Event-based Line Tracking with Neuromorphic Systems

A thesis
Submitted towards the partial fullfilment of
BS-MS dual degree programme
by

HARINI SUDHA J G

DATE: 30/04/2023

under the guidance of

JÖRG CONRADT

KTH ROYAL INSTITUTE OF TECHNOLOGY

from May 2022 to Mar 2023

INDIAN INSTITUTE OF SCIENCE EDUCATION AND RESEARCH PUNE

Certificate

This is to certify that this dissertation entitled **Event-based line segment detection for neuromorphic applications** submitted towards the partial fulfillment of the BS-MS degree at the Indian Institute of Science Education and Research, Pune represents original research carried out by **Harini Sudha J G** at **KTH Royal Institute of Technology, Stockholm**, under the supervision of **Jorg Conradt** during academic year May 2022 to March 2023.

J. Conradt

Jörg Conradt
asst. professor
kth royal
institute of
technology

[illegible]

Harini Sudha J G
20181033
bs-ms
iiser pune

Date: 30/04/2023

Declaration

I, hereby declare that the matter embodied in the report titled **Event-based Line Tracking with Neuromorphic Systems** is the result of the investigations carried out by me at the **KTH Royal Institute of Technology** from the period 01-07-2022 to 30-04-2023 under the supervision of **Jörg Conradt** and the same has not been submitted elsewhere for any other degree.

J. Conradt

Jörg Conradt
asst. professor
kth royal
institute of
technology

[illegible]

Harini Sudha
20181033
BS-MS
IISER Pune

Date: 30/04/2023

This thesis is dedicated to people who pursue things outside their comfort zone

Acknowledgements

I'm grateful to have had the opportunity to work with compassionate, intellectual and incredible people over the duration of this thesis. They've shown me the world of opportunities and that doing science is as important as having a life and self-worth outside of it. I'm indebted to my supervisor Jörg who went to great lengths to get me to Sweden, managed logistical difficulties and ensured I had everything so that I could focus on science. He showed me that he would support me on days when I didn't have results as enthusiastically as on the days I did. He was always available to discuss even the most minor details. I'd like to thank Jens, Juan, Emil and Dimitris for being wonderful labmates; always helpful. I learnt to maintain a codebase through Jens, and I aspire to write clean, structured code like him. I've yet to learn to become a disciplined morning person like Juan. Emil showed me that age is just a number, and you can do anything you want in life. Dimitris was just a door away for quick brainstorming sessions. The knowledge exchange with many visitors to our lab made doing science more enjoyable, especially Matthew Cook and Xander.

Science happens in between the messiness of life. This thesis would have been impossible without my support system, who helped me manage life outside of the thesis. Arun was always one call away during my breakdowns and brought me a daily dose of comfort. I'll reminisce about the crazy inter-euro adventures with Wridhdhi. Varna was always there to relate to my struggles and rant hours about work and life. Terence was there as my go-to person in Stockholm, from house shifting to hiking to cooking plans. I'm thankful for having a familiarity of home just a short flight away with Rabin, Amit, Anuja, and Madhesh. There are lots of friends I made in Stockholm who made me look forward to every day.

Finally, I'd like to thank my parents for giving me the wings to fly in this whole wide world and IISER for building my confidence to make the world mine.

Contributions

Contributor name	Contributor role
Harini Sudha J G, Jörg Conradt	Conceptualization Ideas
Harini Sudha J G, Jörg Conradt	Methodology
Harini Sudha J G	Software
Harini Sudha J G	Validation
Harini Sudha J G	Formal analysis
Harini Sudha J G	Investigation
Jörg Conradt	Resources
Harini Sudha J G	Data Curation
Harini Sudha J G	Writing - original draft preparation
Harini Sudha J G, Jörg Conradt	Writing - review and editing
Harini Sudha J G	Visualization
Jörg Conradt	Supervision
-	Project administration
Jörg Conradt	Funding acquisition

This contributor syntax is based on the Journal of Cell Science CRediT Taxonomy[1]

[1]https://journals.biologists.com/jcs/pages/author-contributions

Abstract

Neuromorphic computing is an emerging field that implements bio-inspired perception, computation and processing with the potential of low power and low latency processing. Current traditional computing and hardware are optimized for each other, but they are not well suited to bring out the efficiency of neuromorphic processing principles. In this thesis, we develop an event-based line and line segment tracking algorithm that captures the low-level features of a scene, on top of which other neuromorphic embedded vision tasks can be developed. We use the Dynamic Vision Sensor (DVS) event camera to obtain low-latency inputs. Our algorithm is a Hough Transform implemented as Spiking Neural Networks (SNNs) to detect lines. We use SNNs to leverage asynchronous event camera inputs and perform sparse computation. SNNs enable low power consumption due to the parallel processing of each neuron when implemented on neuromorphic hardware. The number of spiking neurons is minimized to reduce power consumption further. Fixing the number of neurons makes the Hough space coarsely discretized, in which population coding techniques allow finer discretisations. The algorithm takes in variable high-resolution event camera inputs with minimal loss of accuracy. We utilize artificial, recorded, and real-time event camera data to check the functionality of our algorithm running on a GPU, which makes parallel processing of SNN possible. We also show promising results of line-tracking running on the SpiNNaker neuromorphic hardware with limited neurons. We propose extensions to the algorithm to detect line segments and conduct preliminary experiments to evaluate the idea. Hence, we have designed a building block neuromorphic algorithm that paves the way for on-edge neuromorphic applications requiring low latency and low power consumption.

Contents

Chapter 1

Introduction

1.1 Motivation and Outline

There have been many exciting developments in the field of AI in recent years. One of the most notable is the rapid advancement of deep learning algorithms, which have enabled breakthroughs in a variety of applications such as image recognition, natural language processing, and speech recognition. Generative Pre-trained Transformer-3 (GPT-3), the language model by OpenAI to produce human-like text, consists of 175 billion parameters. It took an estimated 355 Graphics Processing Unit (GPU)-years, 4.6m dollars and 1 GWh of energy to train it.(Lenz 2021) The training of such modern AI models consumes enormous amounts of energy, and these energy requirements are growing astonishingly. The computational resources needed to produce a best-in-class AI model have, on average, doubled every 3.4 months, as illustrated in Figure1.1. The computing community is looking for new ways to enable performance scaling for AI models.

The brain of a fruit fly consumes microwatts of power while performing real-time intricate and disparate tasks such as flight control, path planning, predator avoidance, and food and mate search(Scheffer 2021). We can learn some principles from the brain to help us reduce several gigawatts of power our current artificial systems use. When it comes to real-time action perception tasks, current systems cannot compete with the reaction times and low latency that the brain offers(Hendy and Merkel 2022). Neuromorphic computing is an emerging field that takes inspiration from biological systems; rethinks how perception, computation and processing can be done from the ground up. The broad focus of this thesis is to look at neuromorphic vision and perception.

Neuromorphic systems are electronic and silicon implementations of neural systems, ranging from vision to sensorimotor actions. Many companies and global projects are heavily investing in building neuromorphic systems to leverage the advantages of low power consumption and real-time application benefits. Big players in processor manufacturing, like IBM and Intel, have invested in building digital neuromorphic chips like Loihi(Davies et al. 2018), Lava(Law n.d.), and TrueNorth(DeBole et al. 2019). Some lead-

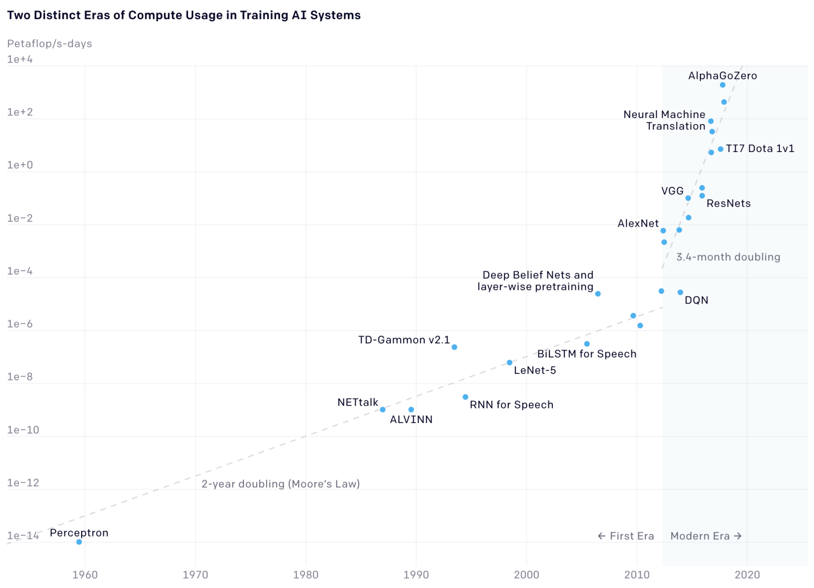


Figure 1.1: Popular Machine Learning models(*AI and compute* 2023): *Number of total operations needed to train some of the most well-known models in computer vision, natural language processing and reinforcement learning.*

ing academic research projects have contributed to open-source neuromorphic hardware like SpiNNaker(Furber et al. 2013), BrainScaleS(Grübl et al. 2020), and DYNAPs(Moradi et al. 2018). The chips and hardware are inspired by the structural connectivity of the brain. Sony, IniVation, and Prophesee are leading players developing event cameras inspired by the human eye. But when these systems are used separately in a traditional computation pipeline, they do not yield the phenomenal results we dream of achieving. This is because current algorithms and hardware are co-optimised for each other, keeping in mind the constraints and requirements of traditional hardware.

For efficient real-world applications of neuromorphic systems, we require specially designed algorithms that leverage the neuromorphic paradigm. Such algorithms draw inspiration from the computational principles of the brain. A simple neuromorphic vision pipeline can consist of a bio-inspired sensing system like event cameras as input, algorithm design inspired by the asynchronous decentralised processing of neurons like Spiking Neural Networks, and hardware capable of parallel processing like GPUs, or neuromorphic hardware. Optimising brain-inspired algorithm design for a neuromorphic pipeline will push us toward real-time low-power computing.

This thesis aims to design a building block neuromorphic algorithm for neuromorphic vision processing, on top of which other embedded vision tasks of object classification, collision avoidance, pattern recognition, etc., can be implemented. One of the traditional ways that people approach embedded vision tasks in traditional Computer Vision is first to identify low-level features like lines and line segments that can best represent a scene. They use these low-level features to interpret the scene further to perform the task of desire(Yan et al. 2019)(Zhou et al. 2021). Similarly, neurons in the brain detect orientations and edges as low-level features that are processed further(Reinagel 2001). Also, neuroscience studies show that in the initial layers of the visual cortex, line segment representations are equivalent to the original scenes they depict (Sayim and Cavanagh 2011). The biological and mathematical evidence convinces us that a line-tracking algorithm can be used as a pre-processing unit for the neuromorphic paradigm. Most current algorithms work for frame-based traditional camera inputs. But, in a neuromorphic pipeline, we have a different type of input from bio-inspired sensors - event cameras; we only get information about the changes in a scene rather than an image frame(Gallego et al. 2022). We must develop event-based algorithms that process sparse and asynchronous event camera inputs. This thesis is an effort to use inspirations from Neuroscience and geometric Computer Vision to develop a line and line segment tracking algorithm that works efficiently for the neuromorphic vision pipeline.

Contributions of this thesis:

- Processing real-time event camera inputs with Spiking Neural Networks.

- Line detection that runs on GPUs and Neuromorphic Hardware with significantly less computing time for real-time processing.

- Line segment detection that is adaptable to the neuromorphic paradigm.

The rest of Chapter 1 explains neuromorphic systems in a simple neuromorphic vision pipeline. The rest of the thesis is organised into 4 Chapters. Chapter 2 explains the line detection and tracking algorithm suitable for the neuromorphic pipeline. Chapter 3 proposes extending the line detection algorithm to detect line segments. Chapter 4 presents the qualitative and quantitative results and analysis of the line tracking algorithm. Chapter 5 concludes the thesis by summarising the outcomes and provides future directions that can be explored.

1.2 Event Cameras

The human retina efficiently encodes visual information by reducing 125 million photoreceptor outputs to just 1 million ganglion cells(Posch et al. 2014). This is achieved by grouping photoreceptor outputs into receptive fields(a set of stimuli that activate a neuron) of different sizes for each ganglion cell. The ganglion cells transmit information about the spatial contrast due to their organisation into centre and surround cells. This means that we wouldn't be able to see when there is no contrast in a scene, i.e. in a brightness and motion static world. But, our eye employs rapid eye movements called micro-saccades(Rolfs 2009). These tiny eye movements (that we are unaware of) ensure there's always contrast detection possible.

