbol{X_0}$ being any point in space used as the center of rotation, $\boldsymbol{F_0}$ the equivalent force applied to this point, and $\boldsymbol{\Gamma_0}$ the common rotation constraint of the body[4]. In this vision, each spring force $\boldsymbol{F}$ and its application point $\boldsymbol{X}$ is a measurement of this constraint field we want to characterize. Such a field can be visualized in a $2D$ representation in Figure 3.6.

To recover our parameters $\boldsymbol{F_0}$ and $\boldsymbol{\Gamma_0}$, we take advantage of the trust factor $w_T = w(\delta)$ defined in section 3.2.4.d. Used as weight, it allows to evaluate our constraint field taking into account potential outliers for tracking or depth estimation. We use the weighted average measurement center as origin, as it can be chosen arbitrarily:

$$\begin{aligned} W &\equiv \sum_T w_T, \\ \boldsymbol{X_0} &= \frac{1}{W} \sum_T w_T \boldsymbol{X_T}, \\ \boldsymbol{F_0} &= \frac{1}{W} \sum_T w_T \boldsymbol{F_T} \end{aligned} \tag{3.10}$$

Next, we compute $\boldsymbol{\Gamma_0}$ by minimizing $E \equiv \sum_T w_T ||\boldsymbol{F_T} - \boldsymbol{F}(\boldsymbol{X_T})||^2$, keeping in mind that F is a function with parameters $\boldsymbol{X_0}$ arbitrarily chosen, $\boldsymbol{F_0}$ computed above, and $\boldsymbol{\Gamma_0}$. With $\boldsymbol{\delta F_T} \equiv \boldsymbol{F_T} - \boldsymbol{F_0}$

[4]We do not refer here to this constraint as a torque, as in terms of physical unit, $[\boldsymbol{\Gamma}] = N.m^{-1}$, which is not a usual torque.

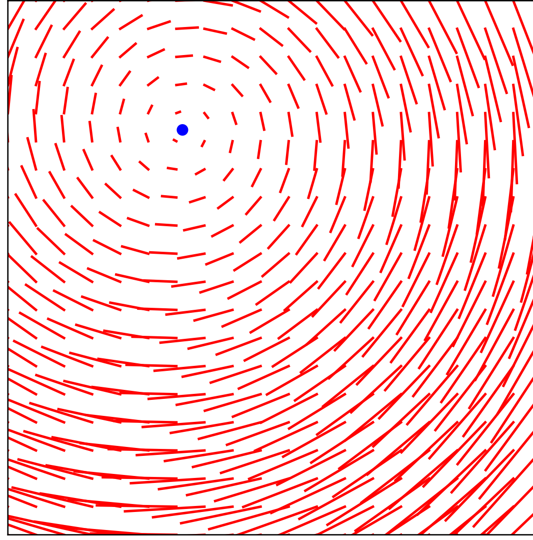

Figure 3.6 *Visualization of a constraint field as defined in equation 3.9. The blue dot represents the location where the force is null, in our case the center of rotation of the visual system. Each spring in our model is a measurement of this field, and its characterization allows us to apply a unique constraint in translation and rotation to the virtual model.*

and $\boldsymbol{\delta X_T} \equiv \boldsymbol{X_T} - \boldsymbol{X_0}$, we obtain the following expressions:[5]

$$\begin{aligned} \boldsymbol{\Gamma_0} &= M^{-1}\boldsymbol{\Sigma}, \text{ with} \\ M &= \sum_T w_T \left(||\boldsymbol{\delta X_T}||^2 \mathbb{1} - \boldsymbol{\delta X_T} \otimes \boldsymbol{\delta X_T} \right), \\ \boldsymbol{\Sigma} &= \sum_T w_T \boldsymbol{\delta F_T} \times \boldsymbol{\delta X_T} \end{aligned} \tag{3.11}$$

Finally, the equivalent rig constraints are computed according to this computed field. If the rotation component is the same in all points of space, we use the rig center $\boldsymbol{X_{\mathcal{R}}}$ as application point for our forces:

$$\begin{aligned} \boldsymbol{F_{\mathcal{R}}} &= \boldsymbol{F_0} + (\boldsymbol{X_{\mathcal{R}}} - \boldsymbol{X_0}) \times \boldsymbol{\Gamma_0} \\ \boldsymbol{\Gamma_{\mathcal{R}}} &= \boldsymbol{\Gamma_0} \end{aligned} \tag{3.12}$$

We use these values into our dynamical simulator, both in rotation and translation. This allows for a single expression for all trackers, stating the overall pose variation needed to bring keypoints closer to their initially set locations.

3.2.5 Physical model equations

The previous sections have detailed the two different contributions uses to a visual odometry. The first contribution affects the virtual rig on a first order, through velocity error, and uses a previously estimated ego-kinetics of the sensors. The second ones fuses objects reprojection error and estimated depth error into an overall pose correction. These contributions are displayed in mechanical symbols in figure 3.7, in the case of a 2-sensors system.

[5] $\mathbb{1}$ defines the 3×3 identity matrix, while $\otimes$ defines the outer product of two vectors.

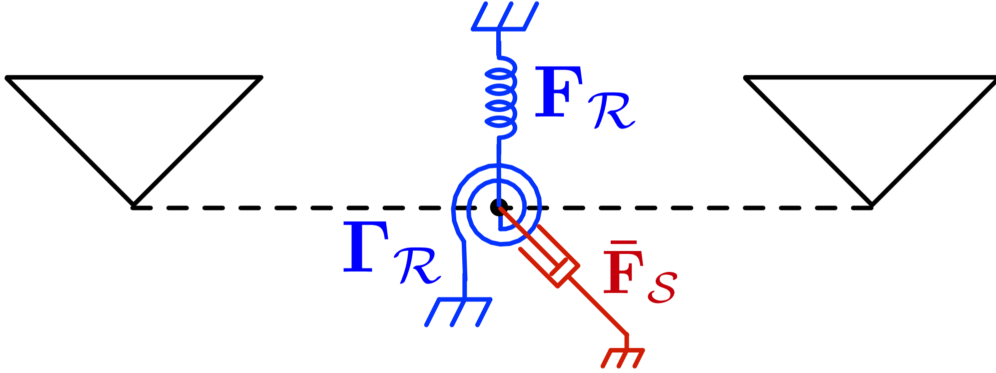

Figure 3.7 *2D representation of the different contributions used to update the visual odometry. The keypoints contributions - in blue - consist in a translation contribution $\boldsymbol{F}_{\mathcal{R}}$ and a rotation contribution $\mathbf{\Gamma}_{\mathcal{R}}$. The translation fluid contribution - in red - is displayed here as $\bar{\boldsymbol{F}}_{\mathcal{S}}$. To avoid overloading the schematics, its rotational counterpart is not shown.*

Assuming N_S sensors in our model ($N_S = 2$ for a binocular system, however the system can use any number of sensors), we average the velocity contributions of each sensor, and compute the overall spring constraint field as described above. The reason for averaging velocity contributions is the same as previously stated : more sensors should not results in higher forces and torques amplitudes, rather than more accurate ones, allowing to converge faster both in pose and system velocities. As all sensors are assumed equals in terms of physical properties, constraints defines in 3.1 and 3.12 lead to the main physical equations 3.13.

$$\begin{aligned} M\ddot{\boldsymbol{X}}_{\mathcal{R}} &= \frac{1}{N_S}\sum_{\mathcal{S}} \boldsymbol{F}_{\mathcal{S}} + \boldsymbol{F}_{\mathcal{R}} \\ I\ddot{\boldsymbol{\theta}}_{\mathcal{R}} &= \frac{1}{N_S}\sum_{\mathcal{S}} \mathbf{\Gamma}_{\mathcal{S}} + \mathbf{\Gamma}_{\mathcal{R}} \end{aligned} \tag{3.13}$$

with M the system virtual mass, and I its inertial moment. Their actual values don't have to match the system physical ones, as we will show in section 3.2.6. We assume the whole mass to be located at the center of mass of the system, i.e at the rig center location which defines our recovered pose. Figure 3.8 shows an overview of the visual odometry pipeline, with the inputs and subsequent contributions, in the case of a 2-sensors system.

3.2.6 Parameters estimation through oscillator stability

We present in this section a stability study on the system defined above. Common dangers with simulating an inertial system such as this one are either oscillations instability or response latency after a stimulus. The system virtual properties have to be set according to oscillators physics in order to get the best response to the system input, i.e. a fast response in position and speed to the correct values. To that end, we need to perform stability study of our system. As previously stated, the modification performed on the repel force expression allows to protect the system to outliers, but leaves the usual oscillator response valid while near the minimum energy state.
However, the complexity of the system forces us to use a simplified model, to extract general values for M, I, κ, μ and ν. We assume that the distances δ involved remain low compared to L, and simplify the system to a $1D$ oscillator of coordinate x. In this case, the governing equation

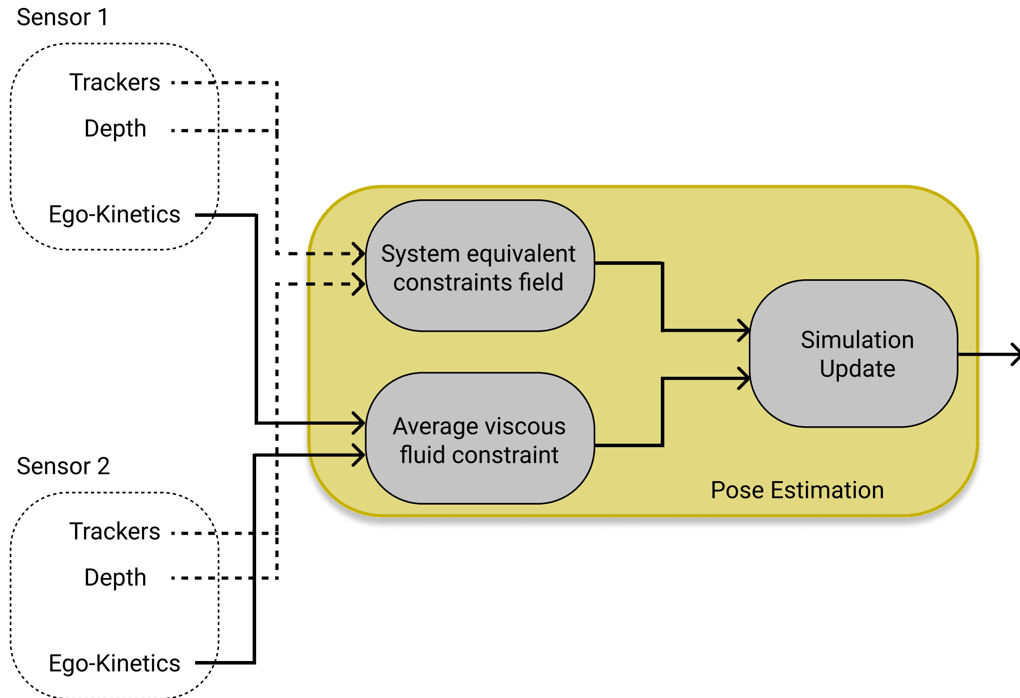

Figure 3.8 *Overview of a 2-sensors visual odometry pipeline. Each sensors can provide ego-kinetics, tracking information and depth estimation. Each contribution is used in a physical simulation, aggregating the constraints and minimizing the overall energy of the system.*

becomes

$$M\ddot{x} = -\kappa x - \mu\dot{x} \tag{3.14}$$

We define the springs pulsation $\omega = \sqrt{\kappa/M}$ and dampening ratio $\zeta = \frac{\mu}{2\sqrt{\kappa M}}$, and we write the previous equation as

$$\ddot{x} + 2\zeta\omega\dot{x} + \omega^2 x = 0 \tag{3.15}$$

In this type of equation, we want $\zeta \sim 1$ for the system to be critically dampened. To ensure that critical dampening, we set μ as a function of the other parameters:

$$\zeta \sim 1 \iff \mu \sim 2\sqrt{\kappa M} \tag{3.16}$$

Next, we set the springs repel constant κ relative to the system mass M to have a small response time w.r.t the observed scene dynamics. Visual odometry is mainly used in robotics, autonomous vehicles, and human motion. In most cases, the typical motion will take place on a timescale $\tau_{scene} \sim 1s$. The system needs to reach the minimum energy state after a sudden dynamics change over a duration small compared to this timescale. If the system is still stabilizing while another change appears, energy excess could build up, to the point of getting out of the global minimum energy well.

To meet this constraint, we set the spring repel constant such that the typical response time τ_{system} is small compared to the scene timescale τ_{scene}:

$$\begin{aligned} \tau_{system} \equiv \frac{2\pi}{\omega} \sim \tau_{scene}/10 \\ \iff \kappa \sim \frac{4000M}{\tau_{scene}^2} \end{aligned} \tag{3.17}$$

Finally, we define rotational counterparts I and ν from a dimensional study. As we use a rotational constraint (denoted $\Gamma_{\mathcal{R}}$ in section 3.2.4.e) lengthless, the rotation equations units do not involve any length; both mass parameters and fluid viscosity show similar unit in translation and rotation.

$$\begin{aligned} [I] = [M] = kg \\ [\nu] = [\mu] = kg.s^{-1} \end{aligned} \tag{3.18}$$

From this observation, we can set $I \sim M$ and $\mu \sim \nu$, giving the system similar response times in translation and rotation.

Using those constraints, we report in table 3.1 the system parameters values used in our experiments.

Parameter	Value	Unit
M	1	kg
I	1	kg
κ	4000	$kg.s^{-2}$
μ	125	$kg.s^{-1}$
ν	125	$kg.s^{-1}$
L	0.3	m

Table 3.1 *Pose solver parameters values used throughout our experiments. They result from a stability study over a simplified* 1*D harmonic oscillator, but have produced the expected results in terms of visual odometry convergence.*

3.3 Preliminary results

Results of our pose estimator are reported in this section. These results are preliminary, as time and data constraints made more in-depth results hard to acquire. First of all, our visual odometry does not rely on usual sensor events, but on previously analyzed data, transformed into ego-kinetics, objects tracking on screen, and depth estimation. We had to wait the maturation of previous algorithms for us to know what this later algorithm could rely on. This, along with initial trials to show the correctness of the implementation, made the use of synthetic data necessary.
More crucially, the ground-truth comparison have been a difficult task, as recovering the baseline position of a stereoscopic system requires hardware unavailable to us at that time. This further pushed the need for synthetic data, reported in this section.
If additional self-recorded sequences were tested, issues with on-screen tracking made input values, and thus estimated pose, unreliable. Thus, these results will not be reported in this document, however we expect that future works will allow to fully benchmark our algorithm on real-world data.

3.3.1 Data Simulator

We developed a basic event-based simulator, that emulates the motion of a 2-sensors rig. It is placed inside an environment in which *generators* create sensors events, and are being tracked along with depth information. The simulator also transmits ego-kinematics values for each sensor, and thus allows to have complete control on the data fed to our visual odometer. The size of the virtual environment matches the typical size of common environments used in robotics, ranging from a few meters to about $100m$. Such motion can be visualized in figure 3.9.

Notice that our pose estimator does not use sensor events information *per se*, rather than

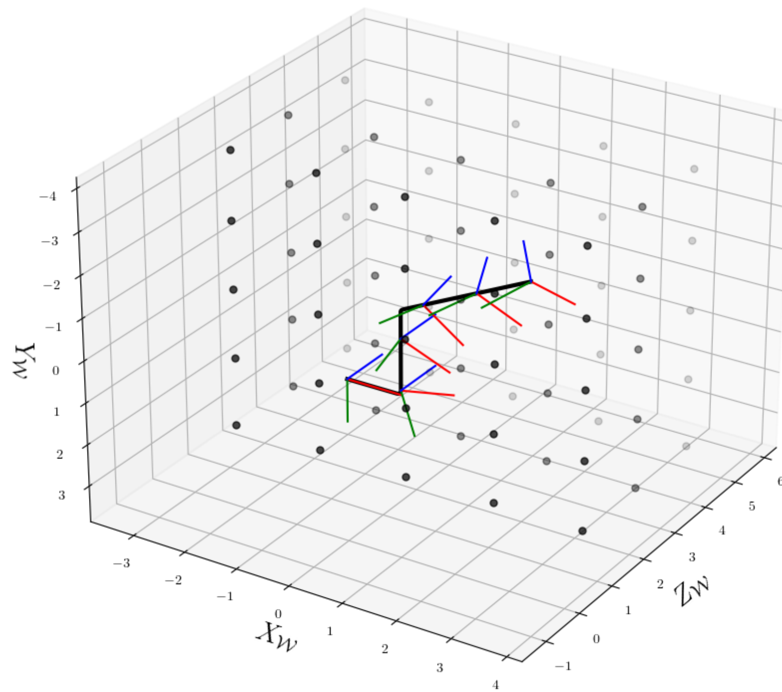

Figure 3.9 *Example of the motion - black line - used in these experiments, visualized in* $3D$*. In this case, no translation occurs along axis* Z*, which is the initial vision axis. The sensor axis are displayed at* 6 *different occasions. Blue: Forward vision axis on the sensor. Red: Horizontal axis of the sensor. Green: Vertical axis of the sensor. Each black sphere is a generator or our simulator, from which are extracted camera events, trackers and depth information.*

pre-processed higher-level information. Camera events are merely a support for the simulator time steps. This feat allowed us to use a simulator with little regard for the quality of the sensor events generated, used in our case for visualization and display. We only care for the number of events per time unit, set to $10^5 ev/s$. This number is set to meet the order of magnitude usual neuromorphic visual sensors event rate [99, 11].

3.3.2 Noiseless results

Using such raw, noiseless data led to encouraging results, following the expected dynamics of a dampened oscillator. The error in translation and rotation of this experiment can be visualized in Figure 3.10. The simulated motion is a sequence of different movements of various amplitudes affecting one or more degrees of freedom of the virtual rig. We can observe the errors locally

increasing when the motion dynamics change. This is due to need of dissipating energy excess - typically kinetic energy - between two successive motion patterns. If a theoretically infinite time is needed to completely remove this error, parameters described in section 3.2.6 allow to suppress the majority of it within a $\tau_{scene} \sim 1s$ time window.

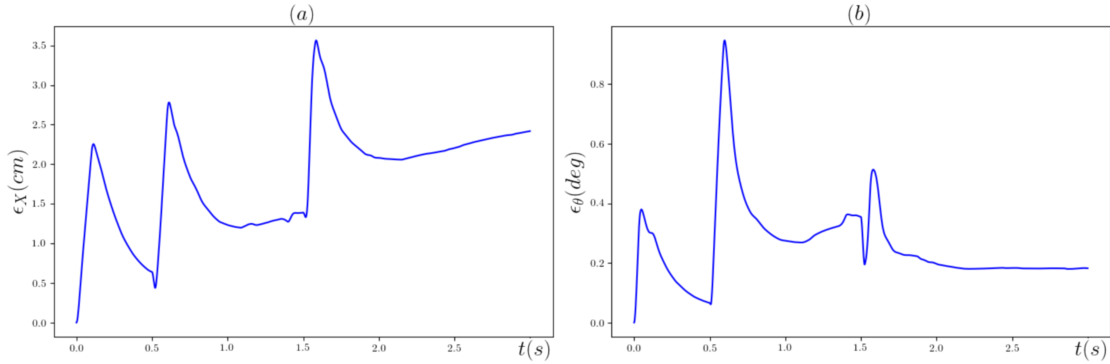

Figure 3.10 *Odometry error for a noiseless synthetic motion, in translation (left) and rotation (right). The overall motion involves a* $6.07m$ *translation, with a maximum relative error of* 2.82%*, and a rotation of* $125deg$*, peaking at* 3.6% *relative error. Both maximum relative errors occur at* $t \sim 0.7s$*, which is a time of large dynamics changes. More generally, for each of those dynamics change moment, the inertia of the system generates an error, that gets reduced over the typical scene timescale set* τ_{scene}.

The fact that the system presents a certain response time induces an error in the estimated $3D$ location of newly arrived trackers and *keypoints*. This error is then propagated through the pose correction process of the system, leading to a constant drift of the system that can be observed on the left graph. This is not unlike the usual $3D$ error reprojection of other visual odometers, emphasized by the response time of the initial system. A solution, succinctly explored, was to force the system to release energy upon *keypoint* initialization, and thus finding the actual optimal location of the sensor, regardless of the velocity. If this approach improved results, more work is needed concerning the stability and computation efficiency of this process, as it breaks free from infinitesimal computation steps the system is built upon.

These noiseless results still allowed to assert the proof of concept of our method, as well as the correctness of its implementation. However, having full control over the data fed to the system allowed us to implement noise in it, and thus estimate our pose estimator robustness limits.

3.3.3 Adding noise to synthetic inputs

If characterizing neuromorphic sensors noise is still an ongoing question [100, 101, 102], the noise associated with the data used in our system can be roughly estimated. Indeed, this data is macroscopic[6], often resulting in much more regular behaviour of their values and thus simpler noise models. Such models are necessary in the case of synthetic data to benchmark the input error our algorithm can tolerate.

[6] As opposed to infinitesimal events, carrying information about a single pixel.

Indeed, as previously explained, the main requirement here is to stay as close as possible to the minimum-energy state. If temporary increase in energy is necessary to set the virtual body in motion, it needs to stay in the global minimum well of mechanical energy for visual odometry to remain correct. An excessive input noise is the one reason that can take the virtual model outside of this minimum-energy location.

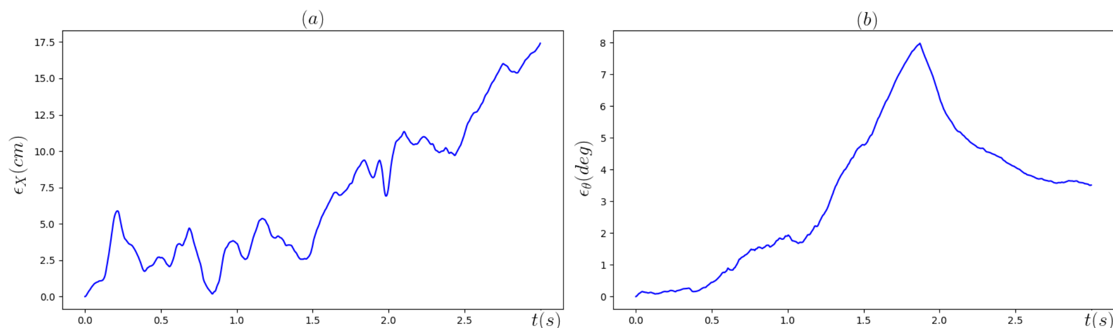

Figure 3.11 *Odometry error for a noised synthetic motion, in translation (left) and rotation (right). The theoretical motion is the same as in Figure 3.10, however each data input is noised according to models extracted from literature and existing algorithms typical errors. In this case too, changes in dynamics result in local errors peaks later dampened, however the increase in error over time is grater. The maximum relative error in this case is* 3.0% *in translation and* 8.5% *in rotation at the end of the experiment.*

We thus tried to characterize the behaviour of failing trackers, stereo-matching error for depth estimation and ego-kinetics estimation, by analyzing the output of algorithms described in the previous chapters, along with related works. We implemented from that analysis the following noise models:

- Trackers update asynchronously, and this update is based on events, generated by the movement of the tracked shape. An event located outside of the tracker ROI will not trigger an update of this tracker location.
- Trackers have a typical gaussian noise error, centered around the object they are following, of typical size $\sigma \sim 1px$[56].
- A tracker fails on average every $\tau \sim 0.3s$[56]. Depending on the algorithm at play, the failure will either be detected and the tracker disappears, or the tracker remains active, either fixing to another location in space, or drifting away at constant speed. We used the later, as it presents the biggest impact on our system response.
- The disparity presents a rounded normal distribution, centered around the theoretical value, with typical deviation $\sigma \sim 1px$.
- Ego-kinetics estimation such as the one presented in Chapter 2 are normal $3D$ distributions, centered around the theoretical value, with standard deviations scaling with the amplitude of the theoretical value : $\sigma_v \sim \rho||\boldsymbol{v}||$ and $\sigma_\omega \sim \rho||\boldsymbol{\omega}||$. In our case, we increase this ratio ρ up to $\rho = 0.5$ to check the robustness of our model to large variations of ego-kinetics estimations.

This noise was implemented in our simulator, and an experiment with similar motion parameters as in section 3.3.2 is performed. The results are reported in Figure 3.11. As before, we observe

peaks in error when a sudden change in dynamics occur. However, the translation error undergoes a much constant, steeper error curve, up to a maximum of $17.5cm$. The rotation error presents a high $8deg$ peak at the middle of the sequence that gets reduced over time.

3.4 Conclusion

Our model presents a new way to compute the pose of a visual sensors through a new, dynamical approach. Any number of sensors can be used, as long as depth can be estimated. If we used stereo-matching for the depth in our real-world experiment, other methods, such as defocus, could be used. Additionally, the model used does not rely on a specific number of sensors, and increasing this number should allow to further improve the correctness of the pose estimated.

This work is still in its infancy, as this model was developed during the latest stages of this PhD. Initial implementations, using a more simple model for *keypoints* correction led to unstable results. The first, additive formalism of forces and torques implemented made the dynamics of the model unpredictable when more trackers were added. The equivalent constraint field, formalized in the last weeks of this work, allowed to overcome this instability, by aggregating all trackers contributions into a single, stable correction applied to the model. Through the use of synthetic data, we have shown that the stability of the current model makes it robust to high noise values, with an important ratio of trackers failing at the same time with contained impact of the estimated pose.

Still, important work needs to be done in order to acquire more in-depth results. We expect that systems such as OptiTrack, set up within a controlled environment should allow for accurate ground-truth comparison. If such a system was initially used to record some experiments, the restricted displacements we were able to record made data too sparse to be used as ground-truth. Also, issues were spotted during these initial experiments. As the recording was performed in an office-style environment, most shapes present long, aperture-prone edges that are unfit for tracking. This further restricted the amount of real-world results available at the time of writing.

We believe however that this method presents numerous advantages to recover the pose of a passive vision system. The whole visual odometry pipeline uses only events, without requiring additional sensors. Moreover, the model parameters allows to very easily adapt the system response time and typical length involved.

Chapter 4

Events Dynamics & Processing

4.1 Introduction

The final chapter gets out of the scope of event-based visual odometry, and presents a broader analysis of the event-based paradigm, its current implementation, and the implementation attempt made to match those event-based requirements.

The first part will dive into the relation between events and time. Over the previous chapters, time has been at the core of algorithms, and its consideration has brought many challenges to light. Time is indeed a center part of all event-based algorithms, and is, in my opinion, overlooked too often - if not completely ignored in many cases. To properly develop event-based algorithms that can exploit the full advantages of neuromorphic sensors, one must first understand what is time in this paradigm, and how to consider it properly.

We have dealt with time in the first chapter as an intricate part of the algorithm, as a variable that is computed and used in a feedback loop. This has shown effective, even though some instability can result, and had to be taken care of. The second chapter used feed-forward time information, which resulted in much better results. The third chapter also dealt with time, using events timestamps as basis for the numerical simulation of our pose system. This is also a key feature of events, as the event stream gets denser as motion is faster. Those different properties and approaches of time in neuromorphic computations will be discussed, in order to extract some guidelines of how time could be handled in event-based processing.

The second part will describe the framework wrapping algorithms implemented throughout this work, implementing this time computation process, as well as some key elements of the event-based processing philosophy we tried to implement. This framework especially allowed to work on an event-by-event basis, thus allowing to fully understand how those algorithms react upon a single event update.

4.2 Time constants analysis

4.2.1 Time as the elementary data : an overview

Time is the key for neuromorphic computation. All events, however they are encoded whatever they encode, carry a time element, stating when that event occurred. In that regard, it is fundamentally different from an image, or a particular frame of a conventional video stream. An image can be processed as such, regardless of its context. It is a macroscopic object, carrying standalone information that one can extract and infer higher-level information from. On the other hand, a single event does not contain much data. On its own, an event only state that *something* happened at a certain time, with possibly a few additional bits of data of various nature. A set of events have to be taken and processed together in order to make sense, and for that reason, time is of the essence for the event-based paradigm.

Still, time is not easy to grasp for a computer. Like anything, time is stored within a computer as bits, which are purely dimensionless data. By convention, we assign a unit to them. For instance, UNIX time is stored within a computer as an integer number, of value 1632478152 at the time these lines are being written. Only we can understand the link between this number, and the physical unit that is the second. This convention allows for instance a computer to compare the UNIX timestamp to the mass of earth, as both values are just pure numbers. In our case, the time information carried by an event have to be compared to a well-defined timescale to make sense, and thus for this event to be processed correctly. We must thus find a way to define this timescale properly.

This type of analysis about the fundamental meaning of time in neuromorphic processing is rare. The principles we will develop here are similar to the ones from the most important study on this topic [103]. However, this study mostly restricts those principles to the creation of frames, for display or conventional frame-based processing purposes. We believe that the importance of time in neuromorphic computation goes far beyond frames generation for display purposes.

If a frame is accurately produced, in the sense developed in [103], it is often done by maximizing the image gradient, *i.e* by creating sharp edges. The image, as we want it, is made of single-pixel edges, and as such is deeply linked to the lifetime of a single event[1]. This lifetime can be used on many occasions. Not only in frames generation, but also in optical flow, time-surfaces decay, variables averaging, ...

a) Defining the timescale of events

To compute the timescale over which we must consider the events timestamps, we must first know what that time scale should represent. An event carries an elementary amount of information, which is possibly very diverse. Let us focus on usual neuromorphic retinas events, that are triggered by a luminosity change on a single pixel. This luminosity change can be induced by two main causes. The first one is an absolute luminosity change of the object in direct line of sight of that pixel, like a light being switched on/off. This is a special situation, in which many pixels will fire at the same time, and is not often occurring. The second case, and the one that makes event-based cameras so interesting, is a local gradient of luminosity of the scene that moves relative to the sensors.