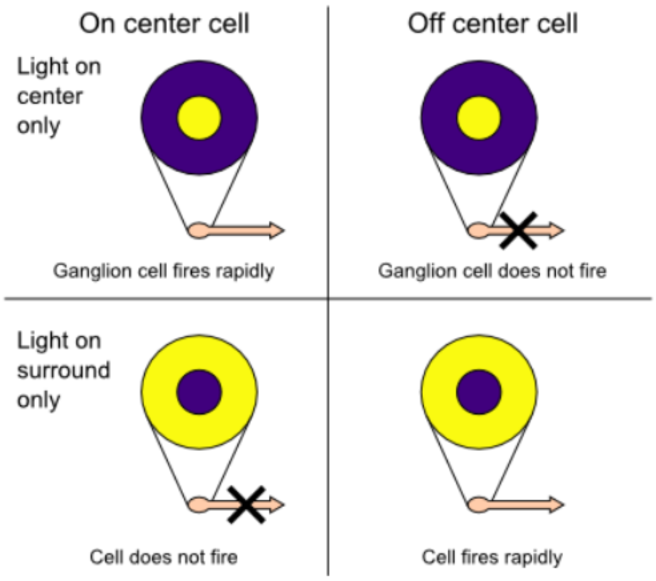


Figure 1.2: Centre surround receptive fields(Lenz 2021): *On and off centre cells emit spikes independently of each other depending on the contrast pattern observed*

Event cameras are inspired by the biological retina. The earliest event cameras created by Misha Mahowald (Mahowald 1992) were inspired by the centre surround principle of ganglion cells that helps to encode the spatial contrast of a scene. Later adaptations of event cameras implemented temporal contrast and temporal difference in a scene for a pixel, which enabled them to function as asynchronous units, similar to the asynchronous receptive field units. The Dynamic Vision Sensor (DVS) is a basic and practically usable class of event cameras in use today that encodes the temporal differences in the illumination of the scene. It is inspired by the 3-layer model of the retina: photo-receptors, bipolar cells and ganglion cells(Figure1.3). Other modern event cameras like DAVIS and ATIS build up on the principle of DVS by complementing it with accumulating static images. (Gallego et al. 2022). In the rest of the thesis, we refer to DVS as the event camera since the work of this thesis employs a DVS camera, and it's one of the basic ones to understand the novel paradigm of sensory data acquisition.

1.2.1 How do event cameras operate?

Event cameras usually have two blocks. The first is the photoreceptor block, made of silicon semiconductors to receive sensory input. The second is a neuron block that receives the changes from the photoreceptor block and encodes it as output information. In an event camera, when a pixel/photoreceptor detects a change in illumination above a threshold, the pixel is activated and sends an *event*. This implies that both changes in the brightness (mathematically the log intensity of a scene) and motion in the scene can cause an event. When a change exceeds the threshold, the neuron layer sends out information in the form of x, y cartesian location, time t and polarity p of the change. p is 1-bit information; $p = 1$ when brightness increases and $p = 0$ when brightness decreases.

[add event camera outputs of a rotating disc]

The sending of information from the photoreceptors of the event camera to the receiving neurons follows an Address Event Representation(AER) protocol.(Posch et al. 2014) In AER protocol, the photoreceptors are the address encoders(AE) and the neurons are the address decoders(AD). AE talks to AD through a digital bus which employs *asynchronous multiplexing*. (Chen et al. 2020) An asynchronous multiplex strategy allocates digital bus 'seats' only to those pixels that have detected a change. If one pixel has nothing to transmit, then that seat can be allotted to another pixel. The AE generates a unique binary address during a change which is transported by the digital bus to the AD, which decodes the binary address as x, y, p, and the time information t is self-encoded by the time of arrival of the information to the AD.

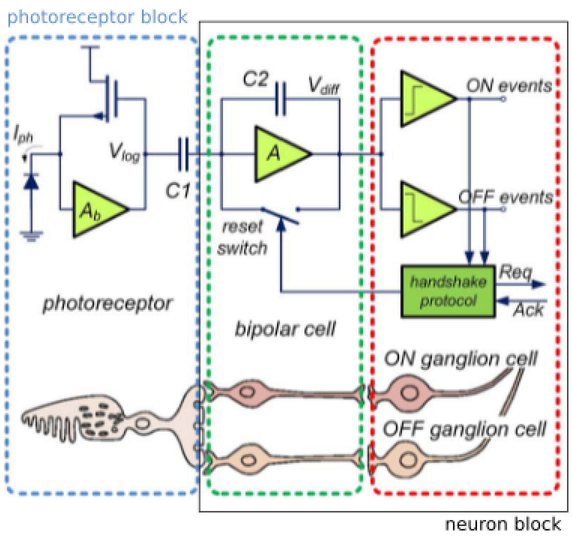


Figure 1.3: Event camera blocks (Posch et al. 2014): *Illustration of biological neuron circuitry along with its corresponding electric circuitry for DVS*

1.2.2 Advantages of event cameras

Standard cameras produce images by capturing pixels that represent the brightness of a scene at fixed time intervals. An image produced at every time interval is called a frame. Hence,frame-based cameras with lower frames per second(fps) have a lower temporal resolution. When recording fast-moving objects with a low fps camera, the low temporal resolution causes motion blur. Each camera pixel is synchronous and is activated only at fixed time intervals depending on the global exposure time of the frame. This leads to high latency for each camera pixel. The shutter speed determines the *dynamic range*(the ratio between an image's brightest and darkest parts) that a frame-based camera can capture. Current-day high-quality frame-based cameras have about 60dB dynamic range. (Gallego et al. 2022)

In contrast, event cameras are a different paradigm of sensing. They are data-driven sensors. The faster the motion, the more events per second are generated since each pixel adapts its sampling rate to the rate of change of brightness at a given x,y location that it monitors. This leads to high temporal resolution avoiding motion blur. Pixels of an event camera record information asynchronously with microsecond resolution and transmit this information with sub-millisecond accuracy. Since event cameras transmit only brightness changes, they remove redundant data and power is only used to process changing pixels. This makes them efficient to be utilised for real-time applications that require recording fast movements and low power consumption. Like biological retinas, event cameras can adapt to very dark as well as very bright stimuli and have a dynamic range of >120 dB. (Lakshmi, Chakraborty, and Thakur 2019)

1.2.3 Challenges due to new paradigm of sensing

One of the significant challenges in neuromorphic algorithm design is creating alternative representations that facilitate extracting low-level features. An *event frame* is one type of temporal compression representation where we create an event image, a boolean 2D grid with the dimensions of the event camera. Events that arrive within a timestep are recorded as a 1 at the (x, y) event locations of the 2D grid. To add the information of polarity to this representation, we can add +1/-1. This gives us a straightforward representation that resembles a traditional camera frame. The advantage of event frame representation is that the time interval can be chosen by the algorithm designer. One event per event frame is also possible if the time interval chosen is to be very small. Event frames differ from traditional frames because traditional frames produce images where the hardware restricts the time interval. Other types of event data representations include *time surfaces*, clubbing events into event packets, motion-compensated event images, etc. (Gallego et al. 2022)

Although, theoretically, event cameras can record and output changes in the order of microseconds, applications cannot use their maximum reactive speed yet. We are limited by the hardware update rates, speed of the processing system, etc. Even interfacing with neuromorphic computers requires a hardware interface that efficiently distributes maximum incoming events for processing.

1.3 Spiking Neural Networks

Traditional artificial neurons used in Machine learning applications are continuous activation functions to transform input signals into output signals. Artificial spiking neurons are more biologically accurate and use a series of discrete pulses to transmit information. Their activity is inspired by the action potential generated by biological neurons. The essential steps of a simple spiking system (artificial or biological neuron) to communicate with neighbouring neurons include

1. Integration - input currents increase the voltage by adding onto the baseline voltage

over time

2. Decay - if there are no inputs for a certain time interval, the voltage of the system decays and the minimum value it can reach is baseline voltage.

3. Spike activation - once enough current inputs have arrived, the voltage of the system exceeds a threshold, and that causes a jump in voltage of the system momentarily. The momentary jump in voltage is called the spike.

4. Reset - the voltage of the system is reset to a reset voltage, usually reset voltage $\leq$ baseline voltage.

5. Refractory period - after a spike, the system does not increase its voltage no matter how much current inputs it receives.

Steps 1 to 5 repeat over time for a spiking system to generate many spikes and communicate with spikes to the neurons they are connected with. The overall general picture stays the same for all biological neurons, but how each step happens varies for different types of neurons. It is regulated by the interplay of different ion channels in a biological neuron, increasing the neuron membrane voltage. There are spiking neuron models that describe the biological mechanisms in detail, some well-established ones being the Hodgkin-Huxley model, Adaptive Integrate and Fire model, Exponential Integrate and Fire model, etc(*Table of contents — Neuronal Dynamics online book* 2023). But, in this thesis, we are interested only in looking at simplified neuron models that capture the five steps superficially. Our goal is to use the 'spiking' aspect of an SNN to be able to encode information in the spatial and temporal dimensions. We want to describe a network topology of spiking neurons with a simplified bio-inspired neuron model as basic units. One of the most basic neuron models is the Integrate and Fire model.

Similar to spiking neurons inspired by action potential digital communication, there are neuron models inspired by graded potentials. The output of neurons varies depending on the strength and duration of the incoming current input. Biological neurons communicate through graded potentials in situations when information needs to be sent to short distances and when fine resolution sensitivity of outputs is required. An example is neuromuscular junctions, where muscles require finer actuation. This can be thought of as analog communication method between neurons. One of the basic neuron models that model Graded Potentials is Leaky Integrate model.

Leaky Integrate-and-Fire model

Leaky integrate-and-fire(LIF) neuron model was first proposed by Lapicque, to represent a neuron as an electric RC circuit. The diffusion of ions is captured by the decay term, the membrane time constant. *Membrane time constant* can be defined as the time taken for the voltage to become 0.63 of its starting value. LIF model assumes a 'point neuron', where the details of the dendritic structure of the neuron are ignored, and all inputs are effectively applied direct to the soma. The dynamics of a spiking neuron can be captured by the following equations:

$$\frac{\mathrm{d}V}{\mathrm{d}t} = (1/\tau_m)(V_{rest} - V + I) \tag{1.1}$$

$$z = \Theta(V - V_{th}) \tag{1.2}$$

where V_{rest} is the resting membrane potential(baseline voltage), V_{th} is the voltage threshold, I is the input current, and τ_m is the membrane time constant of the neuron.

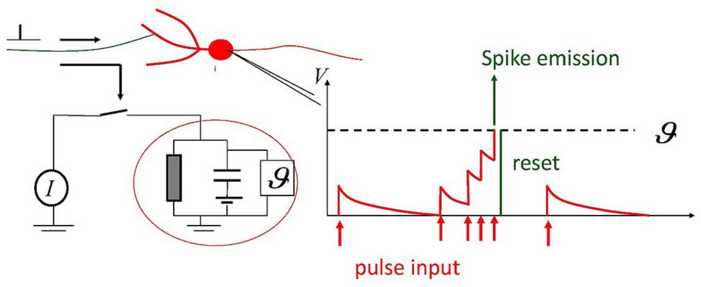


Figure 1.4: LIF neuron dynamics

Leaky Integrate model

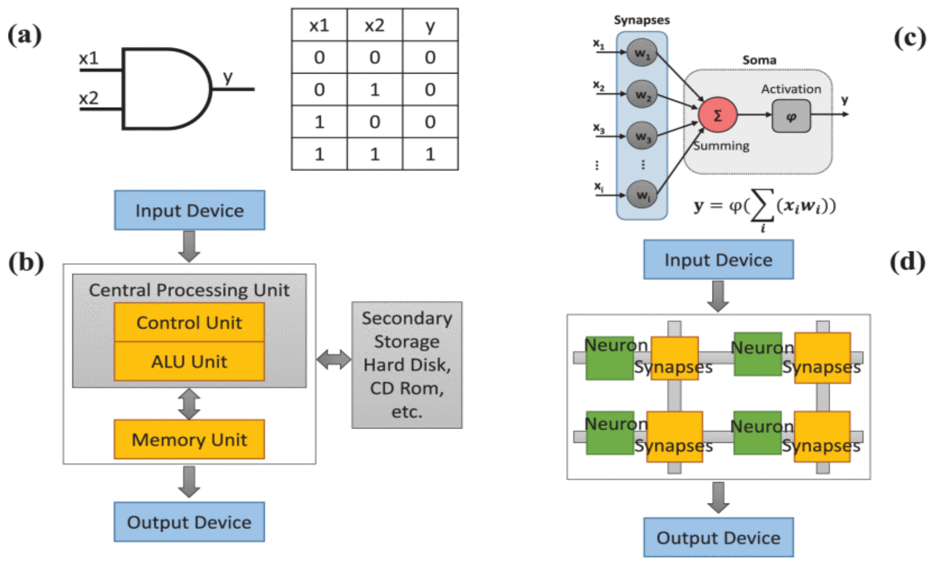
Leaky integrators describe a leaky neuron membrane that integrates incoming currents over time but never spikes. In other words, the neuron adds up incoming input current while leaking out some of it in every timestep. The dynamics of the model are same as the LIF but without thresholding.

$$\frac{\mathrm{d}V}{\mathrm{d}t} = (1/\tau_m)(V_{rest} - V + I) \tag{1.3}$$

1.4 Accelerating Hardware

A biological brain is built of interconnected layers and groups of neurons. Each biological neuron is a simple processing unit that relies on non-deterministic operations and decentralised computation. Each neuron uses action potential or graded potential to talk to each other. There is no central authority that controls the activity of all neurons. Instead, it's the orchestra of networks of neurons that leads to processing. Dynamic synaptic strength and network connectivity provide biological systems versatility in performing tasks.(Nawrocki, Voyles, and Shaheen 2016) In contrast, a traditional von Neumann architecture computer separates the memory storage unit and the information processing unit. The separation leads to the von Neumann bottleneck, which increases computational time. They use boolean logic for computation, and their architecture is static.

For accelerating tasks in the neuromorphic paradigm, we look for alternative hardware architectures inspired by the asynchronous parallel processing of the brain. Analysing all hardware alternatives is out of the scope of this thesis. We consider only two types(among others) that enable us to implement Spiking Neural Networks efficiently.



x1	x2	y
0	0	0
0	1	0
1	0	0
1	1	1

Figure 1.5: von Neumann computers vs biological brain architecture(Nawrocki, Voyles, and Shaheen 2016)

1.4.1 SpiNNaker neuromorphic hardware

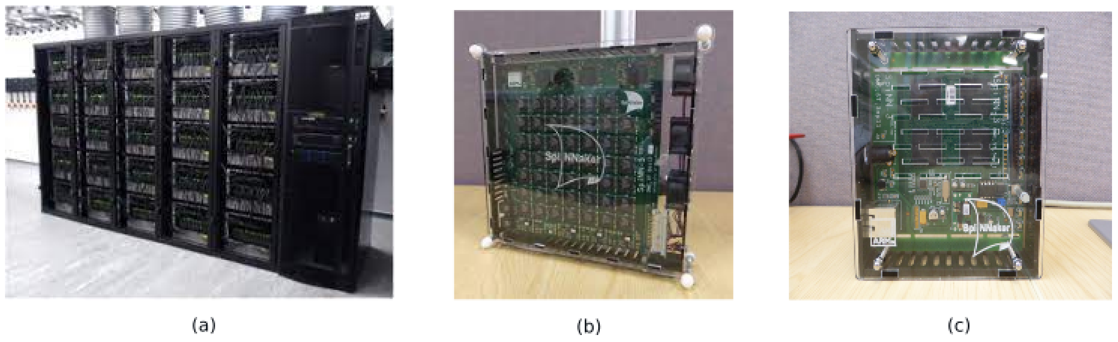


Figure 1.6: SpiNNaker chips and hardware(*Research Groups: APT - Advanced Processor Technologies (School of Computer Science - The University of Manchester)* 2023): (a) Large machine used for whole brain simulations (b) 48-chip board (c) 4-chip board

Architecture: SpiNNaker(Spiking Neural Network Architecture) is a high-performance computing system designed to simulate the collective behaviour of billions of neurons in real time. It consists of a custom chip with one million cores, each optimized for low-power consumption and high-speed communication. The chip is connected to a network of routers that enable communication between the cores. The system uses a multicast communication protocol that allows the same message to be sent to multiple destinations simultaneously, enabling efficient communication between neurons in the network. The architecture also includes specialized hardware for managing the timing of neuronal events, which is critical for simulating the behaviour of spiking neural networks. SpiNNaker was developed at the University of Manchester to simulate the cortex of a mouse using a million cores. But, the research team also built smaller versions that contained

4 chips and 48 chips. The 4-chip board can simulate up to 10^4 neurons, whereas the 48-core machine can simulate up to 10^5 neurons. These smaller development boards were created to deploy Spiking Neural Networks on robots and on-edge applications (Furber et al. 2013).

1.4.2 Graphical Processing Units (GPUs)

Architecture: The Graphical Processing Unit (GPU) is a massively parallel architecture that can be used for neuromorphic algorithms with superficial biological realism. It has a large number of small computing cores, and when grouped together, it is called processing units or streaming multiprocessors. This architecture allows for Single Instruction, Multiple Data (SIMD) threads. SIMD threads help to break down a single instruction (e.g. multiplication of 2 numbers, n, m) into repetitive tasks (e.g. adding n m times) and assign them to multiple computing cores within a processing unit. A processing unit contains its own local cache(temporary memory location) but also has access to a shared cache of the GPU. Data that is frequently used is stored in the local cache, while less frequently used data is stored in the shared cache. This allows processing cores to access data quickly.(Ivanov et al. 2022)

How can GPUs be advantageous to simulate SNNs?

The computations required for simulating the behaviour of many neurons in parallel are distributed across the available GPU cores. Each neuron may require multiple computations per time step, such as calculating membrane potential, spike generation, and synaptic transmission. These computations can be distributed across multiple GPU cores. To achieve high performance when simulating SNNs on GPUs, a network's neurons may be grouped into batches or mini-batches, with each batch assigned to a different GPU core. Alternatively, the computations required for each neuron may be further subdivided into smaller tasks, which can be assigned to different GPU cores for parallel execution. Overall, we want to maximise the parallel processing capability.

Unlike a neuromorphic architecture SpiNNaker, GPUs are general-purpose processors that can be used for a wide range of computing tasks. GPU processing cores are optimized for matrix operations and are connected through a high-speed memory bus. Whereas SpiNNaker's architecture is optimized for performing the computations required for simulating individual neurons, and the network allows for efficient communication and synchronization between the different processing cores. There are tasks where SpiNNaker and GPUs would show similar parallel computing performance nevertheless.

The main advantage of using GPUs for simulating SNNs is their ease of programmability and maturity of the framework relative to other neuromorphic architectures. Several software frameworks have been developed for simulating SNNs on GPUs, including ones that can use the popular PyTorch framework also for learning tasks. These frameworks provide tools for modelling and simulating SNNs. GPUs allow experimenting with different SNN architectures easily.

Chapter 2

Line Tracking

Line tracking is a broad field of work with applications ranging from UAV surveillance(Li et al. 2008) to medical image analysis(Czerwinski, Jones, and O'Brien 1999). In this chapter, we focus on tracking infinitely long lines. We can roughly classify line-tracking algorithms into detect-and-track methods and motion-based methods. Detect-and-track methods estimate line position and orientation in each frame and update the line estimate, leading to tracking. Two popular line detection algorithms are Hough Transform and Canny line detection algorithms. Motion-based methods detect a line based on the apparent motion between two frames. An example will be Optical flow-based line tracking algorithms inspired by biological optical flow navigation of insects.

Also, machine learning algorithms can be used in both methods to learn the representations of lines. These are much more popular nowadays for traditional line-tracking applications. But, in the scope of this thesis, we are interested in adapting feature-based geometric methods to the neuromorphic paradigm. Primarily because learning in Spiking Neural Networks is trickier than in traditional Neural Networks(NNs). Feature-based methods also allow us to know the precise representation used for detecting lines, whereas NNs tend to be 'black boxes'. We will have an algorithm that works out of the box without any training, which is an essential component for a pre-processing step.

We choose to adapt Hough Transforms to the neuromorphic paradigm to detect lines. We use the detect-and-track method for line-tracking. In the following sections, we provide the rationale for choosing Hough transforms and why they are a good candidate for a spiking line tracking algorithm. Also, many works with traditional hough transforms have extended it to detect line segments. Our goal is also to propose a neuromorphic line segment detection algorithm.

2.1 Standard Hough Transform (SHT)

Consider an input image size of n_x^{in}xn_y^{in}. Let's suppose the goal is to find line L from the input image, as shown in Figure 2.1(a), which resides in $x - y$ cartesian space. The slope-intercept form of a line is by the equation:

$$y = mx + c \tag{2.1}$$

where m is the slope of the line and c is the intercept of the line from the origin. So, two parameters, i.e., m and c, are sufficient to know all the information about where the line lies in the cartesian space. Now, consider a point $p_1(x_1, y_1)$ on L in image space as shown in Figure2.1. We can ask, "What are all the possible lines passing through p_1?" If we exhaustively review each of these possibilities, we will find that L will obviously be one of them. So, if we create a space parameterised with m and c, called the *Hough space*, we can now represent each one of those possible lines passing through p_1 as a point. All the possible lines passing through the point,i.e. all the possible values of m and c passing through p_1, form a line in the Hough space as evident from rearranging Equation 2.1 where m and c are the variables describing a line in the Hough space: $c = (-x)m + y$

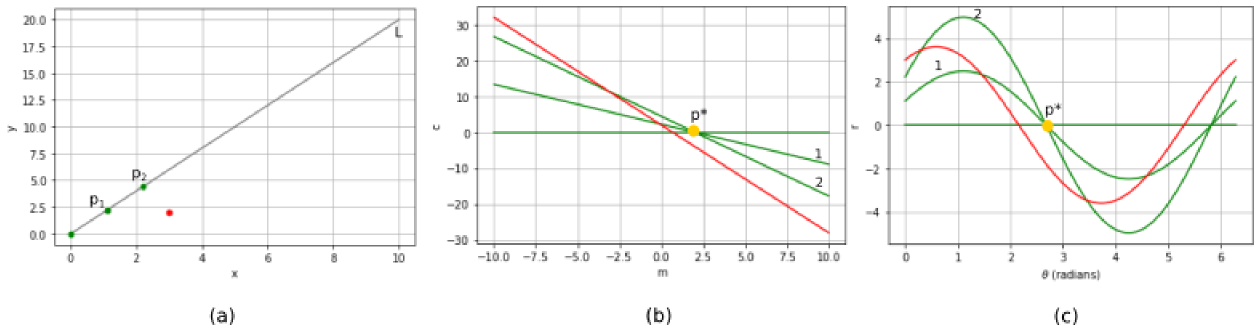


Figure 2.1: Hough transforms

If we map a second point from L, $p_2(x_2, y_2)$ to the Hough space, we see that there is an intersection of two Hough space lines generated by the two points. This intersection suggests that there exists one line that passes through both points. Similarly, we can do this for every point on L. The point in Hough space with the maximum intersection(p^*) of Hough lines gives us the parameters m_l, c_l sufficient to describe L. The line detection problem in the cartesian space reduces to a peak detection problem in the Hough space. So, every point from the cartesian space is a line in the Hough space, and every point in the Hough space is a line in the cartesian space.

A vertical line will correspond to a line whose slope is unbounded since m and c have the range (-∞,∞). This method cannot map all sets of collinear points in the image space within a bounded region defining the Hough space. (Mukhopadhyay and Chaudhuri 2015)

So, we can choose other parameters to define a line: the perpendicular distance of the line from the origin r and the angle that the perpendicular to the line makes with the origin θ. This way, θ ranges $[0,2\pi)$ and r ranges [0, R] where R is the distance of the point on the line that's farthest away. Now we can map each point in the cartesian space to the Hough space parameterised with r and θ using the equation below:

$$r = xcos\theta + ysin\theta \tag{2.2}$$

So, by doing this mapping, we are essentially asking, "What are all the possible curves described by the equation that go through p_1?". So, each point generates a Hough curve in the Hough space. The value of r, θ at the intersection of these curves, r_l,θ_l, are the parameters required to describe L. Here, every point in the cartesian space is a curve in the Hough space, and every point in the Hough space is a line in the cartesian space.

This method was first described in the 1970s by Duda and Hart(Duda and Hart 1972) and is a popular line extractor for pre-processing images in modern Computer Vision libraries and tasks. To traditionally implement this idea in an algorithm as described by Duda and Hart, we discretise the Hough space to be represented by an accumulator array. We increment the accumulator array by 1 along the curve described by Equation2.2 for every point in the cartesian space. This process is called *voting* by each point. We search for the coordinates of the cell with the highest value. That gives us the parameters r_l,θ_l of our desired line L.

Hough Transform Extensions

One of the most straightforward cases of Hough Transform(HT) is the SHT which has been used as a robust line detection method. Many further extensions have been done to overcome the drawbacks of SHT. A class of HTs called Probability-based Hough Transform have been proposed to reduce the computational time. If the total line points are M, only m $\subset$ M values are considered for voting. Long lines only require a small fraction of their supporting points before their corresponding accumulator array reaches a non-accidental value. SHT has the voting phase time complexity as O(MxN_θ), which reduces to O(mxN_θ) for Probability-based Hough Transforms. (Mukhopadhyay and Chaudhuri 2015)

Hough Transform can be extended to detect other shapes and parametric curves. The requirement is to know the equation/parametrisation of the shape-of-interest. We then ask the question for every point in the image space, "What are all the possible shape-of-interest that passes through a particular point?" We increment the accumulator array based on the parametric equation and then find the peak in the Hough space. The accumulator array scales according to the minimum number of parameters necessary to identify the shape of interest. For example, a circle requires the knowledge of 3 parameters, hence a 3D Hough space. The Generalised Hough Transform (GHT) was proposed to be able to identify curves and shapes whose parameters are unknown. (See Appendix A1)

Advantages

SHT is robust to noise. Noisy points that deviate from the actual line do not contribute coherently to the accumulator array peak. It is also easy to extend single line detection to multiple-line detection. Instead of finding one peak, if we define a threshold value above which we can convince ourselves about the existence of a line. The accumulator array values that exceed the threshold will be considered as multiple peaks and hence multiple lines. Note that the choice of the threshold will affect the line detection results. A computationally convenient advantage is that this method can be implemented parallelly since each image point is considered independently from one another.(Illingworth and Kittler 1988)

Drawbacks

One of the major drawbacks is the algorithm's storage and computational complexity. The memory complexity scales O(N_rxN_θ), and the computational cost for the voting phase is O(MxN_θ), and for searching the peak is O(N_θx$log(N_r)$), where N_θ is the quantisation of theta, N_r is the quantisation of the r, and M is the total number of points(line supporting points + noise). N_θxN_r is the size of the accumulator array. This can be inefficient for real-time line tracking, where we deal with sequences of images that form a video, and the computation time should be less than 1 ms for us to observe the line tracking to be smooth. Also, parameter quantisation affects the results of line detection. If the quantisation is too coarse, then the detection accuracy is bad; if the quantisation is too fine, then the possibility of detecting multiple wrong lines is high.