[1] We assume here that an object passing by a pixel creates a single event on that pixel, and that all successive pixels on the object trajectory generate events.

This can be either an object moving between a camera and the background, a camera moving, or a combination of both. The part of the scene that triggers the event can be the edge of an object, its texture, ... This trigger is thus locally defined by a 1-dimensional moving shape, both in $3D$ space and on the $2D$ sensor. To recreate an image of the scene at time t, one must know where that $1D$ shape is at that particular moment, using only the generated events. Put differently, one must know which events e_i received at times $t_i < t$ are still a valid representation of the scene.

Let us define the ideal lifetime τ_i of an event as such : for any two events e_i and e_j, created by the relative motion of local edges[2] $d\mathcal{E}_i$ and $d\mathcal{E}_j$:

$$t_j \in [t_i, t_i + \tau_i] \implies d\mathcal{E}_i \neq d\mathcal{E}_j \tag{4.1}$$

With that definition, any image displaying the set of events

$$\mathcal{I}(t) = \{e_n, t_n \leq t, t_n + \tau_n \geq t\}$$

should display single pixel edges, as we are sure we only take into account one pixel generated by each part of the scene. With that definition, we know that after a duration τ_i, an event e_i is outdated, meaning that more recent information can be found in the surroundings of e_i.

This definition describes an ideal case, in which we can define a time constant for each and every event. An event-specific timescale will be discussed later, specifically in sections 4.2.2.c and 4.2.2.d. However, such a definition presents algorithmic and computational challenges. First, if each event displays its own time constant on which it must be considered, then we cannot create and compute sensor-wide variables, like sliding averages, as we have done on many occasions in the previous sections. This type of computation requires a single time constant to be considered for all events at once, and a global decay of such a variable would not be able to take into account the specific timescale of each event. The solution would be to store, and decay each event, one by one. If this can be done, the computational cost would be tremendous. To be implemented, it requires to store all past events and not only the latest events at a certain location like the spatio-temporal context does, as well as the associated time constants. Then, upon computation of a decayed variable X, we would need to loop over all those events, decay them one at the time, and compute the updated value of X.

The second and more fundamental issue with definition 4.1, is its non-causality. We assume that any event e_j triggered between t_i and $t_i + \tau_i$ can not have the same origin as event e_i. If we can compute values that gives insight on the signal dynamics, as shown later, any sudden - and thus unforeseen - change of dynamics will make the previously estimated τ_i value void. Online computation, by definition, requires local dynamics assumptions whose impact must be minimized, and this affects timescales among other things. This paradox is shown in figure 4.1. Successive events are represented, with increasing x coordinate at times $t_{1...5}$, and we assume they are triggered by the same moving object. We show that this ideal lifetimes τ_i of event e_i can only be computed once the next event e_{i+1} occurs. This implies that the lifetime of an event can not be computed as the event is processed.

A solution to the first problem, as implemented in almost all algorithms, is to consider a screen-wide time constant. To that end, we assume that the average value τ over all events lifetimes describes with sufficient accuracy the entirety of the events timescales.

[2] We call local edge the portion of an object's edge whose projection on the focal plane would be of size $\sim 1px$.

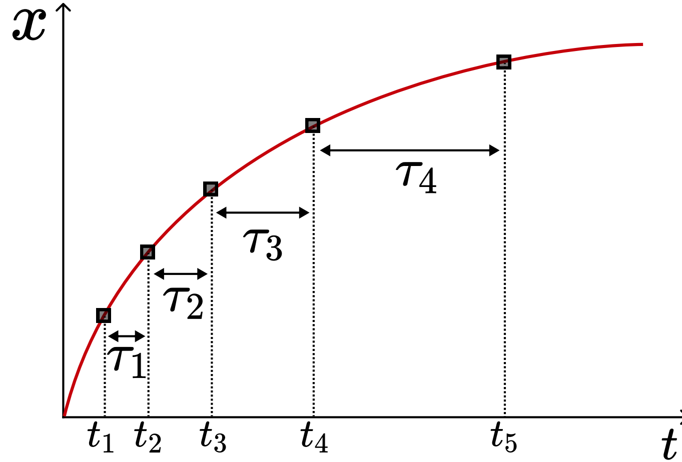

Figure 4.1 *Example of ideal timescales that should be used to compare events (black squares) timestamps in the case of a single-pixel moving object (red line). Each event validity time window ends when the next event triggered by the same object starts, leading to single pixel edges representations, or accurate and low delay response computations.*

$$\tau = \overline{\tau_i} \tag{4.2}$$

This paradigm have been widely used throughout this work, allowing for simple and efficient object tracking, depth estimation, ego-kinematics and visual odometry. The second issue, however, requires more in-depth work on how to define this screen-wide timescale, and how to consider events lifetime in comparison.

b) Static vs. dynamic timescales

In many cases, a time parameter is set manually for every processing block of the framework. Such algorithms are often quite sensitive to their timescale, as all time values they will process, will refer to that static timescale. However, even if it is changed for each and every experiment, such a static timescale can not match the whole dynamics of a recording, apart from rare cases. Using sets of events defined by static time windows to define a dynamics-changing signal will fatally lead to over or under-sampling, which are the very flaws event-based engineering allows to avoid.

An example of this type of pipeline is displayed in figure 4.2. With this constant timescale paradigm, not only must we set a timescale for each processing unit, we must also carefully set it accordingly to the data dynamics. However, these issues can be overcome using the processed signal itself, as we have used on several occasions in the previous chapters.

As data gets processed within a framework, the information rises to higher levels of abstraction. From initial raw events, one can extract optical flow, stereo matching and depth information, or visual odometry to only name the ones studied in this thesis. Along with this higher level of information comes a higher level of understanding for the time constants themselves. Indeed, in most cases, abstracting data comes with abstracting from the signal noise. Thus, a time constant

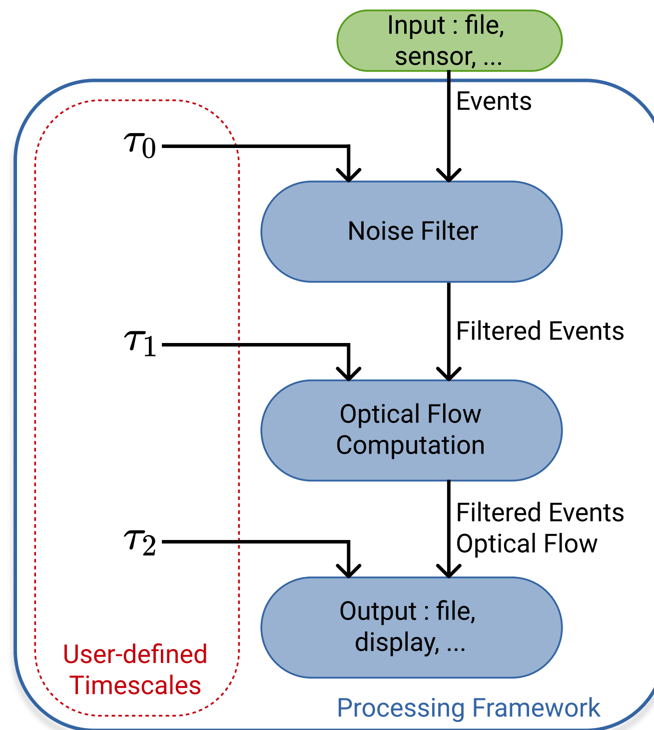

Figure 4.2 *Example of an event-based processing framework with a noise-filter, followed by an optical flow processing module. Each module is given a time-constant as parameter. As the time scale of the data is not immediately accessible. there is no guaranty that the time constants* τ_0*,* τ_1 *and* τ_2 *set as parameters fit the likely time-dependent dynamics of the signal.*

extracted from this high-level information should also be more accurate.

From that idea, we can build, in parallel to the data processing pipeline, a time processing pipeline[3], where each block that allows it improves the accuracy of the events timescale. Once we reach the desired precision, or the adequate level of abstraction, we will have built the time constant that fits best our previous definition. An example of this type of pipeline is displayed in figure 4.3. An initial timescale τ_0 is set as parameter, but the pipeline allows to extract more accurate and dynamical timescales $\tau_{\mathcal{A}}$ and $\tau_{\boldsymbol{f}}$ as data is being processed.

4.2.2 From low-level to high-level time constant

a) Initial time constant

The problem remains that to build timescales from processed data, one needs, at the start of the process, an initial time constant τ_0. That time base is expected to be raw and inaccurate, and must be set at the right order of magnitude, for it allows the lowest-level computation to be performed.

[3]In terms of implementation, the two processing frameworks are intertwined. However, a fundamental distinction have to be made. While one processed and created higher-level data, the second one computes and improves the timescale that data should be compared with.

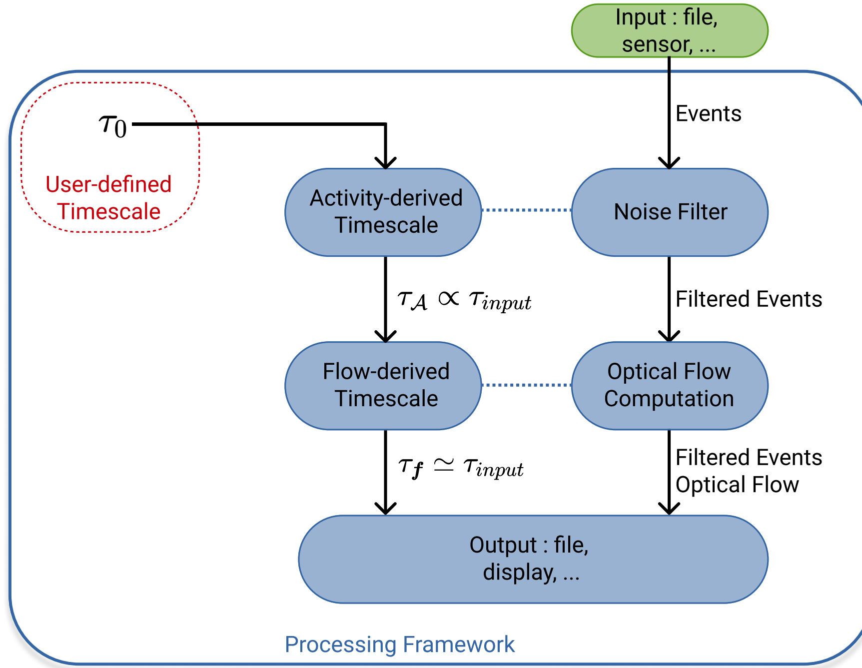

Figure 4.3 *Example of an event-based processing framework with a noise-remover module, followed by an optical flow processing unit. Only the first module of the pipeline is given a time-constant τ_0 parameter, and feeds the following module with a data-dependant timescale $\tau_{\mathcal{A}}$, based on events activity. This first variable time scale can be proportional to the data time-scale τ_{input} up to a user-parameter ρ. It is fed to a second module that computes the optical flow, extracting at the same time a second variable time-scale, τ_f, matching closely the signal dynamics. This type of architecture has been used in the presented works, as it requires less parameters tweaking to adapt to any type of scene.*

This hierarchy, however, allows the framework to break away quickly from this parameter τ_0. Even if the system result will depend - strictly speaking - on it, most of the timescales we tried and developed came out fairly independent from this initial value. Experimentally, τ_0 is set of the order of $1ms$, but we were able to change it, from $10ms$ to $0.1ms$ with little to no effect at all on the outcome. Let us now develop three data-derived timescales, which were implemented and used in our experiments.

b) Activity-derived timescale

The first and most basic computation we can do on events is counting them, yet that basic operation allows to extract a timescale $\tau_{\mathcal{A}}(t)$. Let us assume a $w \times h$ pixels neuromorphic sensor. As developed in [103], a frame ideally generated presents sharp edges, and this criterion is often used for events to frame conversion. By studying this constraint over multiple recordings, we observe that this constraint can be approximated by a certain density ρ of active pixels on a frame. If this density value can change with the sensor resolution, its order of magnitude often remains constant when the observed scene changes. Crucially, as this value depends on static properties of the scene - which are the *amount* of edges on screen at any time t -, it is independent from the sensors motion. This leads, in the first order, to a constant number of pixels $N \equiv \rho w h$ plotted at

once on screen.
We have experimentally set that $\rho \simeq 3\%$ for a DAVIS-346 sensor [4], leading to around 2700 events displayed at once.

This number of active pixels can be linked to the activity of the event stream, using the initial constant timescale τ_0. By assuming that each event affects a different pixel over the considered timescale, we can count the events, and derive the stream activity in an event-by-event fashion. Between two successive events e_n and e_{n+1}, we update the activity $\mathcal{A}_{\tau_0}$:

$$\mathcal{A}_{\tau_0}(t_{n+1}) = \mathcal{A}_{\tau_0}(t_n)e^{-\frac{t_{n+1}-t_n}{\tau_0}} + 1 \tag{4.3}$$

We can suppose that in the incoming event stream, each event represents a new information, without occlusion, spikes trains, ... This means that the activity $\mathcal{A}_{\tau_0}(t)$ computed here is a measure of the average number of new pixels that fired between $t - \tau_0$ and t[5]. Assuming τ is set accordingly to the signal dynamics at all time, that activity $\mathcal{A}_\tau(t)$ should be constantly equal to the expected number of events N needed to generate a frame. Let us define $\tau_\mathcal{A}(t)$ as this time-dependant value. With the goal being $\mathcal{A}_{\tau_\mathcal{A}} = N$, we define $\delta t = \overline{t_{n+1} - t_n}$, and assume a steady state is reached. This leads to

$$\begin{aligned} \frac{\mathcal{A}_{\tau_0}(t) - 1}{\mathcal{A}_{\tau_0}(t)} = e^{-\frac{\delta t}{\tau_0}}, \frac{N-1}{N} &= e^{-\frac{\delta t}{\tau_\mathcal{A}(t)}} \\ \Rightarrow \tau_\mathcal{A}(t) &= \tau_o \frac{\log(\frac{\mathcal{A}_{\tau_0}(t)-1}{\mathcal{A}_{\tau_0}(t)})}{\log(\frac{N-1}{N})} \end{aligned} \tag{4.5}$$

For large activities $\mathcal{A}_{\tau_0}$ and expected pixel occupancy N, we can simplify this expression, and get

$$\tau_\mathcal{A}(t) = \tau_0 \frac{N}{\mathcal{A}_{\tau_0}(t)}$$

Experimentally, we have seen little to no dependency on τ_0 as previously stated. The only constraint is for τ_0 to be small enough to react to very fast changes in dynamics, and large enough to allow for decent averaging on the number of events. However, this constraint leaves plenty for margin to set τ_0. Changes in sensor dynamics are often on the $0.1s \sim 1s$ timescale, while most sensors produce several hundred thousand events per seconds, meaning an average time interval between two events of the order of $10\mu s$.