2.2 Spiking Line detection

Let's consider the cartesian image space to be a sheet of 2D LIF neurons called the *Camera Neurons*. It has the same dimension as the event camera space. We create another 2D sheet of neurons corresponding to the Hough space called the *Hough Neurons*. Each Camera Neuron is connected to the Hough neurons using synapses. This synaptic connectivity follows the SHT mapping as given by Equation2.2. So, we have a single Camera Neuron connected to a curve of Hough Neurons. Whenever an event with event coordinates (x_i, y_i) occurs, the corresponding neuron in the 2D Camera Neurons spike. Hence, the Hough Neurons connected to neuron (x_i, y_i) increase their membrane potential. Once a Hough Neuron gets multiple such inputs from a single line, its membrane potential increases above the voltage threshold to be able to record a spike. If that makes a Hough Neuron spike, it gives us evidence that the Camera Neurons have received enough events within a time interval corresponding to a line. The coordinates of the Hough Neuron that spikes give us the line parameters (r_l, θ_l).

If the Camera neurons spike corresponding to multiple line input stimulus, it causes many Hough Neurons to spike. In the ideal case, the number of Hough Neurons that spike corresponds to the number of lines present in the stimulus.

Advantages of SNN adaptation:

When detecting lines in a real-time setup with conventional cameras and SHT, we calculate an accumulator array for every frame. This is extremely expensive computationally, as mentioned before. The SNN adaptation ensures that we calculate a weight matrix

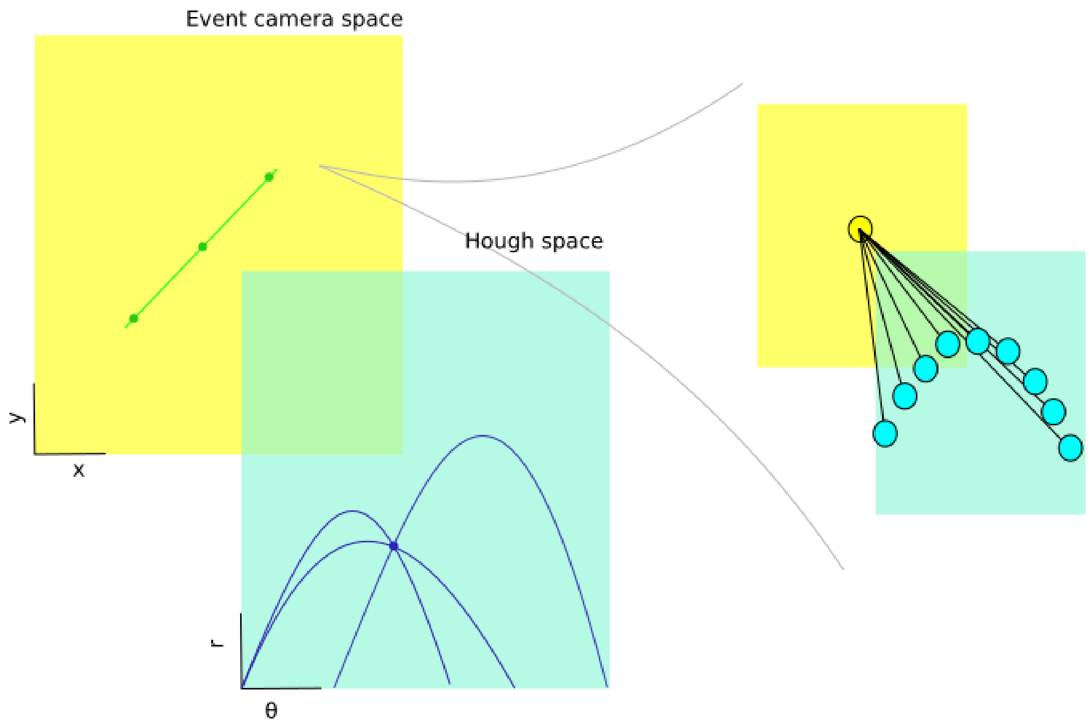


Figure 2.2: SNN Connectivity

that specifies how each Camera Neuron is connected to the Hough Neurons. The weight matrix computation is performed only once for a given value of input camera dimensions. Each neuron performs computation independent of another, so the Spiking Neurons parallelise the time taken for the voting phase in the traditional SHT. The peak detection problem reduces to tracking neurons that spike.

Challenges:

Ideally, we want the SNN-HT to work for any input camera dimension. If we assume there are n Camera Neurons, scaling up Hough Neurons will require atleast n^2 Camera Neurons. This will ensure that the discretisation of Hough space can capture approximately all possible values of (r, θ). But, it requires high power consumption due to the increased number of computing Hough Neurons when high-resolution event cameras are used. To run the algorithm on-edge, we require low power consumption, and one way to do that will be to fix the number of Hough Neurons. It will lead to cases where Hough Neurons are less than Camera Neurons. It leads to a constrained hough space coarsely discretised such that an accurate estimate of the line becomes difficult. It is because the (r, θ) corresponding to the line is far away from any position of the neuron positions' discretisation.

2.2.1 Population Coding

We take inspiration from population coding in biological neurons to represent the finer resolution of values in hough space. Neurons generally employ population coding when individual neurons have limited information-carrying capacity.The combined activity of many neurons can convey complex information. In the medial temporal (MT) visual area of the brain, neurons are specialized to respond to the direction of movement.(Blakemore and Tobin 1972) When an object moves in a specific direction, a large number of neurons in MT become active with a pattern of activity that is noisy and bell-shaped across the population. By considering the activity across the population, the brain can extract information about the direction of the moving object and reduce the impact of noise present in any individual neuron's activity. An entire population ensures greater fidelity and precision. In this thesis, we employ two types of population coding:

Sparse coding

Each activated neuron generates only a few action potentials (temporal sparseness) for a highly specific stimulus configuration. Only a small fraction of all the neurons available for encoding a spike are used. (population sparseness). This type of population coding supports the *efficient coding* hypothesis. The efficient coding hypothesis states that the visual processing system has an efficient strategy for transmitting as much information as possible due to the constraints of reducing the dimensionality of information as it passes from the retina to the visual cortex. Sparse coding also reduces the overlap between different representations for a given input stimulus. (Willmore, Mazer, and Gallant 2011)

Position coding

Position coding is a type of population encoding when each neuron has a Gaussian tuning curve of activity,i.e. neurons respond strongly to a stimulus near the mean of the Gaussian. When each neuron has multiple firing fields, i.e. can spike due to different stimuli differently, it allows encoding exponential amounts of information with a polynomial increase in inputs.

We use the ideas of position and sparse coding to solve the problem of a constrained Hough Space.

2.2.2 SNN adaptation with population code

As described previously, there are two layers of 2D sheets of LIF neurons, the Camera and Hough Neurons. Instead of using all the neurons in the Hough space to enable population coding, a subset of neurons that trace the perfect curve described by Equation 2.2 is used. Each Camera Neuron is connected to Hough Neurons that trace the perfect curve - *Hough band*. Hough Neurons closer to the perfect curve described by Equation 2.2 have higher synaptic weights than neurons far away from the perfect curve. The synaptic strengths across the hough band follow a Gaussian spread out in the r dimension of the hough space.

Multiple spikes due to a single line cause multiple hough bands to increase their membrane potential. Instead of having one neuron that spikes, there are multiple neurons that spike at different times at the intersection of hough bands. Since the build-up of activity in the hough band of neurons takes time, the spikes transmitted are *sparse.* A third layer of LI neurons called *Readout Neurons* is introduced to capture the temporal information of these spikes. LI neurons communicate with graded potential. Graded potentials have higher information rates capable of encoding more states, i.e. higher fidelity, than spiking neurons.

The shape and number of the Readout Neurons and Hough Neurons are the same. Hough Neurons are connected to the Readout Neurons with one-to-one connectivity. The synaptic weights are uniform and fixed. Whenever a Hough Neuron spikes, it increases the membrane potential of a single Readout Neuron that it is connected to. Then, the membrane potential of the readout neuron decays with time until the next spike arrives. LI Neurons do not spike, so when multiple Hough Neurons that lie at the intersection of *Hough bands* spike, a blob of activity builds up in the Readout layer.

When there are multiple line input stimuli, it creates multiple Hough band intersections and hence leads to multiple blobs of activity in the Readout neurons. The number of blobs corresponds to the number of lines in the input stimuli. Clusters of neurons can be found to identify blobs of activity in the Readout layer.

A finer resolution position estimate is available if all the neurons in the blob are considered for calculating the (r, θ) position of a single line. Weighted averaging of the positions of Hough neurons gives us a better estimate of the actual position of (r, θ). The weighted averaging for a single blob j is mathematically given by:

$$(r, \theta)_j = \frac{1}{k} \sum_{i=0}^{k} a_i (r_i, \theta_i)_j \tag{2.3}$$

where k is the number of neurons in blob j. a_i is the magnitude of the membrane potential of the Readout Neuron i in blob j.

2.3 Methods

2.3.1 Weight matrix of line detection

For every event camera pixel (x_i, y_j) where i=0,1,2,...,n_x^{in} and j=0,1,2,...,n_y^{in}, we calculate the values of r for n_θ^{out} discretisations of θ given by equation 2.2. r values range from 0 to diagonal length of the event input space, $\sqrt{(n_x^{in})^2 + (n_y^{in})^2}$, that are discretised into n_r^{out} units. This is because the furthest perpendicular distance of a line from the origin cannot be more than the diagonal. θ ranges from 0 to 2π so that the curve $x \cos\theta + y \sin\theta$ 2.2 has a wrap around in the θ dimension. While many traditional versions of Hough transforms extend the r in the negative direction and restrict the θ from 0 to π to cover all line instances. It ensures that no double peak occurs due to 2 intersection points when theta

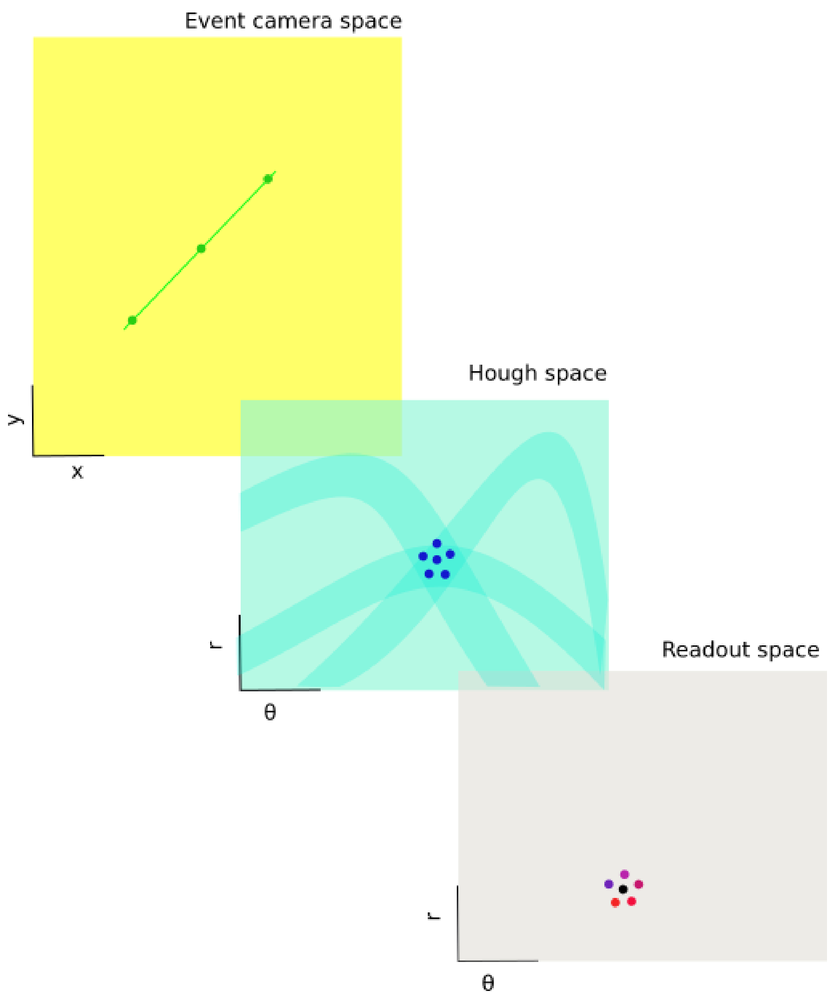


Figure 2.3: SNN Population Coding connectivity

extends upto 2π. This is valid when we are assured that there exists one neuron that has the exact (r, θ) coordinates of desired line. In our case, we use population coding, which requires that we find neuron clusters of activity. So, having the wrap-around in the θ dimension helps the clustering algorithm. refer **??**.

We maintain a list of r range that has n_r^{out} values.After finding the perfect value of r r_k using 2.2, we find the distance between all values of r discretised by n_r^{out}, to the perfect value r_k. We define a threshold of neurons that constitutes the band. In our implementation, we choose it to be $4(diagonal/n_r^{out})$. So, if $|r - r_k| <$ bandwidth, then we only distribute activity to neurons in those positions r_{pos}. The activity is distributed according to a Gaussian distribution of distances of neurons (r_{dists}) within the bandwidth. The Gaussian distribution is centred at 0, and the variance is 1, given by the function $exp(-x^2)$. The maximum value of the weight matrix can reach up to 1 if $|r - r_k| = 0$.

The weight matrix is computed and stored as a 4-dimensional array. The first two dimensions hold values of the event camera coordinate. The other two dimensions hold values of Hough space discretisations. When supplied to an SNN, the weight matrix is flattened to a $(n_x^{in}.n_y^{in})$ x $(n_\theta^{out}.n_r^{out})$ size 2D weight matrix.

Algorithm 1 Generating 2D weight matrix

```
1:  $\theta$=[0,2$\pi$) r=[0,diagonal]
2:  for each event camera pixel $(x_i, y_j)$ do
3:      $r \leftarrow x_i \cos\theta + y_j \sin\theta$                    ▷ curve equation 2.2
4:      for each point on the curve $(\theta_k, r_k)$ do
5:          $\theta_{pos} \leftarrow$ position of $\theta_k$
6:          $r_{pos} \leftarrow$ positions of r within bandwidth of $r_k$
7:          $r_{dists} \leftarrow$ distance of r within bandwidth of $r_k$
8:          weightMatrix[x$_i$, $y_j$,$\theta_{pos}$,$r_{pos}$] $\leftarrow$ $gaussian$(r$_{dists}$)
9:      end for
10: end for=0
```

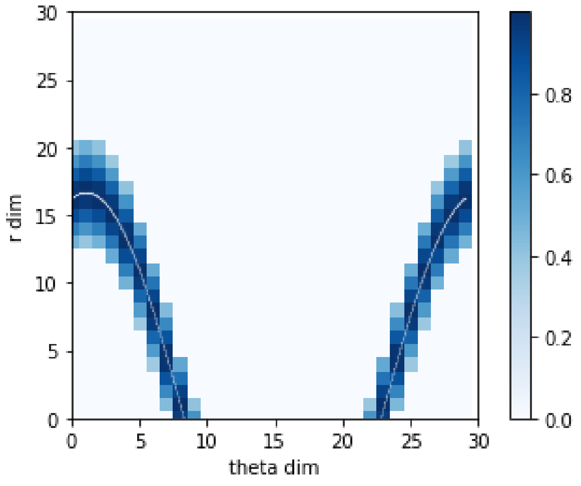


Figure 2.4: Weights corresponding to event (250,90)

2D weight matrix check: We check the results of weight matrix construction by sampling the weights with Hough space dimensions (30,30) that correspond to (250,90) as an example. In Figure 2.4, the white line represents the exact curve generated by a scaled-down version of Equation2.2. We scale down the r dimension by $\frac{n_r^{out}}{diagonal}$ and θ dimension by $\frac{n_\theta^{out}}{2\pi}$ since the hough space is smaller than the event camera space. We can notice that the thickness of the band is not uniform throughout but becomes thinner when the slope of the curve is high. This is because we calculate the r positions of neurons to activate by assuming that θ fluctuations contribute significantly less to the spread out. It is safe to make this assumption since the magnitude of r is two orders of magnitude greater than magnitudes of θ. The small fluctuations of theta do not impact the finding of the r position of the neuron.

2.3.2 Spiking Neural Network implementations

We use the GPU implementation for parameter tuning and to evaluate the robustness of our algorithm. We port the same model to neuromorphic hardware and evaluate line detection results. The appendix contains snippets of code implementation.

GPU implementation

Norse is a SNN library that extends PyTorch with primitives for bio-inspired neural components. It has neuron models that are compatible with being run on GPUs. It's also compatible with deep learning SNN components. We want to use Norse's ability to integrate seamlessly with PyTorch to build our SNN model. PyTorch computation requires that we bunch together events as "frames" and feed them to the SNN layer. More details about creating frames are provided in Data collection.

We use the LIFBox model in Norse that rightly describes the LIF neuron dynamics that we require (as described in Theory) for the LIF layer. We use the LI neuron model from Norse for LI layer. Both neuron models share the same parameters; only there is no threshold voltage to spike V_{th} for LI neurons. Instead, LI neurons act like capacitors.

The neuron parameters to tune in the LIFBox model are V_{th} threshold voltage to spike, τ_{mem}^{-1} inverse of the membrane time constant. All other parameters are kept as default. Increasing V_{th} requires more events to lead to a spike. Increasing τ_{mem} membrane time constant controls how quickly the generated spikes decay. If τ_{mem} is too high or V_{th} is too low, then instead of a blob of activity that we require for identifying lines, we see a region of activity in the LI layer. If τ_{mem} is too low, the LI readout layer gives inconsistent outputs. If V_{th} is high, there are no spikes to contribute to the Readout layer. We perform a brute force search starting from default values and find that $V_{th} = 3$ and $\tau_{mem} = 0.001$ are the values that contribute to blobs of activity as we desire. We use a weight scaling factor for the weight matrix $weight\_scaling\_factor = 0.01$ so that the maximum value of the weight matrix becomes 0.01. This helps to reduce the intensity of the connectivity matrix. It can instead also be achieved by carefully tuning 2 parameters previously mentioned, τ_{mem} and V_{th}.

We tune the parameters by fixing 500 events per frame batch with a Hough layer size of (30x30). We normalise the inputs to the SNN because real event cameras do not receive fixed batch sizes of 500 events. Using unnormalised inputs will oversaturate neurons in the Hough layer. We use the normalising factor $= \frac{500}{no. of events in frame}$

We implement the algorithm in inference mode. We define an initial PyTorch linear layer comprising of $(n_x^{in} \text{x} n_y^{in})$ neurons. We initialise the weights of the linear layer with the weight matrix. The linear layer is connected to the LIFBox layer using all-to-all connections whose weights are modified by the weight matrix. The LI layer is one-to-one connected to the LIFBox layer.

Neuromorphic hardware implementation

We stream real-time data from event cameras and check the output and performance on SpiNNaker with 48 chips(spin-5). We can use an ethernet cable to stream events to SpiNNaker, but it has a high latency during streaming. The latency is due to an intermediate computer that acts as a ”portal” which sends correct address mapping to SpiNNaker cores. SPIF - SpiNNaker peripheral interface is an FPGA board design created as a collaboration between KTH Royal Institute of Technology and the University of Manchester [add citation]. SPIF acts as a bridge to bypass the portal and directly route event-camera inputs to the SpiNNaker cores. It allows streaming high throughput data, i.e streaming $> 500,000$ events/s, with low latency of <1ms. Using SPIF ensures we acquire low-latency event camera data, which can then be processed by our Spiking line detection algorithm. The research development on SPIF is still an ongoing effort that has shown exceptional results.

We use PyNN, a simulator-independent language for building SNNs in Python. It is intended not to have to use different syntax to build the same SNN on different neural simulators. Currently, it supports neural simulators NEURON, NEST and Brain without making any modifications. sPyNNaker package extends PyNN and provides classes and functions to program SpiNNaker using Python, a high-level programming language.

To test our algorithm written with sPyNNaker for real-time detection of lines, we interface the event camera using `SPIFInputDevice` method of external device class. This method allows SPIF to be recognised as an external device connected with SpiNNaker, and we can connect remotely to SPIF to stream event camera inputs from anywhere. SpyNNaker considers SPIFInputDevice as a population of neurons so that it can be easily connected with other populations. SPIFInputDevice ”neurons” do not live on SpiNNaker,i.e. take up space and memory. It’s just a python representation to route events to the actually defined population (LIF neurons in our case) in SpiNNaker.

It’s intuitive to define neuron layers in PyNN/sPyNNaker. To create a neuron layer, we define a `Population` of neurons and specify the spatial structure of the population. We require a 2D spatial structure for the LIF and LI neuron layers. We use the same LI and LIF neuron models as described in the Norse implementation. All the parameters found from the Norse implementation are ported here. We use the `FromListConnector` function to store the values of connections and their corresponding weights from the 2D weight matrix. Then we connect the input population(which is SPIFInputDevice neurons), LIF layer, and LI layer using the `Projection` function.

Each SpiNNaker core has a limited number of neurons that it can process to provide output, so we need to assign the number of neurons per core (npc) through software to SpiNNaker. Determining the npc varies from application to application. If we aggregate more neurons in a core, then we risk the possibility of oversaturating the core when there are too many incoming events for every neuron in that core. An analogy will be when a traditional computer hangs because of too many threads it is running. Similarly, when a core, with its own little memory and processing, receives too many spikes to route to the neurons it contains, it gets stuck. This can be addressed by spreading the neurons into multiple SpiNNaker cores. But, the communication latency between cores is higher than within a core. Also, it uses more power. By trial and error, we set $npc = 10$.

After routing events to SpiNNaker, `SPIFLiveSpikeConnection` method of external devices class receives live spikes back from SpiNNaker. We define a readout population of neurons `SPIFOutputDevice`, again as a virtual population of neurons that do not take up space on SpiNNaker but help to stream out live spikes from SpiNNaker. We activate the `activate_live_output_to` method from the external devices class to enable reading out.

2.3.3 Clustering

Once we have the activity of all neurons from the LI layer, either from the GPU or neuromorphic implementation, we perform CPU clustering. The LI layer output is a 2D array that might be considered a 2D sheet. In reality, the LI layer output lives in a cylindrical space since θ has a wrap-around such that 0 and 2π are the same. Figure 2.5(a) shows the Readout layer activity at one timestep for two near horizontal lines captured from a DVS camera. We notice the requirement of a cylindrical clusterical space as the blobs of activity spread at the ends of the Readout layer.

So, to find neurons that are close to each other and group them together as clusters, we need to perform clustering in the cylindrical space. To do so, we can define a transformation function

$$f_t : (\theta, r) \rightarrow (\sin\theta, cos\theta, r)$$

The resultant cylindrical space has a width of 1, and the height of the cylindrical space is limited by the bounds of $r \in [0, diagonal]$. The choice of width of the cylindrical space is irrelevant. But, r is two orders of magnitude greater than θ. So, we require to normalise the values of r comparable to θ magnitude order so that the clustering algorithm is not preferential towards θ. We normalise by dividing r values by order of magnitude 10^2. Figure 2.5(b) shows the correct two clusters when clustered in the cylindrical space.

We implement the KMeans algorithm from the sklearn python library on the transformed cylindrical vector space. Initially, we provide KMeans with the number of clusters we want for test cases with artificial data. We perform a weighted clustering based on the activity of the neurons. This can be easily done by setting the parameter `weights = LIOutput` of the `fit_predict` method of the KMeans clustering algorithm from sklearn. The resultant clustering output assigns cluster numbers to each point.

We consider the activity of neurons that are greater than 0.5 times the maximum spiking neuron for each cluster. This is to minimize *hough band interferences*, i.e. the overlap of activity between hough bands that correspond to different lines. For each cluster, we perform a weighted average based on the individual activity of the neurons. This provides us with $(average(\sin\theta), average(cos\theta), average(r))$. Then we need to find the inverse transform to find the parameters of the desired line.

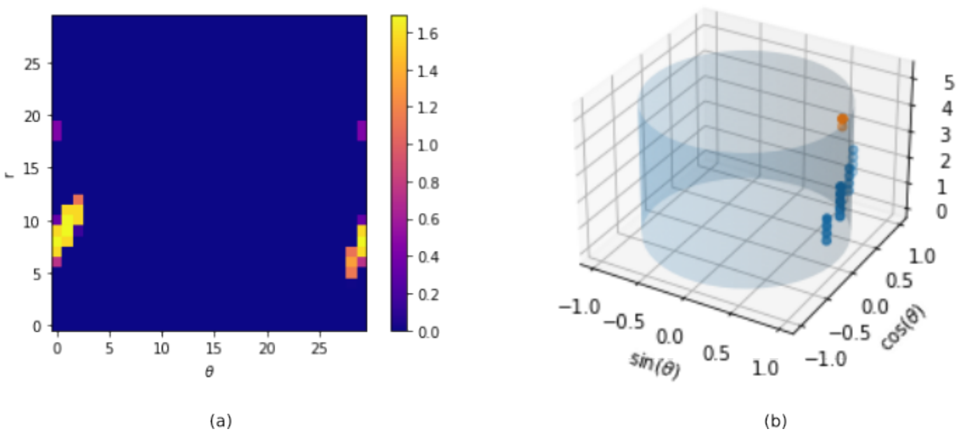


Figure 2.5: Clustering: (a) Readout Layer activity of near horizontal lines (b) Cluster mapping with f_t to cylindrical space

$$f_t^{-1} : (sin\theta, cos\theta, r) \rightarrow (\tan^{-1}(\frac{sin\theta}{\cos\theta}), r)$$

2.3.4 Data collection

Three types of stimuli are provided: The quality of the line detection algorithm is checked using recorded event camera data. To enable fidelity in analysis, artificial data is used for quantitative analysis. Real event-camera streaming is performed to further evaluate the performance of a real-world task like tracking a pendulum or tracking a template.