This computation performs best when $\mathcal{A}$ is around 100 to 1000, but those figures are indicative, as they are sensor dependent. Notice that by definition, the timescale defined in this section is sensor-wide, and thus cannot be specific to a certain part of the scene, as opposed to the two following proposed timescale definitions.

c) Optical flow-derived timescale

From the definition we gave of an event time-constant, which is the time for a $1px$ displacement of the object on screen, we expect the ideal computation of the time constant to be derived from the

[4] `https://shop.inivation.com/collections/davis346/products/davis346-academic-rate`

[5] If we assume $n_e = 1/\delta t_e$ events are produced per time unit, and assuming $\delta t_e \ll \tau_0$, we have

$$\frac{\partial \mathcal{A}_{\tau_0}}{\partial t} = -\frac{\mathcal{A}_{\tau_0}}{\tau_0} + n_e \tag{4.4}$$

The steady state is then reached for $\mathcal{A} = n_e \tau_0$, thus the activity is the number of events produced by the sensor over a duration τ_0.

optical flow. That optical flow translates the apparent motion of an edge, and thus is the inverse of that time constant. We define the optical-flow derived time constant as $\tau_{\boldsymbol{f}}$.

To compute the optical flow, we need first a set of events to consider. Usually, for each incoming event, we consider a set of events located in the spatial neighbour, and with a temporal window. Set aside the spatial length considered, we need a first time constant for that second parameter. Again, we consider τ_0 the initial timescale used for this processing unit. The set of events considered to compute the optical flow for event e_n, located at (x_n, y_n), and that occurred at time t_n, is then $\mathcal{S}_n$ such that

$$\mathcal{S}_n = \{e_i, ||(x_i, y_i) - (x_n, y_n)||_k \leq R, t_n - N\tau_0 \leq t_i \leq t_n\} \tag{4.6}$$

with k the considered norm, often being the $k = 1$ norm for square matrices sensors, and $N \sim R$ a constant being the number of past location of the same object we want to consider.

They are several methods to extract the optical flow, either from a stream of events [38, 52, 87], or directly from a sensor [22, 23]. In both cases, the result can be given as a $2D$ vector $\boldsymbol{f_n}$ representing the direction and speed of the local edge object on screen, in *pixels per second*. As long as that method gives a good estimate of the optical flow amplitude, we then have the timescale $\tau_{\boldsymbol{f},n}$ specific to event e_n:

$$\tau_{\boldsymbol{f},n} = \frac{1}{||\boldsymbol{f_n}||_2}$$

To compute the screen-wide time scale, we propose to average those extracted event-dependant timescales, with the event-based average update rule:

$$\begin{aligned}
\lambda_{n+1} &= e^{-\frac{t_{n+1} - t_n}{\tau_0}} \\
\mathcal{A}_f(t_{n+1}) &= \mathcal{A}_f(t_n)\lambda_{n+1} + 1 \\
\Sigma_{\boldsymbol{f}}(t_{n+1}) &= \Sigma_{\boldsymbol{f}}(t_n)\lambda_{n+1} + \tau_{\boldsymbol{f},n} \\
\tau_{\boldsymbol{f}}(t_{n+1}) &= \frac{\Sigma_{\boldsymbol{f}}(t_{n+1})}{\mathcal{A}_f(t_{n+1})}
\end{aligned} \tag{4.7}$$

Notice that in this case, the specific timescale $\tau_{\boldsymbol{f},n}$ depends on the event location (x_n, y_n). The averaged timescale must discard the location specificity to represent a larger set of events. However some algorithms may use this location specificity to their advantage, when considering local sets of events, or performing single-pixel computation.

d) Ego kinetics-derived timescale

A similar time-constant method to the optical flow one is to use the result of a visual odometry, similar to the one developed in Chapter 2. We use information of depth and optical flow to compute the instantaneous velocity of a sensor with respect to the observed scene. Using equation 2.22 allows us to compute the theoretical on-screen velocity for a certain event e_n, and through a similar process to the one described in the previous section, compute the event-specific timescale $\tau_{\boldsymbol{v},n}$.

$$
\begin{aligned}
\boldsymbol{v_n} &= K \begin{pmatrix} -\frac{X'_n Y'_n}{f} & \frac{f^2+X'^2_n}{f} & -Y'_n & \frac{f}{Z_n} & 0 & \frac{X'_n}{Z_n} \\ -\frac{f^2+Y'^2_n}{f} & \frac{X'_n Y'_n}{f} & X'_n & 0 & \frac{f}{Z_n} & -\frac{Y'_n}{Z_n} \end{pmatrix} \boldsymbol{\Omega(t_n)} \\
\tau_{\boldsymbol{v},n} &= \frac{1}{||\boldsymbol{v_n}||}
\end{aligned} \tag{4.8}
$$

In this case, we need depth information, along with pixel location, to compute the event-specific timescale. The overall computation is computationally heavy, but should result in the most accurate timescale for this particular event, as it uses all the available information, and overcomes sensors noise through screen-wide average. The average timescale $\tau_{\boldsymbol{v}}$ is finally computed in a similar fashion as in equation 4.7.

$$
\begin{aligned}
\lambda_{n+1} &= e^{-\frac{t_{n+1}-t_n}{\tau_0}} \\
\mathcal{A}_v(t_{n+1}) &= \mathcal{A}_v(t_n)\lambda_{n+1} + 1 \\
\Sigma_{\boldsymbol{v}}(t_{n+1}) &= \Sigma_{\boldsymbol{v}}(t_n)\lambda_{n+1} + \tau_{\boldsymbol{v},n} \\
\tau_{\boldsymbol{v}}(t_{n+1}) &= \frac{\Sigma_{\boldsymbol{v}}(t_{n+1})}{\mathcal{A}_v(t_{n+1})}
\end{aligned} \tag{4.9}
$$

4.2.3 Retrieving past events

Let us now consider our screen-wide, dynamical timescale $\tau(t)$, computed for instance by one of the above blocks. This timescale reacts to the signal dynamics, meaning that slow relative object motion will lead to large timescales, while a static sensor in front of a static scene, producing almost no event, could present a timescale $\tau \sim \infty$. As already analyzed in [104], event-based algorithms divide into two main categories: fully event-based methods or accumulative methods, the latter using sliding windows to define a set events to be worked upon.

Both categories, however, need the previously defined timescale to select and/or process these events. The main idea for this timescale, as suggested before, is to define the relation between an event and its temporal neighbourhood. More specifically, for online algorithms, it defines the relation between an event, and past events. How should we consider past events, or update values built from past events from the current one and its timescale ?

The update rule, used several times in the past chapters and sections, as in equations 1.9, 2.14 or 4.3 decays events participation, and is a fundamental element of fully event-based methods. Each event is processed, and its data is discarded: only the infinitesimal contribution remains in the computed variable. The computation is thus efficient, light-weight, low-latency. However, accumulative methods will undoubtedly always be required to compute certain values. Storing and recovering past events from $2D$ maps is needed for optical flow computation[6], stereo-matching, object recognition, etc. Let us then analyze how to retrieve events in such cases.

[6] A fully event-based version of [86] was developed during this thesis. However, it required to store a equivalent representation of events data onto $2D$ maps. If the update was made in an event-per-event fashion, and the result mathematically correct, the resulting computational cost was much higher than the initial version. This was due to more data to store and retrieve from RAM.

For a sliding window with a dynamical duration $\tau(t)$, the timestamp of the oldest event considered at t is given by

$$t_{old}(t) = t - \tau(t) \tag{4.10}$$

If the timestamps of events t_n are assumed to increase, nothing ensures with the construction of τ that $t_{old}(t)$ is monotonic. A sudden change in dynamics can rapidly produce an extremely large timescale. If seemingly insignificant, the consequences of this decreasing oldest timestamp are troubling. This property implies that a past event, unused at time t, would regain importance at time $t' > t$. Put differently, an event discarded because its contents could not accurately represent the scene anymore is suddenly deemed valid anew. From equation 4.10, and with t_{old} increasing, it appears that any dynamic timescale $\tau(t)$ used as such to retrieve events from a sliding window must verify

$$\frac{\partial \tau}{\partial t} \leq 1 \tag{4.11}$$

To enforce this rule, we need to fix the lower boundary of our sliding window, and ensure the oldest timestamp monotonic behaviour. For any events set $\mathcal{S}_n$ retrieval triggered by event e_n at time t_n defined by constraint $\mathcal{C}_n$ on the events content - apart from time -, we propose the following process:

$$\begin{aligned} t_{old,n} &= \max(t_{old,n-1}, t_n - \tau(t_n)) \\ S_n &= \{e_i, t_i \in [t_{old,n}, t_n], \mathcal{C}_n(e_i)\} \end{aligned} \tag{4.12}$$

4.2.4 Dynamic window experiment

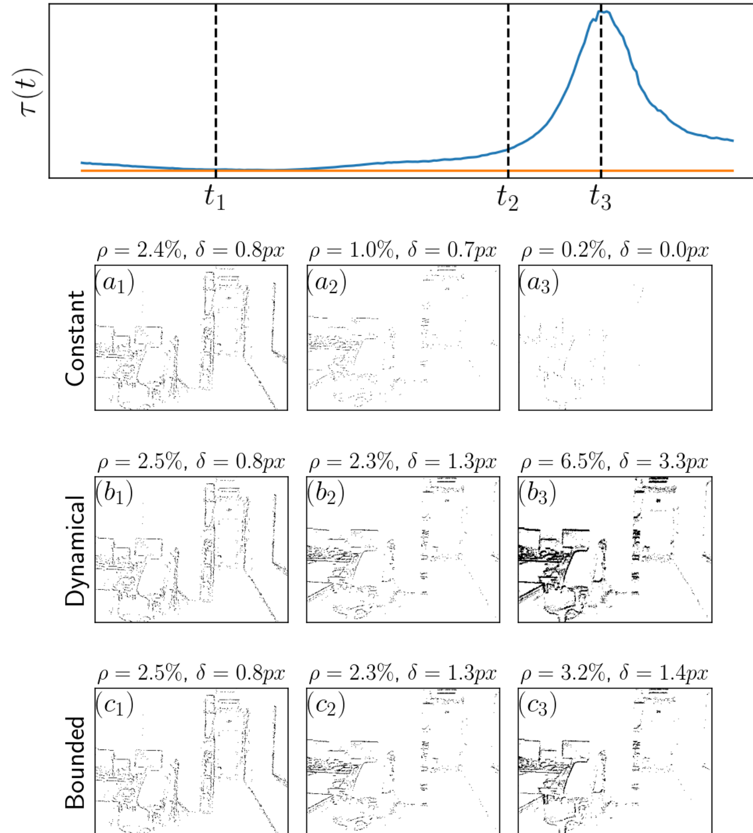

Figure 4.4 *(Top graph) : constant timescale - orange line - vs. adaptive timescale - blue line - during dynamics changes in a recorded scene. The almost complete stop of the sensor leads to a large spike in the estimated timescale of the scene. (Bottom pictures) : comparison of different frame reconstruction methods. A constant timescale can be set according to the scene at a particular moment* (a_1) *but will not be able to describe other dynamics of the signal, leaving blurred or under-sampled reconstructed frames* (a_2, a_3)*. An adaptive timescale can react to changing dynamics* (b_1, b_2)*, but leaving the sliding window unconstrained can lead in extreme cases to the display of previously discarded events* (b_3)*. Using an adaptive time window and forcing the monotony of the timestamp of oldest events* (c_1, c_2, c_3) *allows for proper reconstruction in most situations.*

Using a sliding window along with a constrained dynamical timescale allows to retrieve the most accurate representation of the visual scene at any time, regardless of the dynamics of the scene. In Figure 4.4 we compare three different types of time window: *Constant* (a), *Dynamical* (b) and *Bounded* (c). *Constant* uses a constant time window, *Dynamical* and *Bounded* an adaptive one, with *Bounded* adding a constrain to ensure monotonic t_{old} values. During this sequence, the dynamics of the sensor drastically changes. We observe that the *constant* time window results in information loss (a_2, a_3), and that only adaptive timescales (b, c) can cope with these drastic changes. We also observe that in extreme cases, blurring can still occur if the time window is unconstrained

(b_3), as the highly increasing time window allows the use of previously discarded events. The *bounded* window, with monotonic older timestamps, allows for proper frame reconstruction in most situations, regardless of the signal dynamics.

To quantitatively qualify the reconstruction, we compute the average density of the frame ρ, as well as the average width δ of edges appearing at times $t_{1,2,3}$. As the observed scene remains almost unchanged and so the same edges appear throughout the experiment, the density is expected to remain constant for all frames. Table 4.1 reports these metrics extremas.

Window Size	Density ρ		Edges width δ	
	Min	Max	Min	Max
Constant	0.2%	2.4%	0.0px	0.8px
Dynamical	2.3%	6.5%	0.8px	3.3px
Bounded	2.3%	3.2%	0.8px	1.4px

Table 4.1 *Results of frame density and average edge width for three different frame reconstruction methods. We observe that only a dynamical time window with ensured monotonic older timestamp (bounded) allows to minimize the variation of the metrics.*

With this experiment, we observe that only the *bounded* time window allows for constant frames densities. This correlates with low variations of the average size of the considered edges. The unconstrained dynamical time window, allows to properly reflect most of the signal dynamics, but cases such as sudden slower motion of the sensor are still a problem. This example finally highlights in a straightforward way the limits of the commonly used fixed time-window.

4.2.5 Conclusion on events timescales

This section has shown that time is a key element of event-based processing but is too often overlooked. We have presented three different methods to estimate the correct timescale from events, using only visual data and an initial timescale. We have also shown that this initial timescale is necessary only to ensure that the proper order of magnitude is set for time handling. However, the proposed methodology allows to have little to no dependency to this initial timescale. This ensures that events timestamps are considered with respect to the signal dynamics, and not to an arbitrarily set and tweaked time parameter.

We have also shown that adaptive time windows are not the only requirement to properly address event-based computation, and additional care must be taken, especially when retrieving past events for accumulative methods. The experiment using frame reconstruction as main support has shown that if this dynamical time window allows to extract a set containing at least all relevant events, this set often also comprises older, out of date events. A monotonic, increasing lower time boundary is thus required, in order to allows for the necessary and sufficient set of event to be extracted in any situation.