The event camera in use is a Dynamic Vision Sensor camera by IniVation with 640x480 dimensions. DV-GUI is used to collect event data recordings. A noise filter is attached as a part of the event recording pipeline to eliminate oversaturated event spikes that leads to high noise content. AEStream is used for streaming events live. AEStream is a python package that can efficiently read event camera inputs and stream them with low latency (Pedersen and Conradt 2022). The package supports multiple event camera input types and also can be used for recording events. The advantage that is lucrative for us to use AEStream here is that it also provides readymade PyTorch tensor frame outputs with the time interval of grouping together that we require. That is desirable to work with GPU experimentations of our algorithm.

Chapter 3

Line Segment Tracking

Line segments provide more concise information about the confines of an object/shape than infinitely long lines. Shapes form building blocks of much more geometrically complex objects. To identify line segments, we need to know the start and end points of the segment within an infinitely long line. In this chapter, we focus on finding the constraints that enable line segment detection. We take inspiration from the line segment detection method as proposed in (Bachiller-Burgos, Manso, and Bustos 2017)(Seifozzakerini, Yau, and Mao 2017) and extend the method of adaptation to the neuromorphic paradigm discussed in the previous chapter. This chapter only discusses the idea and method of execution, but further experiments need to be performed to check the feasibility of line segment detection running real time.

3.1 Rethinking Hough mapping

A line described by (r, θ) is mathematically an infinitely long line. Going a step further, we ask, what are the start and the end points of a line segment that lives on an infinitely long line? One of the obvious ways to do that will be to map the line estimate from the 2D Hough space back to the cartesian space and check whether each event belongs to the line or not. But, this is computationally expensive. So, we need to think of ways to tackle this problem by mapping events from the cartesian space to the Hough space such that they can be easily retrieved as extrema points of a line segment.

Consider two points $e_i = (x_i, y_i)$ and $e_j = (x_j, y_j)$ in cartesian image space, which are the extrema line segment points. Assume we already know the exact line $\mathrm{L}(r_l, \theta_l)$ that passes through the line segment. We can consider a local coordinate system wrt to the line as a new coordinate system to locate points. This new coordinate system is one where the Cartesian coordinate system is rotated by θ_l for line L. Similarly, for any given line (θ, r), we can rotate cartesian coordinate system by θ. The new coordinate system is represented in relation to the cartesian coordinate system as follows:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} y \\ x \end{bmatrix} \tag{3.1}$$

$$a' = Ra$$

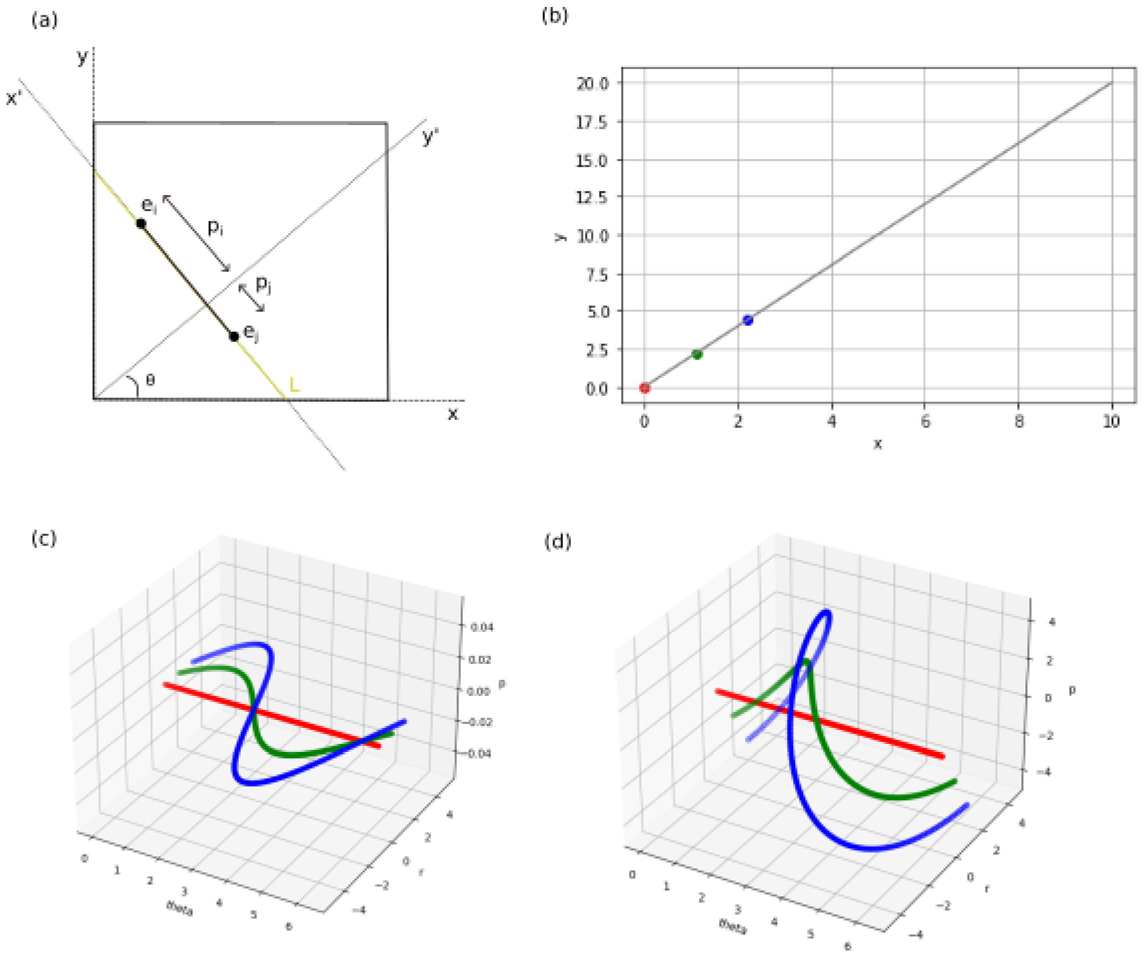


Figure 3.1: Rethinking hough mapping

where R is the rotation matrix

Equation 3.1 can also be rewritten as: $x' = -x\sin\theta + y\cos\theta$ and $y' = x\cos\theta + y\sin\theta$. Recall Equation 2.2: $r = x\cos\theta + y\sin\theta$. Notice that the latter equation is the same as Equation 2.2. So, we can intuitively understand that in the new coordinate system, the perpendicular axis to the line from the origin in the cartesian coordinate system is the new y-axis. Any point on the line $y' = 0$ can be described with the equation:

$$p \equiv x' = -x\sin\theta + y\cos\theta \tag{3.2}$$

The relative coordinates of the points e_i and e_j in the new coordinate system are $(p_i, 0)$ and $(p_j, 0)$ respectively. So, given (r, θ) of a line is known, any point on an infinitely long line can be described in a local coordinate system with a single number p that locates where the point is along that line.

We can construct a 3D hough space with (θ, r, p) dimensions. A 3D hough space with (θ, r, p) dimensions holds information about a line and the location of points on the line. Essentially, we have added an extra dimension to the 2D hough space that encodes for positions of points on a line. In Figure 3.1, observe that an event from the cartesian space has the same 2D hough mapping if p is fixed to 0. Each point from the regular 2D hough curve as given by Equation 2.2 is pushed up in the p dimension with a value of p obtained from Equation3.2

3.2 Extracting Line Segments

The distribution of points in the p dimension informs us about the existence of a line segment. If two points belong to a line segment, then they will be located close together in the p dimension. A region of high density corresponds to line segments, whereas if the points are spread out in the p dimension, it is likely to be noise. The extrema points of a line segment can be found by looking at the region where the density fall-off is steep. Gradients provide information about these fall-offs.

Approximate gradients can be found by using a Sobel filter/operator. A Sobel operator for a 2D matrix A is a 3×3 kernel that calculates the approximations of the derivatives G_x, G_y when convolved. The convolution operator is given by $*$. The Sobel operator for the x and y direction changes is given by:

$$S_x = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} \; S_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$
$$G_x = S_x * A$$
$$G_y = S_y * A$$

They can be generalised to more dimensions since Sobel kernels are decomposable. They can be decomposed as the products of an averaging and a differentiation kernel,i.e. they compute the gradient with smoothing. Hough curves do not overlap when projected in the p dimension as shown in Figure 3.1. So, smoothing is necessary to increase the probability of overlap.

$$S_x = \begin{bmatrix}1\\2\\1\end{bmatrix}\begin{bmatrix}1 & 0 & -1\end{bmatrix} = \begin{bmatrix}1\\1\end{bmatrix} * \begin{bmatrix}1\\1\end{bmatrix}\begin{bmatrix}1 & -1\end{bmatrix} * \begin{bmatrix}1 & 1\end{bmatrix}$$

$$S_y = \begin{bmatrix}1\\0\\-1\end{bmatrix}\begin{bmatrix}1 & 2 & 1\end{bmatrix} = \begin{bmatrix}1\\1\end{bmatrix} * \begin{bmatrix}1\\-1\end{bmatrix}\begin{bmatrix}1 & 1\end{bmatrix} * \begin{bmatrix}1 & 1\end{bmatrix}$$

For a 3D hough space, we require a 3D Sobel operator, a 3x3x3 voxel. The Sobel operator for calculating the approximate gradients in the z-dimension S_z is given by:

$$S_z = \begin{bmatrix}\begin{bmatrix}1 & 2 & 1\\2 & 4 & 2\\1 & 2 & 1\end{bmatrix} & \begin{bmatrix}0 & 0 & 0\\0 & 0 & 0\\0 & 0 & 0\end{bmatrix} & \begin{bmatrix}-1 & -2 & -1\\-2 & -4 & -2\\-1 & -2 & -1\end{bmatrix}\end{bmatrix} \tag{3.3}$$

$$G_z = S_z * A$$

After finding the extrema points of line segments, we need to find which pairs of endpoints belong to a line segment. Pairs of nearby points can only belong to the same line segment. We can search along the p dimension between two points and check whether the in-between gradient values are above a threshold. It is reasonable to set this threshold to be 0.5 times the minimum of the gradient values of the two points in consideration. If the gradient values are above a threshold, then we can say that they belong to the same line segment. If two identified line segments have a common endpoint, we know it's a corner.

One of the problems with this approach is that the Sobel filter might identify noisy points more strongly and wrongly as the endpoints. The endpoints of a line segment received are noisy. Noisy inputs can be elegantly handled by using spiking neurons. As mentioned in Chapter 2, SNN adaptation also helps reduce the number of neurons required using population coding.

3.3 Spiking Line Segment detection

Let's consider the cartesian image space to be a sheet of 2D LIF Neurons called *Cameras neurons*. For detecting line segments, the Hough space is a 3D block of LIF Neurons called *Hough block*. We can define synaptic connectivity between 2D Camera neurons to 3D Hough neurons using Equation3.2. So, each Camera neuron is connected to a curve of Hough Neurons described by Equation3.2 that lives in 3D space. We have another 3D *Readout block* of LI neurons that is one-to-one connected to the Hough block.

Let's consider the case for unconstrained 3D Hough space,i.e. the Hough space dimensions are larger than the image dimensions:

We activate a ribbon of neurons, as shown in Figure 3.3, called a *Hough ribbon*,i.e. they increase their membrane voltage. The width of the Hough ribbon is 1 unit of the

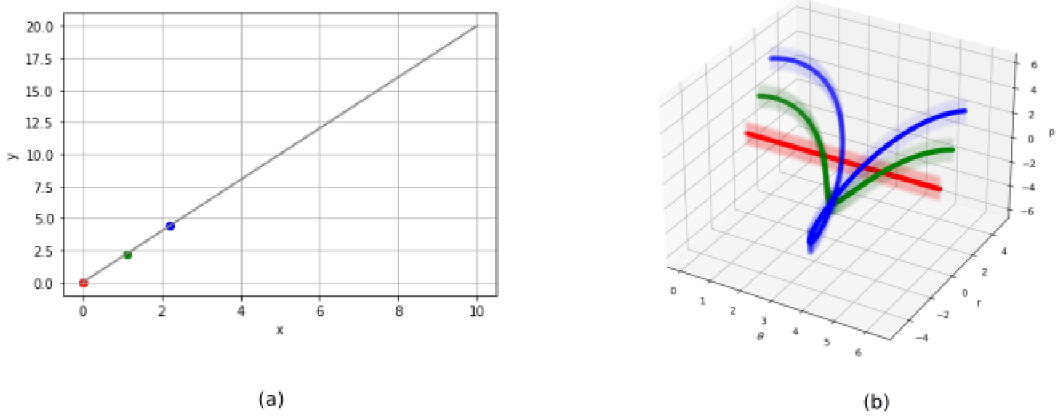


Figure 3.2: Hough ribbons: *(a) Input event camera space (b) Corresponding ribbons of neurons activated in the Hough space*

r dimension, and the height is confined within a height threshold. The values of p are spread according to a Gaussian within the height of the ribbon. So, when *Hough ribbons* overlap in the 3-dimensional space, the membrane potential increases beyond threshold voltage and causes a spike. Usually, the endpoints of a line segment from an event camera can be hard to bound. The Gaussian spread-out ensures that the effect of noisy inputs is minimised. Multiple closely lying points part of a line segment would overlap and cause a spike. Noisy events far away from the line segment do not contribute to a spike.