We believe this type of events retrieval method, if visibly effective on frame reconstruction, can improve sensibly most algorithms that require this type of temporal window. This methodology was tested upon various algorithms (tracking, ego-kinetics estimation, depth estimation) and have led to significant improvements, as shown in section 2.3.3.a. However, this type of timescale interaction can hardly be implemented on any existing event-based computation framework. This, among other development needs, led to the development of our own, interactive event-based development framework, named *PEBBLE*.

4.3 PEBBLE : Python Event-Based BLEnder Framework

4.3.1 Development frameworks overview

All Computer Science domains, from audio processing to game design, have accumulated years of experience in building their specific development tools, with their own philosophy, suited to their particular needs. Similarly, computer vision has its own suited, forged environment. Likewise, image processing and video processing have had their own dedicated tools and libraries, in order to create a simple yet efficient development environment. However, within a domain, different philosophies can appear, each approach requiring different development tools.

This is currently happening within the event-based field. Many frameworks for event-based data processing have been developed and published during the last decade. Some of these frameworks must be highlighted for the way they were build, or the notoriety they have gained over the years:

- jAER (Java Address-Event Representation)[7] is one of the first widely used event-based framework, as it was made public in 2007. Its notoriety is due to the use of Java, a user-friendly yet efficient language, along with its link with the DAVIS sensors family, that is one of the most used neuromorphic cameras. The native interfacing made the use of those cameras very easy, and was a reasonable choice for taking first steps into the event-based world. It also introduced the *AEDAT* file format to store streams of events.
- Tarsier[105][8] is a C++ event-based framework. Its has shown among the best performances in terms of speed and low-latency for event-based processing. The philosophy behind it is functional programming, gaining orders of magnitude of processing efficiency, at the cost of coding simplicity for beginners. Several drivers have been developed to let it interface with different neuromorphic cameras. It introduces a new file format, called *Event Stream*.
- DV[9] is the newer framework replacing jAER. Its interface and recollection of different modules makes it easy to use and interface with, while still allowing good performance. It can also interface with the different DAVIS sensors, and uses upgraded versions of the *AEDAT* format.

Streaming from a live camera allows for testing algorithms in real-use cases or in embedded systems. However this was not a priority during this thesis work, and we focused on interfacing with many file formats instead in order to use publicly available datasets from a wide variety of neuromorphic sensors and to compare our results with other algorithms. Furthermore live streams requires real time processing. Again, this was not a priority in this work, as we were more interested in being able to peek at the event level at any stage of the algorithm. This allowed us to gain a deeper understanding of the on-going processing and helped with algorithmic development. Visualization at all steps for all events is barely compatible with real time. As our need for a simple, easy to interact with, versatile framework was growing more and more, that lead us to develop our own development framework, where data accessibility and interaction are paramount.

Let us introduce PEBBLE : the Python Event-Based Blender. At its core is the idea that complexity should be hidden from the user. It makes interacting with event-based data accessible to anyone, able to deal with all data types, and even extending to possible non vision-related event-based

[7] `http://jaerproject.org/`
[8] `https://github.com/neuromorphic-paris/tarsier`
[9] `https://inivation.gitlab.io/dv/dv-docs/`

data.
On the other hand, computation efficiency was not crucially searched for. Aiming for performance often come to the cost of user-friendliness, or interactivity. But more importantly, improving performance of event-based algorithms on hardware designed for frame-based data is, in my vision, a time ill-spent. Some frameworks will allow the use of GPUs, while they are inherently made for frame-based processing. Those two paradigm completely differ in the way data is considered. If optimization can be found this way, it is because of this hardware's long years of maturation, rather than its relevance for event-based data. Also, the event-based philosophy relies heavily on small chunks of data, updating a system state, as we have seen in the previously described algorithms. Architectures optimizing large amounts of data to be retrieved in memory, processed and stored again are at the opposite of this philosophy. Neuromorphic dedicated hardware is the key for optimization of event-based algorithms; prior to their development, it is hard to imagine any optimized event-based algorithm able to compete in regards to computation efficiency against conventional vision algorithms. For these different reasons, no evaluation in latency or computation speed of PEBBLE was performed.

Yet, this framework was crucial in the ability to rapidly develop, test, and upgrade event-based algorithms, on a variety of different data. Python itself is designed to be easy to use: being a high-level, interpreted language, there is no need to set up complex compilation processes, and debugging is greatly eased. Finally, most of the framework core relies on standard Python libraries, along with NumPy, which is commonly used. This enables a light-weight installation, making PEBBLE a great first step in event-based programming, yet allowing a deeper understanding at the event level.

This section must be seen as a general presentation of its features, underlying philosophy, and the versatility PEBBLE offers. The entirety of the framework, and the associated modules later described are available at `https://github.com/LDardelet/PEBBLE`. We highlight in this final section some of the paradigms implemented during the framework development, along with actual code, highlighted in blue.

4.3.2 Describing an event

a) Fundamental Events

PEBBLE was initially developed with only goal to process events from neuromorphic visual senors only, with the AER - Adress Event Representation - formalism. This can be summed up as a 4 variables set of data, being a timestamp t, a pixel location (x, y) and a polarity p. Thus, all the initial work was done based on this single type of data. However, it rapidly became clear that those were not the only possible events that could be propagated through the framework : data concerning trackers1, or disparity2 appeared, and the need for versatile data type became obvious. Still, a common structure was needed for the framework to remain coherent and easy to use.

The common variable for all those data types was time. And it is actually the most basic event one can think of. Let us forget all about event-based sensors, and focus on general neuromorphic computation. When dealing with a stream of data recorded over time, as the neuromorphic paradigm does - no time means no change in time, thus no data - the first data brought by an event is about *when* the event occurred. If it can seem oversimplified, we have shown that this time information can be enough to recover a time-constant from an event stream in section 4.2.2.b, and improve the computation we perform upon it.

Thus, *PEBBLE* initially propagates basic events, related only to a timestamp. Those events

will then be completed with more complex information, such as camera events, flow events, ... This allows to attach more and more information to an event, as that very event triggers various computations. Those events, seen as containers for different data, are processed through the entire framework, until the last computation required has ended.

$t = 12.345s$

Figure 4.5 *The most basic event that can be propagated through the framework is an empty container, with only data being a timestamp.*

b) Additional data and multiple sensors events

However, a second piece of information that comes with general neuromorphic computation is the origin of the event. In some cases, this origin is pointless, as a single sensor input is set. However, multiple-inputs support became crucial when the work on stereoscopic data started. As two synchronized cameras at least are needed for disparity computation, the two streams have to be ran simultaneously, and we need a variable to tell their origins apart.

To that end, each input sensor creates a *sub-stream* of data, with a specific ID. For instance, the left camera of a stereo rig will generate events for the *substream* 0, while the right camera will generate events for the *substream* 1. The event container is then initially filled with an event of one *substream*, dispatching the data to the different modules of the framework. Due to some peculiar cases, the *substream* ID is not a property of the container, rather than if the event contained. Each container can thus hold data for multiple *substreams* at once, and even multiple events for each *substream*. This allows to pack a container with additional information, extracted modules along the container propagation. This makes the processing and propagation of data through PEBBLE very easy, while maximizing its versatility.

An example of an event container is given in figure 4.6. This container holds events of two different *substreams* at once, the *left* and *right* cameras of ID 0 and 1 respectively. Those two events share a same timestamp, at $0.123s$, however they can be of different nature. This multiple-inputs feature is a basic need for any event-based framework, as suggested by the evolution of neuromorphic sensors over the years. Neuromorphic cochleas are being developed, and their data type widely differ from neuromorphic cameras. Still, anyone who want to use multiple cochleas will be confronted to that very same issue. Even more so, sensors fusion - the interaction and computation upon data from multiple different sources - is, by definition, a computation that needs this kind of feature. The approach we developed, however not computationally optimized, is a convenient solution to this type of requirement.

We want to emphasize that the solution we implemented is mostly about semantics and user-friendliness. It makes very easy to attach more and more information to an event while it is being processed. However, the philosophy behind it is that all the information added to an initial event, whatever its nature, should be caused by this initial event. In the example presented in Figure 4.6, disparity computation is performed on an initial left CameraEvent. Subsequently, a right CameraEvent is created on the opposite camera *substream*, at the corresponding location, with the same disparity value but opposite sign. On the other hand, two events captured by two separate sensors with similar timestamps, should not share a common container, as they are fundamentally unrelated.

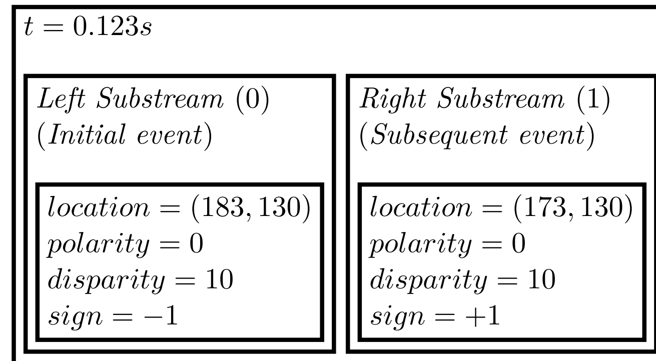

Figure 4.6 *Structure example of an event propagated in a 2-sensors pipeline. This container initially holds a left camera event with disparity information. In this example, the framework emulates a right camera event with a similar structure. In this case, both events are related, as computation on one generated the second. When events are not computationally linked, it is advised to use two different event containers, even for events with equal timestamps.*

c) Events types handled

So far, various event types have been implemented, mostly focused on the work made with neuromorphic cameras. We describe here a non exhaustive list of those types, along with their respective fields.

- The CameraEvent : It carries information from a silicon retina, as 2 fields variables : a tuple location : (**int**, **int**) for the pixel address and polarity : **bool** for the luminosity change sign.
- The Tracker Event : Used during the work on the feature detection and tracking algorithm described in Chapter 1, it hold information about a tracker. The important data information carried is the location : (**float**, **float**) and the ID: **int** of that specific tracker. However, additional information is added, notably fields concerning the state of the tracker or its orientation, as our algorithms also tracks features in rotation.
- The DisparityEvent : Used during the stereo visual computation part of this thesis, it gives information about the disparity - or sometimes referred to as inverse depth - of a CameraEvent. It should, therefore, always be attached to a CameraEvent. Its fields are the disparity : **int** and sign :**bool**. We chose to define the disparity as an integer, as most disparity computations are made at the pixel level, rarely at a subpixel one. Also, we define the disparity as a positive value, as it relates to the depth on the concerned object in the scene. The *sign* field is thus often optional[10].
- The FlowEvent : Carries information about optical flow performed on a CameraEvent. The only field is a tuple flow :(**float**, **float**) describing the amplitude of the optical flow along axis x and y of the sensor.
- The TwistEvent : Propagates information about the speed of the sensor. At it relates to speed, it contains two $3D$ vectors, being omega:(**float**, **float**, **float**) the rotation speed, and v:(**float**, **float**, **float**) the translation speed.

[10] $3D$ information could be passed instead of disparity. Future updates should solve this issue.

4.3.3 Modular conception of a framework

Now that we know what is being propagated through the framework, we need to define how the computation is going to be made. Similarly to other frameworks, like jAER, DV or Tarsier, we use elementary blocks assembled into a general-purpose framework. Due to the simplicity of the language, we assemble it at runtime, with assembling data stored into a project file. Again, similar to those other frameworks, the modules can range from simple modules storing data, to much more complex algorithms. The way we built that framework makes developing a module simple, as most the pipeline complexity is hidden from the user.

The project file is made of a single *JSON*, which can be easily read and modified if necessary. It stores all information about the modules in the pipeline, the ordering of that pipeline and defines some parameters as later described in section 4.3.3.d.

a) Input modules

At the start of any framework are placed one or more input modules. These input modules differ from others in the sense that they are the basis of events propagation, and thus require specific class methods. As the framework handle the event propagation by itself, it is transparent for the user to propagate several inputs at once. In this case, the framework selects the oldest event proposed by each of the *input* modules to be processed. As such, *PEBBLE* is not limited to monocular or binocular vision, but can handle any number of parallel events stream of various nature, with no core modification needed. Currently, two main input modules are available.

The first one, as in every other development framework, is a file reader. For now, it handles the following data files:

- *Dat* : Binary data file used especially by ATIS[12] sensors, and later on Prophesee[11] sensors.
- *Es (Event Stream)* : Compact, binary developed along with the Tarsier framework.
- *Aedat* : Binary file used by DAVIS-relatex software.
- *HDF5* : Binary, multi-purpose data storage files, used for large datasets.
- *Txt* : Plain text data files.

The goal here is to make it easy to implement any new extension through common methods and nomenclature. The variety of file extensions handled allows to work on most publicly available datasets with the same modules and assembled pipelines.
The second input module is a rig simulator, used for example in pose experiments in Chapter 3. It can create camera events, tracker events, disparity events and twist events from virtual spherical objects. As the goal was to create high-level synthetic data, the created CameraEvents must be taken with caution, as that simulator does not accurately replicates the behaviour of neuromorphic sensors, as opposed to [106]. The CameraEvents produced are mainly focused on scene visualization rather than processing purposes, although the base principles are implemented.

As previously stated, no USB-device handling module has been developed. The main reason was the need to benchmark algorithms on controlled data, and the ability to have reproducible results. However, the low computation speed of Python fundamentally forbids real-time handling camera event streams, for which optimized compiled code is required.

[11] `https://www.prophesee.ai/`

b) Memory modules

As most computation processed require to store past events data to later interact with, modules are being developed to deal with this task specifically. The goal is to minimize the amount of data stored, by sharing this memory with other, subsequent modules. It is thus possible, when building a processing pipeline, to create a link between two modules. Thus, stored data within a memory module can be made accessible to other computation modules. At the time of writing, three main memory modules have been implemented:

- Memory : Stores CameraEvents inside a spatio-temporal context, with polarity support.
- DisparityMemory : Stores DisparityEvents inside a spatio-temporal context, with latest time of update for each pixel and the associated disparity value.
- FlowMemory : Stores optical flow values, either on a spatio-temporal context, or as a list of the latest optical flow values received.

c) Other modules

Throughout this work, other computation modules were written, either for usual computations or during algorithmic development. We can classify them into three main categories:

- Filter modules : Small computation blocks, use mainly for noise removal, or for event stream tampering, like artificial noise, ROI selection, ... (example: RefractoryPeriod)
- Common modules : Performs common computations, like optical flow, corner detection, ... Note that if several methods can exist for a single usual task, we write them into a single python file (example: FlowComputer).
- Developed modules : Modules implementing an algorithm being developed. The previous chapters feature such modules (example: VisualOdometer).

d) Parameters

As for any type of data processing, parameters are necessary to tweak and change the program behaviour. *PEBBLE* implements three main ways to define a module's parameters for any experiment. The philosophy behind it, once again, is to improve fast development and versatility. All modules variables defined as parameters have to be defined inside the protected method _OnCreation, and start with an underscore. This allows *PEBBLE* to understand them as a parameters that the user may want to change when running an experiment, and allow for experiments parameters versioning. Among these parameters, protected variables _MonitorDt and _MonitoredVariables, detailed in section 4.3.5.c, allow for easy data monitoring.

The first way it to set default parameters for these variables inside the module python file. However, changing default parameters in the python file is not reliable, and induces changes that can be hard to keep track of.
Thus a second way to change parameters is to set them inside the project file mentioned earlier, which stores the pipeline layout. This is especially useful when a certain pipeline uses non-default parameters, like constant timescales, or noise filter properties. Using the project file to store these parameters allows to set them for this specific project, without having to change them at runtime nor to modify the module themselves.
The third way to change parameters is thus at runtime, when starting the experiment with framework method RunStream. One can set as optional value a dictionary specifying new values for the desired parameters. The set of available parameters can be accessed once the framework has been initialized, with the method GetModulesParameters.

4.3.4 Time propagation

PEBBLE was developed, among other things, to be able to propagate time information along with processed data, in order to overcome the challenges described in section 4.2. This rapid analysis of the importance of timescales in event-based processing have shown different ways time can be considered and used. In particular, we can set either a timescale for each particular event, or set a dynamical sensor-wide timescale. For a maximum versatility, and to allow experiments with both paradigms, the solution implemented uses the framework itself as a time keeper of some sort.

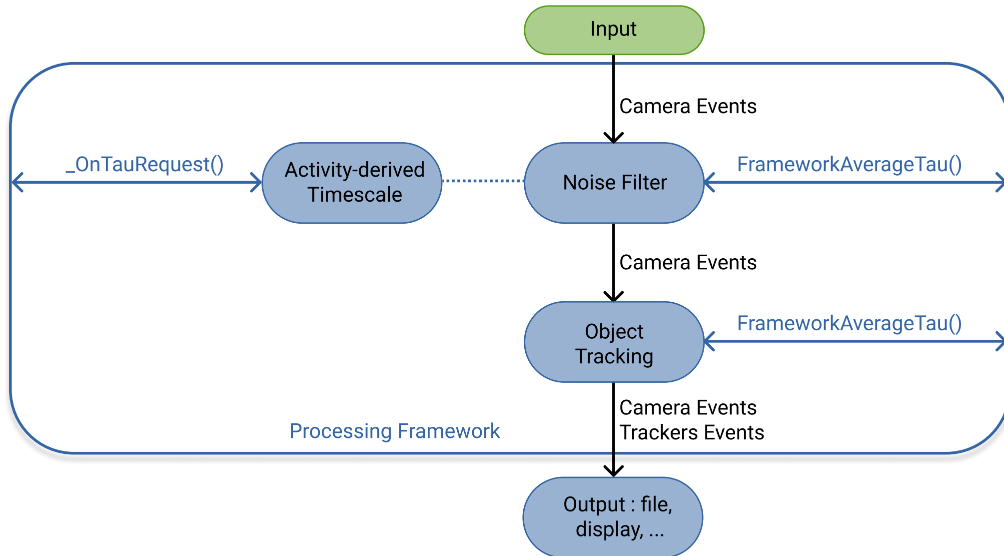

Figure 4.7 *Example of an event-based processing framework highlighting the time propagation solution implemented in PEBBLE. Each module of the framework can request a timescale to the framework through the method FrameworkAverageTau. A module can compute a timescale of its own, and return it with _OnTauRequest. The framework handles the selection of the highest-level timescale to be returned, using the order of the modules in the pipeline. In this example, the noise filter cannot access any timescale from previous modules, and the request returns None. The object tracking module will be given the timescale returned by the noise filter if available.*

Instead of propagating timescales along with the data, we use two modules class methods which request a timescale value from the framework. While FrameworkAverageTau asks for the screen-wide timescale, FrameworkEventTau gives additional information about the ongoing event. The framework then checks the previous modules[12], and returns the highest-level timescale available. Each module that implements a timescale estimation with the method _OnTauRequest can then be asked its current timescale value based on various event information. This highly-versatile implementation is an attempt to solve time propagation paradigm inside an event-based development framework, by allowing simple interactions between modules.

The timescale computed is, by convention, the timescale associated with a CameraEvent. Other

[12] As the pipeline is oriented, with input modules at the beginning, the order of modules it contains is highly important.

type of data, such as DisparityEvent or TwistEvent are expected to have much larger timescales, as they describe macroscopic values, about the scene or the motion of the sensor. This type of information changes over timescales orders of magnitude higher than the average lifespan of a single event. However, both these timescales evolve in a similar fashion w.r.t the signal dynamics. It is then left to each module to handle these timescales with care once retrieved.

4.3.5 Data interaction

a) Events Updates

One core feature of *PEBBLE* is its ability to run within the Python Interactive shell. This means that our framework can be run, and interrupted at any time, keeping all variables in memory. This especially allows to run short parts of an event stream, and observe the results of the tested algorithms, at their core.

More precisely, a stream can be processed in two different ways. The first one is a classic event stream experiment, started with the method RunStream. The function parameters start_at and stop_at allow to cut that experiment to specific moments of the sequence. At any time, that process can be interrupted by typing 'q'[13]. The stream can be resumed with method Resume, as long as the Python interpreter is still running.

The second way allows to process a single event at the time, by using the pipeline method Next. This feature allows to monitor at the smallest possible scale the effects of a single event update on the state of the computation. This especially showed useful for the development of our tracking algorithms, where trackers behaviour could be precisely analyzed.

b) Dedicated live display

As said previously, the goal of this framework is to rapidly develop and test algorithms, on different datasets, whether they are synthetic datasets generated at runtime, or stored in files. Using it inside a Python interpreter allows to gain access to all variables, by pausing the computation, and to have them kept available once data processing is over. However, an important feature most framework provide is the ability to visualize data output during processing itself. To that end, a specific module has been developed which interfaces with a third-party program, called the *Event-Stream Display*.

Using a third-party program allows to share the computer load of displaying live data. Indeed, one rarely need to visualize several live streams at once. On the other hand, it is convenient to be able to access live visual data from multiple sources at the same place.

This program, written in Python as well for simplicity, can be found at `https://github.com/LDardelet/EventStreamDisplay`.

To share data between a running *PEBBLE* framework and the display, we use a simple UDP connection, sending raw data to be interpreted and displayed. This simple type of connection protocol presents a very small overhead to share data between programs. If some data packets can be lost sometimes with this type of protocol, loosing event-based packets for visualization is no major problem. Moreover, using this connection on a local machine only generates little to no

[13] It is advised to use this interrupt rather than the common Python keyboard interruption, as the latter can be harmful to data integrity.

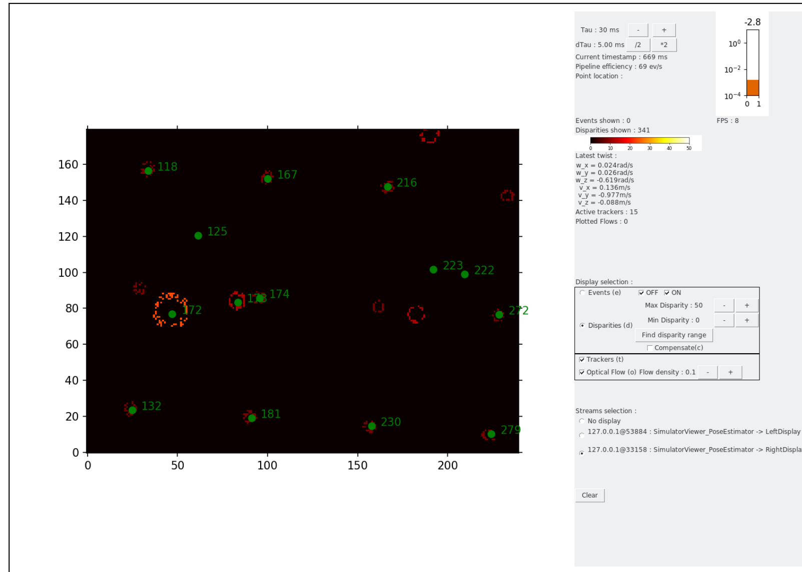

Figure 4.8 *Screenshot of the dedicated event stream display running.*

packet loss, while keeping the associated CPU time low.

Additionally, using this protocol to transfer data between processing unit and visualization allows to deport the processing load on dedicated machines, with more computational power than standard desktops. One can thus follow the live evolution of multiple processes at the time and on different machines, with almost no configuration needed.

At the time of writing, most events are compatible with the display. However, all events are not suited to be displayed on screen. For instance, For instance, TwistEvents described in section 4.3.2.c are not suited to be plotted as on-screen $2D$ information. To that end, they are updated live as text data, as they describe sensor-wide data[14].

c) Data monitoring

Each module, along with its specific _OnEventModule function, can perform data monitoring. The protected keyword _MonitorDt allows to specify a data monitoring time step, and the list _MonitoredVariables sets up the module class variables one want to save during the run. This allows for easy data access, as any monitored variable X can be retrieved in the Python interpreter through:

```
Xs = Module.History['X']
```

This data is then automatically saved in a CSV file, inside an experiment-specific folder. This data saving is done either at the end of the run, or when closing the python interpreter, to make sure no experiment data is lost.

[14]Live temporal graphs should be made available in future works, especially for this type of events.

4.3.6 Conclusion on PEBBLE

This framework was the cornerstone of algorithmic development throughout this thesis. If its implementation continuously evolved along the years, its philosophy of simplicity and user-friendliness remained, and allowed to carefully study the results of various algorithms at the scale of a single event. If time propagation is still an ongoing question, and additional developments are needed to give all the amplitude necessary to solve it, I believe this framework will help better understand the importance of time in event-based processing.

To this day, a comprehensive documentation is being written. As stated multiple times, the framework is designed to make algorithmic development easy. However, this framework will also need modifications of its own to display a maximum range of features. This complete code documentation will help interested developers from the neuromorphic community participate in reaching that goal.

Conclusion and Perspectives

General Conclusion

This PhD thesis introduces a new approach for motion estimation computation from event-based data. It enforces a fully event-by-event consideration of the stream. Using visual odometry as a main support, we have shown that the stream of events can be efficiently and correctly processed by separate processing blocks into higher-level information. Instead of relying on a single, black-box algorithm, we decompose the data in various forms. These pre-processed values can then be reassembled to recover sensor motion. Additionally, each computation block presents in itself a novel way to consider event-based data, allowing a deep understanding of the information carried by each single event.

Chapter 1 has presented a new, asynchronous way to track visual features, with no prior required about the shape or a velocity estimation of the tracked object, nor any relation to other objects of the visual scene. The closed-loop time estimation allows for each tracker to be mostly independent from the global dynamics of the scene. Moreover, these trackers can handle large changes in the dynamics of the tracked object. It also presents a new approach to feature selection, by discarding the detection process that usually takes place before any tracking is possible. Instead, it is the continuous update of various estimators that serves as a criteria if the considered shape can be tracked in a non-ambiguous way or not. This approach seems closer to the way humans perceive objects than usual shape recognition. Rather than looking for specific objects to follow, our vision is guided by microsaccades, stemming from local shapes motion, much like our trackers react.

The two first computation blocks in chapter 2 present new iterative manner on how to estimate optical flow and to perform stereo-matching in an asynchronous way. They have demonstrated state-of-the-art accuracy, while keeping the latency as low as possible by selecting the minimum amount of needed data. The third block carries on in this fully event-driven paradigm, and allows to estimate sensors' velocities in a unique way. This feature opens a new field of applications for neuromorphic engineering, as it only requires passive vision.

Chapter 3 has shown a way to fuse data extracted at two different temporal levels into a single model, while taking advantage of the punctual and asynchronous nature of events as timesteps for the computation. Using them as such allows the model to adapt itself to large changes in dynamics.

Finally, the methods and the data analysis described in the chapter 4 show the core ideas that should drive - in our opinion - proper event-based computation: time must be considered as the most important dimension of visual event-based data. The software presented in this chapter is a

convenient tool to process event-based data and to analyze inputs and outputs of an algorithm at the scale of a single event. We believe that using such tools can help better understand the fundamentals and dynamics of event-based processing, rather than simply comparing an algorithm output to a ground-truth as it is often the case.

From a more general perspective, we have shown that event-by-event processing allows to compute a visual odometry. More importantly, we have shown that the proper consideration of time is a key ingredient for robust event-based computation. We thus believe that this dynamical approach of time could not only improve existing algorithms that rely on fixed timescales, but also create guidelines for developing more efficient algorithms in the future. It fundamentally switches the way data is seen, from fixed temporal slices composed of binned events, to coherent sets of events with the right amount of information to process.
We have also shown that if the presented framework is not optimized for computation speed, such a development environment allows for a proper, time-based, consideration of the data. It enforces that algorithms are designed with an "event-by-event" approach. As such, we hope that it will help to implement new algorithms based on this new approach to neuromorphic computation.

This finally comes back to the screensaver example presented in figure 3.1. Humans can estimate the motion from a monocular view, simply because all the necessary information is available from this single view. Neuromorphic sensors only change the way the data is being recorded and treated, but the crucial motion information remains. There is no doubt that pose can be recovered from purely passive visual sensors, as long as that data is processed in the right way. We believe that our work helped paving the way towards this fundamental understanding of the data, by presenting a new way of considering it.

Perspectives for Neuromorphic Engineering

As we have seen throughout this work, while many theoretical considerations have been presented, and if most are confirmed by proofs of concept, our models still require more experiments, with additional ground-truth assessment, along with a larger variety of visual scenes. The lack of data-complete stereo datasets to run experiments on, especially with pose information, should be addressed.

An efficiency optimization of our framework is also envisioned. Indeed, even though that framework was never meant to be computationally efficient, as the primary goal has been ergonomy and data management from the get-go, the extremely slow data processing made experiments time-consuming, inherently constraining the number of iterations possible on our algorithms.

Some improvements of the proposed algorithms are also considered. The optical flow, as presented in section 2.2.1, is not as accurate as one could hope. We believe the x/y axis rejection process can be changed by an angle-amplitude one. This should improve the quality of the optical flow, by making it more detached to the square matrix it relies upon. Also, the vidual odometer proposed, if stable in most cases, has shown limitations for extreme variations of the dynamics, where *keypoints* estimated locations get further and further away from their theoretical locations. We believe modifications of the model, allowing a displacement margin for those *keypoints* based upon an estimated error could help both stabilizing the system, while also reducing the constant drift inherent to $3D$ points estimation. Also, setting up *keypoint* recognition to cancel out this drift is a necessary step to improve our method, from a simple pose and displacement estimation to a more

advanced environment interaction tool.

However, the most important work started here - in my opinion - is the timescale study for neuromorphic computation. Its implication extends to many more fields than vision processing, and other sensors like neuromorphic cochleas could also benefit from a *dynamical* approach. For data-driven acquisition, with data being received when changes occur, the study of time should be the cornerstone for event-based algorithmic development.