A spike in the Hough space increases the membrane voltage of the Readout block of neurons. The extrema points can be detected by looking for a sudden drop in the presence of a high number of spikes along the third dimension in the Readout block. It results in a blob of activity. At every timestep, we perform a Sobel convolution to enhance the gradient of change/enhance the sudden drop. We find extrema points as those points which have the highest value. The line segment can be detected by checking whether the value between the nearest endpoints is non-negative or not.

Population coding can translate line segment detection problem to a confined Hough space, i.e. the Hough block dimensions are lesser than the input event camera dimensions. Along with the p dimensional Gaussian spread, the hough ribbon is also spread-out in the r dimension such that the width of the ribbon changes from 1 to a certain *bandwidth*. Again, the r spread-out follows a Gaussian. During the identification process after Sobel filtering in the Readout block, we find the highest value of the gradient change. Within a cubic window around the highest value, we perform weighted averaging to get the actual extrema points of a line segment. To determine whether 2 extrema points belong to the same line segment, we look along the p dimension between the 2 nearest highest values.

The decay of membrane voltage of LI and LIF neurons ensure that the line segment can be tracked by the detect-and-track approach.

3.4 Methods

3.4.1 Weight matrix of line segment detection

To construct a 3D weight matrix, we build upon the 2D weight matrix construction process. The third dimension, p, describes the distance of a point on the line L described by $(r,)$ from the point where the perpendicular intersects L. The mapping of an event to p is given by Equation3.2. The p dimension has n_p^{out} discretisation. Previously in the 2D case, we found the Gaussian of r_{dists} as the activity assigned to the weight matrix. Now we also spread the activity in the p dimension and find p_{dists}. We perform an outer product of both these values to find the activity to be spread out 3-dimensionally in the weight matrix. The limits of r and theta are kept the same from the 2D case as $[0, 2\pi)$ and $[0, diagonal]$ respectively.

The range of p is $(-diagonal,diagonal)$. We can prove the upper and lower bounds of p by considering an event point d in the input space, a very small δ distance away from the origin. Several lines can pass through d, of which there exists a line l_δ whose $r = \delta$ and perpendicular to line intersects at d. l_δ described by $(\theta_\delta, r_\delta)$ is δ distance parallel to the diagonal of the input space. The farthest point on l_δ is $\delta \cos\theta$ distance away from the corner point with coordinates n_x^{in}, n_y^{in}. When $\delta \to 0$, $l_\delta \to diagonal$. Depending on the direction of l_δ, the value of p can be $-diagonal$ or $diagonal$. If there exists a point whose distance is $> diagonal$, it should belong to a line longer than the diagonal. But that's impossible since the diagonal is the longest line of event input space.

Algorithm 2 Generating 3D weight matrix

```
 1: $\theta$=[0,2$\pi$) r=[0,diagonal] p=[-diagonal,diagonal]
 2: for each event camera pixel $(x_i, y_j)$ do
 3:     $r \leftarrow x_i \cos\theta + y_j \sin\theta$                      ▷ curve equation 2.2
 4:     $p \leftarrow x_i \sin\theta + y_j \cos\theta$
 5:     for each point on the perfect curve $(\theta, r, p)_k$ do
 6:         $\theta_{pos} \leftarrow$ position of $\theta_k$
 7:         $r_{pos} \leftarrow$ positions of r
 8:         $r_{dists} \leftarrow$ distance of r to $r_k$
 9:         $p_{pos} \leftarrow$ positions of p
10:         $p_{dists} \leftarrow$ distance of p to $p_k$
11:         $dists \leftarrow r_{dists} \otimes p_{dists}$                ▷ outer product
12:         weightMatrix[x$_i$, $y_j$,$\theta_{pos}$,$r_{pos}$,$p_{pos}$] $\leftarrow$ gaussian($dists$)
13:     end for
14: end for
```

3D weight matrix check: We have an input camera space of 30x30. The 3D weight matrix is of size 30x30x60. The dimension of p is double the magnitude of r because of the range of p is twice r. Figure 3.3(a) shows the weight matrix activation for an extreme point of the input space (29,29). The blue points are positions of neurons and the black line is the perfect curve described by Equation3.2. To better understand the spread, Figures 3.3(b) and (c) show the projection to the $\theta - r$ and $\theta - p$ dimensions. We see that in both projections, the value of neurons decay away from the perfect value as we expect from a Gaussian spreadout. There are regions where there are no blue points. This is because we limit the value of r to be positive. Hence, as noticed in (b) there are no points corresponding to negative values of r.

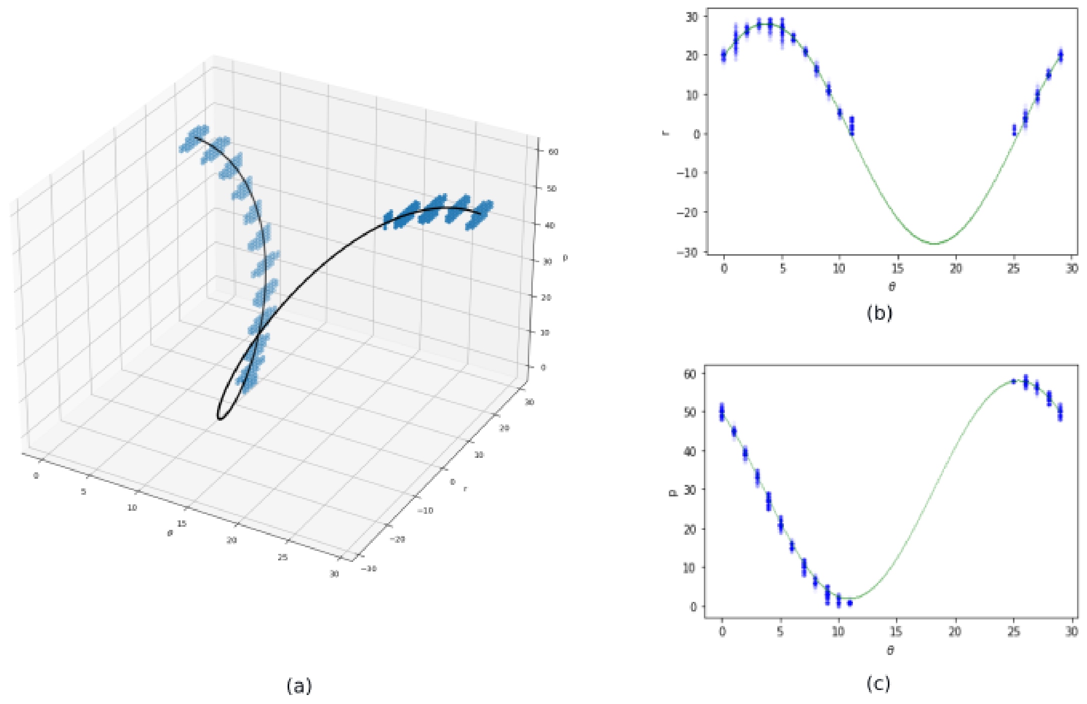


Figure 3.3: Hough ribbons: *(a) 3D weight matrix (b) θ-r projection of weight matrix (c) θ-p projection of weight matrix*

3.4.2 SNN adaptation

To validate the idea using GPU and Norse, we can use the same ideas adapted from the 2D SNN adaptation. We use the LIFBox model for Hough block of neurons and LI neurons as the Readout block of neurons. We define an initial PyTorch linear layer comprising of n_x^{in}xn_y^{in} neurons. We initialise the weights of the linear layer with the weight matrix. The linear layer is connected to the LIFBox block using all-to-all connections whose weights are modified by the weight matrix. The LI layer is one-to-one connected to the LIFBox block.

3.4.3 Sobel filtering

Once we have the activity of all the neurons in the 3D Readout neurons, we can perform sobel filtering to enhance the drop off in the p dimension. We perform a convolution of the sobel operator with the 3D Readout block as described by Equation3.3. But, as described in Chapter 2, θ has a wrap around. Hence, we project the sobel filtered 3D values to a 4D space where this wrap-around is accounted for. The transformation function is given as

$$g_t : (\theta, r) \rightarrow (\sin\theta, cos\theta, r, p)$$

After this, we can find the maximum activity of neurons and the 5x5x5x5 window around them where we perform weighted averaging. The choice of window size is based on the spread-out value in the r and p dimension considered before. To get back the (θ, r, p) values we use the inverse transformation function:

$$g_t^{-1} : (sin\theta, cos\theta, r) \rightarrow (\tan^{-1}(\frac{sin\theta}{\cos\theta}), r, p)$$

This process is performed for every event frame after we receive changes in the Readout block.

To determine whether 2 extrema points belong to the same line segment, we look along the p dimension between the two nearest highest values. Let two extreme points be e_1, e_2 with activity values a_1, a_2 respectively. Consider all Readout neurons as nodes in a 3D space. We construct an edge between two nodes if the value of the node is 0.5x$(min(a_1, a_2))$. If there is a graph theoretical notion of *path* between e_1, e_2(multiple edges forming a distinctive connection between $e1, e2$), they belong to the same line segment. No two *paths* can be same since every line segment is unique.

Chapter 4

Results and Discussion

4.1 Qualitative tests

We run our algorithm on neuromorphic hardware SpiNNaker and GPU, which give similar qualitative results. It shows that our algorithm is a general-purpose algorithm suited for any neuromorphic accelerating hardware. Discussed below are qualitative results from the GPU implementation with Norse to visually inspect the results of the algorithm. Each event frame has 500 events binned for qualitative tests. Code and results for the SpiNNaker version are available here: `https://github.com/theLamentingGirl/Master-Thesis`

Single Line: We consider 1000 timepoints of an event camera recording. Figure 4.1 (a) represents input events as points along with the predicted green line for the 100^{th} timestep (1 ms). The predicted values for 200^{th} timestep are $\theta = 17.87°$, r=233.76. The choice for inspecting after several timesteps were to let the activity build up in the Readout layer, which stabilises line tracking. The line is moving(translation motion), and line detection is performed at every timestep to enable tracking as visualised in Figure 4.1 (c). We observe that the single-line detection and tracking are pretty good.

Multiple lines: Figure 4.2 shows predictions for two lines with three orientations that translate within the confines of the event camera frame. These line orientations are typical cases when the algorithm is used for a collision avoidance task where it detects the legs of a chair as two parallel lines or the contours of an object as angled lines.

We make observations about the tracking ability of our algorithm by examining Figure 4.2:

The line detection at the 200^{th} timestep seems reasonable from inspection of the superimposition of line prediction with events for all three cases. FigureA row1 shows a lag in r compared to the actual events. Observe that its corresponding Readout layer, FigureA row2, shows two separable blobs of activity of which one of them is feeble.

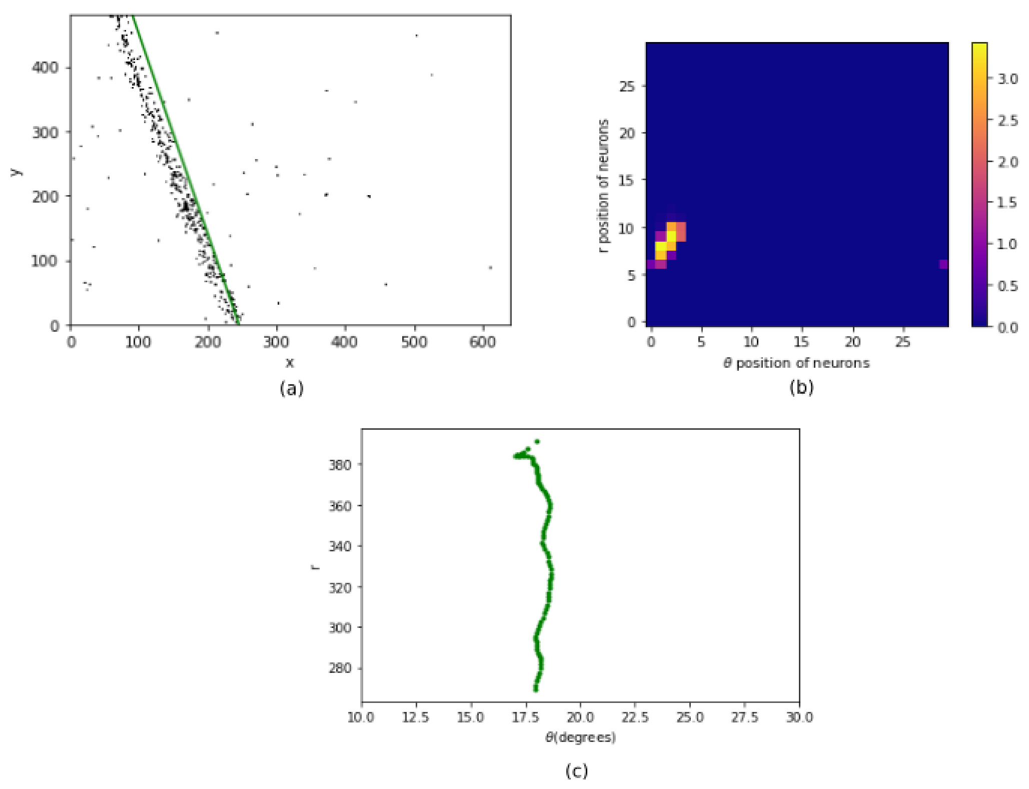


Figure 4.1: Single line tracking prediction: (a) Line prediction + input events (b) Readout layer activity (c) Line prediction over time

The activity of higher cluster is feeble and corresponds to the line that lags. The lesser activity due to fewer input events corresponding to that line might explain the lag. The line tracking predictions of r and θ are given by different colours in row 3 and row 4, orange and blue, to distinguish the two predictions. The colours do not correspond to a particular line. The r tracking shown in row3, seems to follow a trajectory along time except for a short time segment (200,450). The orange line prediction seems to do haphazardly away from the otherwise clearer trajectory. This can be because the line reaches the extremes of the event camera frame. The theta predictions of tracking fluctuate only by 8°. Hence, we can convince ourselves that the motion is primarily translation.