Finally, I believe that a major investigation has to be made about the way event-based processing can be parallelized. If this can seen counter-intuitive, as the parallel implementation of an algorithm is usually based on the nature of the hardware it runs on, I think we can reverse the problem. We need to find how multiple events could be processed at the same time, while still ensuring causality and data ordering. This, I think, would help designing dedicated neuromorphic hardware, as it would uncover one of its fundamental guidelines. Indeed, neuromorphic hardware does exist, but what struck me during these years of PhD was the lack of development of chips outside of SNNs boards. If SNNs are one of the main neuromorphic developments, and the event-based version of ANNs, they are not the only possibility for event-based computation. As we have shown throughout this work, deterministic algorithms are a class of methods of their own that would greatly benefit from dedicated hardware. However, such hardware is nowhere to be found. Current computers architectures are definitely no optimized for single pixel updates, which are the very basis of this field.

List of Figures

0.1 Autonomous robots comparison . . . 1
0.2 Computer power growth over the 20th century . . . 3
0.3 Event-based acquisition paradigm and sensor . . . 7

1.1 Event-tracker association . . . 14
1.2 Events coordinates reference change and optical flow computation . . . 16
1.3 Contour velocities estimation . . . 18
1.4 Convergence process of a tracker . . . 21
1.5 Tracker properties evolution . . . 24
1.6 Trackers' states relations diagram . . . 28
1.7 Tracking panoramic reconstruction . . . 31
1.8 Tracking IMU reconstruction . . . 32

2.1 Stereo Ego-Kinetics pipeline . . . 37
2.2 Optical flow computation block . . . 39
2.3 Stereo-matching spatial neighbourhood and local descriptor . . . 40
2.4 Statio-temporal maps examples . . . 41
2.5 Stereo-matching computation block . . . 44
2.6 Ego-Kinetics computation block . . . 46
2.7 Translating square optical flow comparison . . . 48
2.8 Stereo-matching testing example . . . 49
2.9 Velocities estimation on a sequence simulated on ESIM. The ground-truth - in dashed -, shows a constant angular velocity about the z-axis, of $-0.3 rad/s$ and a negligible y-translation of $2.10^{-2} m.s^{-1}$. . . . 50
2.10 Ego-rotation results on ESIM and Doll Sequence . . . 51
2.11 Ego-rotation results on handheld walking sequences . . . 52
2.12 Events and disparities images of handheld walking sequences . . . 53

3.1 Screensaver . . . 55
3.2 Pose keypoint initialization . . . 60
3.3 Pose anchors' $3D$ points . . . 61
3.4 Standard vs. modified springs mechanical properties . . . 62
3.5 Wrong corrections situations . . . 64
3.6 System equivalent constraints field . . . 65
3.7 Equivalent mechanical system . . . 66
3.8 2-sensors visual odometry block . . . 67
3.9 Synthetic motion example . . . 69
3.10 Noiseless synthetic odometry results . . . 70
3.11 Noised synthetic odometry results . . . 71

4.1 Ideal events lifetimes . . . 76
4.2 Static timescales pipeline example . . . 77
4.3 Adaptive timescales pipeline example . . . 78
4.4 Constant vs adaptive timescale frames results comparison . . . 83
4.5 Elementary event example . . . 88
4.6 Multi-data packed event example . . . 89
4.7 PEBBLE adaptive timescale implementation . . . 92
4.8 Dedicated event stream display . . . 94

Bibliography

[1] J. Weng, N. Ahuja, and T. Huang, "Cresceptron: a self-organizing neural network which grows adaptively," in *[Proceedings 1992] IJCNN International Joint Conference on Neural Networks*, vol. 1, 1992, pp. 576–581 vol.1. 2

[2] Q. Le, M. Ranzato, R. Monga, M. Devin, K. Chen, G. Corrado, J. Dean, and A. Ng, "Building high-level features using large scale unsupervised learning," *Proceedings of ICML*, vol. 1, 12 2011. 2

[3] Y. LeCun, L. Bottoua, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, no. 86(11):2278-2324, 1998. 2

[4] J. George, A. Cyril, B. I. Koshy, and L. Mary, "Exploring sound signature for vehicle detection and classification using ann," *International Journal of Soft Computing*, vol. 4, pp. 29–36, 2013. 2

[5] A. Patnaik and R. Mishra, "Ann techniques in microwave engineering," *IEEE Microwave Magazine*, vol. 1, no. 1, pp. 55–60, 2000. 2

[6] K. Tunyasuvunakool, J. Adler, Z. Wu, T. Green, M. Zielinski, A. Žídek, A. Bridgland, A. Cowie, C. Meyer, A. Laydon, S. Velankar, G. Kleywegt, A. Bateman, R. Evans, A. Pritzel, M. Figurnov, O. Ronneberger, R. Bates, S. Kohl, and D. Hassabis, "Highly accurate protein structure prediction for the human proteome," *Nature*, vol. 596, pp. 1–9, 08 2021. 2

[7] J. Koomey, S. Berard, M. Sanchez, and H. Wong, "Implications of historical trends in the electrical efficiency of computing," *Annals of the History of Computing, IEEE*, vol. 33, pp. 46 – 54, 04 2011. 3

[8] W. B. Levy and V. G. Calvert, "Communication consumes 35 times more energy than computation in the human cortex, but both costs are needed to predict synapse number," *Proceedings of the National Academy of Sciences*, vol. 118, no. 18, 2021. [Online]. Available: https://www.pnas.org/content/118/18/e2008173118 4

[9] C. A. Mead and M. A. Mahowald, "A silicon model of early visual processing," *Neural networks*, vol. 1, no. 1, pp. 91–97, 1988. 4, 7

[10] P. Lichtsteiner, C. Posch, and T. Delbruck, "A 128x128 120 db 15 μs latency asynchronous temporal contrast vision sensor," *IEEE Journal of Solid-State Circuits*, vol. 43, no. 2, pp. 566–576, 2008. 4, 8

[11] C. Posch, D. Matolin, and R. Wohlgenannt, "A qvga 143 db dynamic range frame-free pwm image sensor with lossless pixel-level video compression and time-domain cds," *IEEE Journal of Solid-State Circuits*, vol. 46, no. 1, pp. 259–275, Jan 2011. [Online]. Available: http://dx.doi.org/10.1109/jssc.2010.2085952 4, 7, 8, 69

[12] ——, "An asynchronous time-based image sensor," *Circuits and Systems*, 2008. 4, 8, 90

[13] G. Lenz, S.-H. Ieng, and R. Benosman, “Event-based dynamic face detection and tracking based on activity,” *Frontiers in Neuroscience*, 03 2018. 4, 8

[14] C. Santos, J. A. Jiménez, and F. Espinosa, “Effect of event-based sensing on iot node power efficiency. case study: Air quality monitoring in smart cities,” *IEEE Access*, vol. 7, pp. 132 577–132 586, 2019. 4

[15] A. Amir, B. Taba, D. Berg, T. Melano, J. McKinstry, C. Di Nolfo, T. Nayak, A. Andreopoulos, G. Garreau, M. Mendoza, J. Kusnitz, M. Debole, S. Esser, T. Delbruck, M. Flickner, and D. Modha, “A low power, fully event-based gesture recognition system,” in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, July 2017. 4

[16] G. Gallego, T. Delbrück, G. Orchard, C. Bartolozzi, B. Taba, A. Censi, S. Leutenegger, A. J. Davison, J. Conradt, K. Daniilidis, and D. Scaramuzza, “Event-based vision: A survey,” *CoRR*, vol. abs/1904.08405, 2019. [Online]. Available: http://arxiv.org/abs/1904.08405 4

[17] P. Shor, “Algorithms for quantum computation: discrete logarithms and factoring,” in *Proceedings 35th Annual Symposium on Foundations of Computer Science*, 1994, pp. 124–134. 4

[18] L. K. Grover, “A fast quantum mechanical algorithm for database search,” in *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*, ser. STOC ’96. New York, NY, USA: Association for Computing Machinery, 1996, p. 212–219. [Online]. Available: https://doi.org/10.1145/237814.237866 4

[19] M. Zaffar, S. Ehsan, R. Stolkin, and K. M. Maier, “Sensors, slam and long-term autonomy: A review,” in *2018 NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, 2018, pp. 285–290. 5, 11

[20] M. Mahowald, “Vlsi analogs of neuronal visual processing: a synthesis of form and function,” Ph.D. dissertation, California Institute of Technology Pasadena, 1992. 7

[21] T. Delbruck, “Silicon retina with correlation-based, velocity-tuned pixels,” *IEEE Transactions on Neural Networks*, vol. 4, no. 3, pp. 529–541, 1993. 8

[22] R. Etienne-Cummings, J. Van der Spiegel, and P. Mueller, “A focal plane visual motion measurement sensor,” *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, vol. 44, no. 1, pp. 55–66, 1997. 8, 80

[23] J. Krammer and C. Koch, “Pulse-based analog vlsi velocity sensors,” *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, vol. 44, no. 2, pp. 86–101, 1997. 8, 80

[24] P. Lichtsteiner, C. Posch, and T. Delbrück, “A 128 × 128 120db 30mw asynchronous vision sensor that responds to relative intensity change,” in *IEEE International Solid State Circuits Conference - Digest of Technical Papers (ISSC)*, 2006, pp. 2060–2069. 8

[25] K. A. Boahen, “Point-to-point connectivity between neuromorphic chips using address-events,” *IEEE Trans. Circuits Syst. II*, pp. 416–434, 2000. 8

[26] M.Guo, J. Huang, and S. Chen, “Live demonstration: A 768x640 pixels 200meps dynamic vision sensor,” in *IEEE International Symposium on Circuits and Systems. Baltimore, MD, USA*. IEEE, 2017. 8

[27] C. Brandli, R. Berner, M. Yang, S.-C. Liu, and T. Delbruck, “A 240 x 180 130 db 3 µs latency global shutter spatiotemporal vision sensor,” *Solid-State Circuits, IEEE Journal of*, vol. 49, pp. 2333–2341, 10 2014. 8

[28] R. Sarpeshkar, R. F. Lyon, and C. Mead, *A Low-Power Wide-Dynamic-Range Analog VLSI Cochlea.* Boston, MA: Springer US, 1998, pp. 49–103. [Online]. Available: https://doi.org/10.1007/978-0-585-28001-1_3 8

[29] R. Lyon and C. Mead, "An analog electronic cochlea," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 36, no. 7, pp. 1119–1134, 1988. 8

[30] E. Fragniere, "A 100-channel analog cmos auditory filter bank for speech recognition," in *ISSCC. 2005 IEEE International Digest of Technical Papers. Solid-State Circuits Conference, 2005.*, 2005, pp. 140–589 Vol. 1. 8

[31] A. van Schaik and S.-C. Liu, "Aer ear: a matched silicon cochlea pair with address event representation interface," in *2005 IEEE International Symposium on Circuits and Systems*, 2005, pp. 4213–4216 Vol. 5. 8

[32] B. Bhattacharya, C. Patterson, F. Galluppi, S. Durrant, and S. Furber, "Engineering a thalamo-cortico-thalamic circuit on spinnaker: a preliminary study toward modeling sleep and wakefulness," *Frontiers in Neural Circuits*, vol. 8, 05 2014. 8

[33] M. Davies, N. Srinivasa, T.-H. Lin, G. Chinya, Y. Cao, S. H. Choday, G. Dimou, P. Joshi, N. Imam, S. Jain, Y. Liao, C.-K. Lin, A. Lines, R. Liu, D. Mathaikutty, S. McCoy, A. Paul, J. Tse, G. Venkataramanan, Y.-H. Weng, A. Wild, Y. Yang, and H. Wang, "Loihi: A neuromorphic manycore processor with on-chip learning," *IEEE Micro*, vol. 38, no. 1, pp. 82–99, 2018. 8

[34] X. Lagorce and R. Benosman, "Spike time interval computational kernel, a framework for general purpose computation using neurons, precise timing, delays, and synchrony," *Neural Computation*, vol. 27, no. 11, pp. 2261–2317, 2015. 8

[35] X. Clady, J.-M. Maro, S. Barré, and R. B. Benosman, "A motion-based feature for event-based pattern recognition," *Frontiers in Neuroscience*, vol. 10, p. 594, 2017. [Online]. Available: https://www.frontiersin.org/article/10.3389/fnins.2016.00594 8

[36] X. Lagorce, G. Orchard, F. Galluppi, B. E. Shi, and R. Benosman, "Hots: A hierarchy of event-based time-surfaces for pattern recognition," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2016. 8

[37] T. Stoffregen and L. Kleeman, "Event cameras, contrast maximization and reward functions: An analysis," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2019. 8

[38] M. Liu and T. Delbrück, "Adaptive time-slice block-matching optical flow algorithm for dynamic vision sensors," in *BMVC*, 2018. 8, 80

[39] S. A. S. Mohamed, M.-H. Haghbayan, J. Heikkonen, H. Tenhunen, and J. Plosila, "Towards real-time edge detection for event cameras based on lifetime and dynamic slicing," in *Proceedings of the International Conference on Artificial Intelligence and Computer Vision (AICV2020)*, A.-E. Hassanien, A. T. Azar, T. Gaber, D. Oliva, and F. M. Tolba, Eds. Cham: Springer International Publishing, 2020, pp. 584–593. 8

[40] G. Gallego, T. Delbrück, G. Orchard, C. Bartolozzi, B. Taba, A. Censi, S. Leutenegger, A. J. Davison, J. Conradt, K. Daniilidis, and D. Scaramuzza, "Event-based vision: A survey," *IEEE transactions on pattern analysis and machine intelligence*, vol. PP, 2020. 8

[41] D. Tedaldi, G. Gallego, E. Mueggler, and D. Scaramuzza, "Feature detection and tracking with the dynamic and active-pixel vision sensor (davis)," in *2016 Second International Conference on Event-based Control, Communication, and Signal Processing (EBCCSP)*. IEEE, 2016, pp. 1–7. 8

[42] A. I. Maqueda, A. Loquercio, G. Gallego, N. García, and D. Scaramuzza, "Event-based vision meets deep learning on steering prediction for self-driving cars," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018, pp. 5419–5427. 8

[43] H. Rebecq, R. Ranftl, V. Koltun, and D. Scaramuzza, “Events-to-video: Bringing modern computer vision to event cameras,” in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2019, pp. 3857–3866. 8

[44] C. Scheerlinck, H. Rebecq, D. Gehrig, N. Barnes, R. Mahony, and D. Scaramuzza, “Fast image reconstruction with an event camera,” in *The IEEE Winter Conference on Applications of Computer Vision*, 2020, pp. 156–163. 8

[45] X. Lagorce, G. Orchard, F. Galluppi, B. E. Shi, and R. Benosman, “Hots: A hierarchy of event-based time-surfaces for pattern recognition,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2016. 9

[46] G. Haessig and R. Benosman, “A sparse coding multi-scale precise-timing machine learning algorithm for neuromorphic event-based sensors,” in *Micro-and Nanotechnology Sensors, Systems, and Applications X*, vol. 10639. International Society for Optics and Photonics, 2018, p. 106391U. 9

[47] S. Afshar, Y. Xu, J. Tapson, A. van Schaik, and G. Cohen, “Event-based feature extraction using adaptive selection thresholds,” *arXiv preprint arXiv:1907.07853*, 2019. 9

[48] B. Lucas and T. Kanade, “An iterative image registration technique with an application to stereo vision,” in *Proceedings of the 7th International Joint Conference on Artificial Intelligence (IJCAI ’81)*, vol. 81, 04 1981. 9

[49] H. Xue, Y. Liu, D. Cai, and X. He, “Tracking people in rgbd videos using deep learning and motion clues,” *Neurocomputing*, vol. 204, pp. 70–76, 2016, big Learning in Social Media Analytics. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0925231216300959 9

[50] X. Clady, S. Ieng, and R. Benosman, “Asynchronous event-based corner detection and matching.” *Neural Computation*, vol. 66, pp. 91–106, 2015. 9

[51] V. Vasco, A. Glover, and C. Bartolozzi, “Fast event-based harris corner detection exploiting the advantages of event-driven cameras,” in *International Conference on Intelligent Robots and Systems*. IEEE, 2016, pp. 4144–4149. 9

[52] R. Benosman, C. Clercq, X. Lagorce, S.-H. Ieng, and C. Bartolozzi, “Event-based visual flow,” *IEEE Transactions on Neural Networks and Learning Systems*, vol. 25, no. 2, pp. 407–417, Feb 2014. [Online]. Available: http://dx.doi.org/10.1109/tnnls.2013.2273537 9, 10, 16, 37, 47, 80

[53] E. Mueggler, C. Bartolozzi, and D. Scaramuzza, “Fast event-based corner detection,” in *British Machine Vision Conference*, 2017. 9

[54] E. Rosten and T. Drummond, “Machine learning for high-speed corner detection,” in *European Conference on Computer Vision. Graz, Austria*, 2006, pp. 430–443. 9, 14

[55] I. Alzugaray and M. Chli, “Asynchronous corner detection and tracking for event cameras in real time,” *IEEE Robotics and Automation Letters*, vol. 3, no. 4, p. 3177–3184, Oct 2018. [Online]. Available: http://dx.doi.org/10.1109/lra.2018.2849882 9

[56] ——, “Ace: An efficient asynchronous corner tracker for event cameras,” in *2018 International Conference on 3D Vision (3DV)*, 2018, pp. 653–661. 9, 29, 71

[57] C.Brandli, R.Berner, M.Yang, S.C.Liu, and T.Delbruck, “A240x180 130db 3 μs latency global shutter spatiotemporal vision sensor„” *IEEE Journal of Solid-State Circuits*, vol. 49, no. 10, pp. 2333—2341, 2014. 9, 29

[58] D. Tedaldi, G. gallego, E. Mueggler, and D. Scaramuzza, "Feature detection and tracking with the dynamic and active-pixel vision sensor (davis)," in *International Conference on Event-based Control, Communication, and Signal Processing. Krakow, Poland.* IEEE, 2016. 9

[59] D. Gehrig, H. Rebecq, G. Gallego, and D. Scaramuzza, "Asynchronous, photometric feature tracking using events and frames," in *Proceedings of the European Conference on Computer Vision (ECCV)*, September 2018. 9

[60] A. Z. Zhu, N. Atanasov, and K. Daniilidis, "Event-based feature tracking with probabilistic data association," in *International Conference on Robotics and Automation. Singapore.* IEEE, 2017. 9

[61] Y. Chen and G. Medioni, "Object modelling by registration of multiple range images," *Image Vision Comput.*, vol. 10, no. 3, pp. 145–155, Apr. 1992. [Online]. Available: http://dx.doi.org/10.1016/0262-8856(92)90066-C 9

[62] G. Gallego, H. Rebecq, and D. Scaramuzza, "A unifying contrast maximization framework for event cameras, with applications to motion, depth, and optical flow estimation," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018. 9

[63] R. Benosman, S.-H. Ieng, C. Clercq, C. Bartolozzi, and M. Srinivasan, "Asynchronous frameless event-based optical flow," *Neural Networks*, vol. 27, pp. 32–37, 2012. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0893608011002930 10

[64] A. Zhu, D. Thakur, T. Ozaslan, B. Pfrommer, V. Kumar, and K. Daniilidis, "The multi vehicle stereo event camera dataset: An event camera dataset for 3d perception," *IEEE Robotics and Automation Letters*, vol. 3, no. 3, pp. 2032–2039, 2018. 10

[65] C. Lee, A. K. Kosta, A. Z. Zhu, K. Chaney, K. Daniilidis, and K. Roy, "Spike-flownet: Event-based optical flow estimation with energy-efficient hybrid neural networks," 2020. 10

[66] H. Akolkar, S. Ieng, and R. Benosman, "Real-time high speed motion prediction using fast aperture-robust event-driven visual flow," *IEEE Transaction on Pattern Analysis and Machine Intelligence*, 2020. 10, 37, 45

[67] A. Rajagopalan and S. Chaudhuri, "A block shift-variant blur model for recovering depth from defocused images," in *Proceedings., International Conference on Image Processing*, vol. 3, 1995, pp. 636–639 vol.3. 10

[68] G. Haessig, X. Berthelon, S.-H. Ieng, and R. Benosman, "A spiking neural network model of depth from defocus for event-based neuromorphic vision," *Scientific Reports*, vol. 9, p. 3744, 03 2019. 10

[69] S. T. Barnard and W. B. Thompson, "Disparity analysis of images," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. PAMI-2, no. 4, pp. 333–340, 1980. 10

[70] R. Benosman, S.-H. Sio-Hoï Ieng, P. Rogister, and C. Posch, "Asynchronous event-based hebbian epipolar geometry," *IEEE Transactions on Neural Networks*, vol. 22, no. 11, pp. 1723–1734, 2011. 10

[71] P. Rogister, R. Benosman, S.-H. Ieng, P. Lichtsteiner, and T. Delbruck, "Asynchronous event-based binocular stereo matching," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 23, no. 2, pp. 347–353, 2012. 10

[72] S. Tulyakov, F. Fleuret, M. Kiefel, P. Gehler, and M. Hirsch, "Learning an event sequence embedding for dense event-based deep stereo," in *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, October 2019. 10

[73] A. Z. Zhu, L. Yuan, K. Chaney, and K. Daniilidis, "Unsupervised event-based learning of optical flow, depth, and egomotion," *CoRR*, vol. abs/1812.08156, 2018. [Online]. Available: http://arxiv.org/abs/1812.08156 10

[74] Y. Zhou, G. Gallego, and S. Shen, "Event-based stereo visual odometry," *IEEE Transactions on Robotics*, 2021. 10, 11

[75] M. Osswald, S. Ieng, R. Benosman, and G. Indiveri, "A spiking neural network model of 3d perception for event-based neuromorphic stereo vision systems," *Scientific Report*, vol. 7, no. 40703, 2017. 10

[76] H. Kim, S. Leutenegger, and A. J. Davison, "Real-time 3d reconstruction and 6-dof tracking with an event camera," in *Proceedings of the European Conference on Computer Vision*, 2016, pp. 349—-364. 11

[77] H. Rebecq, D. Gehrig, and D. Scaramuzza, "ESIM: an open event camera simulator," in *Conf. on Robotics Learning (CoRL)*, 2018. 11

[78] A. R. Vidal, H. Rebecq, T. Horstschaefer, and D. Scaramuzza, "Ultimate slam? combining events, images, and imu for robust visual slam in hdr and high speed scenarios," *IEEE Robotic Automation Letter*, vol. 3, pp. 994–1001, 2018. 11

[79] E. Mueggler, G. Gallego, H. Rebecq, and D. Scaramuzza, "Continuous-time visual-inertial odometry for event cameras," *IEEE Transaction on robotics*, vol. 34, pp. 1425–1440, 2018. 11

[80] D. Migliore, "Sensing the world with event-based cameras," in *ICRA Workshop on Sensing, Estimating and Understanding the Dynamic World*, 2020. 11

[81] H. Rebecq, T. Horstschaefer, G. Gallego, and D. Scaramuzza, "Evo: A geometric approach to event-based 6-dof parallel tracking and mapping in real-time," *Ieee Robotics and Automation Letters*, 2016. 11

[82] A. Vitale, A. Renner, C. Nauer, D. Scaramuzza, and Y. Sandamirskaya, "Event-driven vision and control for uavs on a neuromorphic chip," 2021. 11

[83] C. Harris and M. Stephens, "A combined corner and edge detector," *Proceedings of the 4th Alvey Vision Conference*, pp. 147–151, 1988. 14

[84] X. Clady, S.-H. Ieng, and R. Benosman, "Asynchronous event-based corner detection and matching," *Neural Networks*, vol. 66, pp. 91–106, 2015. 16

[85] P. Liu, M. R. Lyu, I. King, and J. Xu, "Selflow: Self-supervised learning of optical flow," *CoRR*, vol. abs/1904.09117, 2019. [Online]. Available: http://arxiv.org/abs/1904.09117 16

[86] R. Benosman, C. Clercq, X. Lagorce, S.-H. Ieng, and C. Bartolozzi, "Event-based visual flow," *IEEE Transactions on Neural Networks*, vol. pp, p. 1, 11 2013. 16, 81

[87] H. Akolkar, S. Ieng, and R. Benosman, "See before you see: Real-time high speed motion prediction using fast aperture-robust event-driven visual flow," *CoRR*, vol. abs/1811.11135, 2018. [Online]. Available: http://arxiv.org/abs/1811.11135 16, 80

[88] E. Mueggler, H. Rebecq, G. Gallego, T. Delbruck, and D. Scaramuzza, "The event-camera dataset and simulator: Event-based data for pose estimation, visual odometry, and slam," *The International Journal of Robotics Research*, vol. 36, no. 2, p. 142–149, Feb 2017. [Online]. Available: http://dx.doi.org/10.1177/0278364917691115 28, 29, 30, 32

[89] C. Posch, D. Matolin, and R. Wohlgenannt, "A qvga 143 db dynamic range frame-free pwm image sensor with lossless pixel-level video compression and time-domain cds." *J. Solid-State Circuits*, vol. 46, no. 1, pp. 259–275, 2011. [Online]. Available: http://dblp.uni-trier.de/db/journals/jssc/jssc46.html#PoschMW11 29

[90] A. Lukezic, T. Vojir, L. Cehovin Zajc, J. Matas, and M. Kristan, “Discriminative correlation filter with channel and spatial reliability,” in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, July 2017. 29

[91] H. Kim, A. Handa, R. Benosman, S.-H. Ieng, and A. J. Davison, “Simultaneous mosaicing and track- ing with an event camera,” in *British Machine Vis. Conf. (BMVC)*, 2014. 36

[92] C. Reinbacher, G. Munda, and T. Pock, “Real-time panoramic tracking for event cameras,” in *IEEE International Conf. Comput. Photography (ICCP)*, 2017, pp. 1–9. 36

[93] G. Gallego and D. Scaramuzza, “Accurate angular velocity estimation with an event camera,” *IEEE Robotics and Automation Letters*, vol. 2, no. 2, pp. 632–639, 2017. 36

[94] I. Cvišić, J. Ćesić, I. Marković, and I. Petrović, “Soft-slam: Computationally efficient stereo visual simultaneous localization and mapping for autonomous unmanned aerial vehicles,” *Journal of Field Robotics*, vol. 35, no. 4, pp. 578–595, 2018. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/rob.21762 55

[95] R. Munguía and A. Grau, “Monocular slam for visual odometry: A full approach to the delayed inverse-depth feature initialization method,” *Mathematical Problems in Engineering*, vol. 2012, 09 2012. 56

[96] T. Y. Yeh, G. Reinman, S. J. Patel, and P. Faloutsos, “Fool me twice: Exploring and exploiting error tolerance in physics-based animation,” *ACM Trans. Graph.*, vol. 29, no. 1, Dec. 2009. [Online]. Available: https://doi.org/10.1145/1640443.1640448 57

[97] D. Reverter Valeiras, S. Kime, S.-H. Ieng, and R. B. Benosman, “An event-based solution to the perspective-n-point problem,” *Frontiers in Neuroscience*, vol. 10, p. 208, 2016. [Online]. Available: https://www.frontiersin.org/article/10.3389/fnins.2016.00208 59

[98] D. Reverter Valeiras, X. Lagorce, X. Clady, C. Bartolozzi, S.-H. Ieng, and R. Benosman, “An asynchronous neuromorphic event-driven visual part-based shape tracking,” *IEEE Transactions on Neural Networks and Learning Systems*, vol. 26, no. 12, pp. 3045–3059, 2015. 61

[99] J. Costas-Santos, T. Serrano-Gotarredona, R. Serrano-Gotarredona, and B. Linares-Barranco, “A spatial contrast retina with on-chip calibration for neuromorphic spike-based aer vision systems,” *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 54, no. 7, pp. 1444–1458, 2007. 69

[100] I. Brouk, A. Nemirovsky, and Y. Nemirovsky, “Analysis of noise in cmos image sensor,” in *2008 IEEE International Conference on Microwaves, Communications, Antennas and Electronic Systems*, 2008, pp. 1–8. 70

[101] H. Tian, B. Fowler, and A. Gamal, “Analysis of temporal noise in cmos photodiode active pixel sensor,” *IEEE Journal of Solid-State Circuits*, vol. 36, no. 1, pp. 92–101, 2001. 70

[102] A. Khodamoradi and R. Kastner, “Space spatiotemporal filter for reducing noise in neuromorphic vision sensors,” *IEEE Transactions on Emerging Topics in Computing*, vol. 9, no. 1, pp. 15–23, 2021. 70

[103] E. Mueggler, C. Forster, N. Baumli, G. Gallego, and D. Scaramuzza, “Lifetime estimation of events from dynamic vision sensors,” in *2015 IEEE International Conference on Robotics and Automation (ICRA)*, 2015, pp. 4874–4881. 74, 78

[104] G. Gallego, H. Rebecq, and D. Scaramuzza, “A unifying contrast maximization framework for event cameras, with applications to motion, depth, and optical flow estimation,” in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018. 81

[105] A. Marcireau, S.-H. Ieng, and R. Benosman, "Sepia, tarsier, and chameleon: A modular c++ framework for event-based computer vision," *Frontiers in Neuroscience*, vol. 13, 2020. [Online]. Available: https://www.frontiersin.org/article/10.3389/fnins.2019.01338 86

[106] H. Rebecq, D. Gehrig, and D. Scaramuzza, "ESIM: an open event camera simulator," *Conf. on Robotics Learning (CoRL)*, Oct. 2018. 90