In FigureB row1, we see that the vertical line prediction is only off by a small angle. In the corresponding Readout space FigureB row2, we see that neurons towards the edges of the 2D sheet get activated. At the extremes of the Hough space, one can expect distortion. This is because uniform parametrisation of theta is not the best discretisation possible (but the simplest). If a line is located near the perpendicular, a small change in θ results in a small displacement of r. But, the same amount of change in θ at the extremes, far away from the perpendicular line, causes large displacement. By choosing a uniform discretisation of θ, we represent the middle values of r in the Readout layer more finely than extreme values. From FigureB row3 and row4, we can see that the prediction of r shows high variance and θ fluctuations are within 5°. Similarly in FigureC, the angle predicted for the 200^{th} timestep is off by a small amount. The variance of tracking r and θ is high.

Line tracking accuracy isn't optimal, as discussed in the above observations. This can be due to the following:

1. Latency of detection and encoding
2. Hough band interferences for lines that are very close to each other
3. Size of Hough space.

We can address these problems by identifying the limit upto which the hough interferences don't affect the detection quality, replaying the recording of event data in reverse to find out the lag due to spiking neuron accumulation, and searching through different sizes of Hough space. We discuss these issues during the quantitative analysis.

4.2 Quantitative tests

Firstly, setting up a baseline error against which we can compare the modifications is essential. We create two baseline error values, one for the translating line and the other for the rotating line: a single translating artificial line at $\theta = 45°$ with 30x30 hough space; a single artificial rotating line at r=280 with 30x30 hough space. The error is calculated as $(\frac{|actual-predicted|}{\#timepoints})$. Figure 4.3 shows the baselines. The parameters changed are with respect to the baseline and are mentioned.

The mean average error in r is 16.90, which is less than one discretisation of the r dimension $\frac{diagonal}{n_r^{out}} = \frac{800}{30} = 26.66$. The standard deviation of absolute errors in r is 8.66, which is nearly one-third of one discretisation of r. The minimum absolute error is 1.34 and the maximum absolute error is 39.23. The maximum errors are towards the r extremes of the Hough space, whereas the minimum error is towards the centre as can be seen in Figure 4.3 as well. The mean average error in θ is 1.82, and std is 1.26, negligible compared to the discretisation of θ: $\frac{2\pi}{n_\theta^{out}} = \frac{360}{30} = 12$. The minimum and maximum values are 0.01 and 1.55, respectively.

4.2.1 Analysis

Latency of detection: Let's consider the baseline translating line moving forwards and backwards, and identify the latency error due to residual activity in neurons. For each neuron position, we find the absolute difference between the forward and backward predictions and find the averages of neuron positions across timesteps. $latency = \frac{|backwardsestimate-forwardestimate|}{n}$. We observe that the error while reverse playing the same data recording, the error is lesser. This can be due to [?].

Hough interferences: Consider two moving lines towards each other. One line starts at $\theta = 45, r = 100$ and moves at a velocity = 0.25, whereas another line starts at

$\theta = 45\, r = 600$ and moves at a velocity = -0.25. Both terminate at $r = 350$ after 1000 timesteps. In this setup, we take note of the distance between the two tracked lines. The distance decreases, so the slope of the actual distance in between the two lines is supposed to decrease. We find that the hough interferences start playing a huge role when the slope of distance in between the two lines stabilises near or above zero. Figure 4.5(a) shows the tracked line over time, where the green lines represent the true values of the lines. Figure 4.5(b) shows the distance between two tracked lines, where the green line gives the actual distance between the two lines. At timestep=704, we notice that the slope stabilises to become positive; the actual distance in between is 147.64, plotted in Figure 4.5, where we can see that the tracking accuracy rightly becomes worse.

Size of hough space: We build weight matrices of varying sizes to find the value for which the value of the mean absolute error of line tracking is the least. We keep the input image space constant and brute force search for the least error pair of hough dimensions $(n_\theta^{out}, n_r^{out})$. for line detection. We see that 60x60 is the most optimal.

4.2.2 Scalability

The algorithm is designed keeping scalability in mind so that the algorithm can be utilised with any neuromorphic system. We show how scaling the image or the hough space affects the line tracking results. It is seen that the number of hough neurons required for a robust output is smaller than the number of event neurons.

Scaling the Image space: We consider 10 different sizes of image space with $n_x^{in} = n_y^{in}$ from 100 to 1000. We only consider a square input image space since we want to check only for the size that We plot their corresponding r errors and θ errors as shown in Figure 4.6. Line detection accuracy stays similar. Hence, we can say that our algorithm can take in variable inputs and do decently

Scaling the Hough space: Line detection accuracy for a single line increases and then stagnates. For two lines it's better to have a bigger matrix, that solves hough interferences

4.3 Task evaluation

We present line-tracking results for two tasks with real-time event camera inputs. Here each event frame has all events received within one millisecond.

Tracking a pendulum: We demonstrate the tracking ability of the algorithm by tracking a pendulum swinging with real event camera inputs as well as real-time tracking. We see that there's a lag, but the overall tracking seems to do well. The video of pendulum tracking can be found here

Tracking shapes: We track the stimulus of a rectangle. We see that the algorithm performs well in detecting the four lines that comprises the rectangle. The video of tracking a rectangle can be found here

4.4 Challenges and Limitations

We see that the lag due to the accumulation of spiking neurons can be an issue when tracking fast-moving obstacles. The algorithm cannot detect lines in a crowded setup. That will require us to scale up the hough space. But, the proof of concept here shows that even for cluttered spaces, we can limit the number of neurons required, and the neurons do not have to be in the same order as the number of event pixels. Depending on the application scenario, one can tune the size of the Hough space. We provide an initial comparison that shows that the granularity of r needs to be much greater than the granularity of theta.

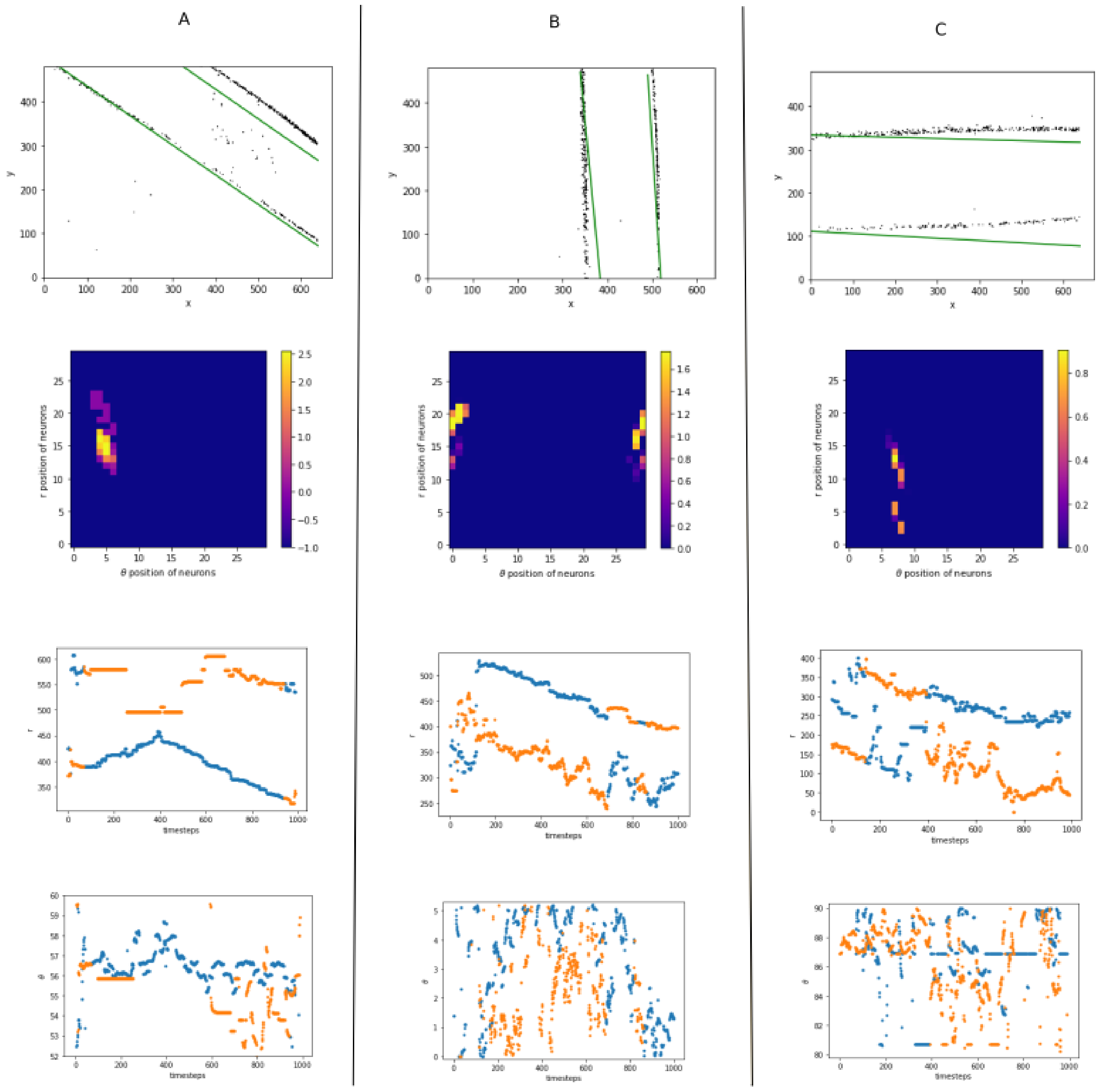
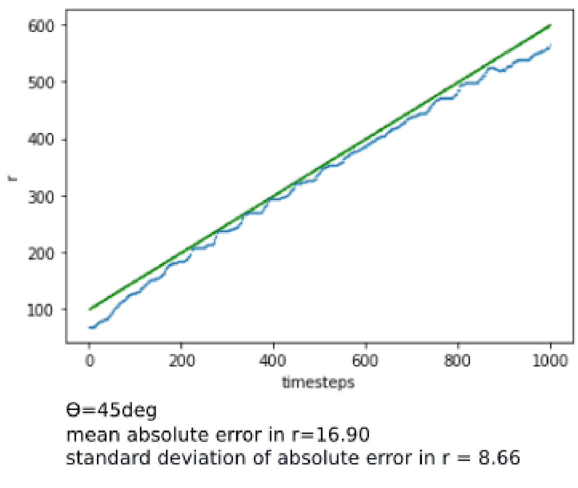
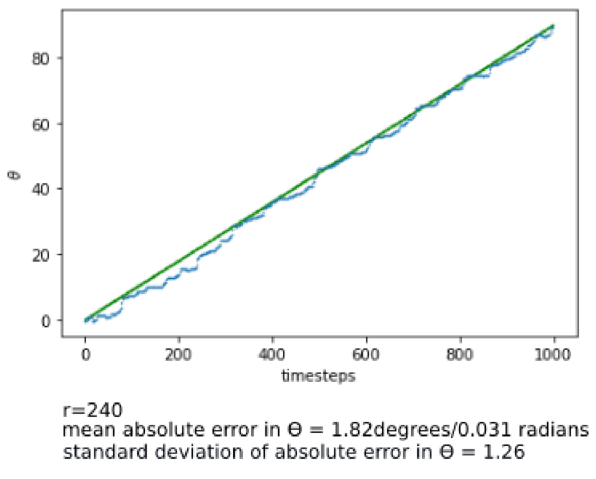


Figure 4.2: Multiple line tracking prediction: **A** Slanted lines **B** Vertical lines **C** Horizontal lines; (Row1)Line prediction + input events at t=200 (Row2)Readout layer activity at t=200 (Row3)r predictions over time for translating lines (Row4)θ predictions over time for translating lines



Θ=45deg
mean absolute error in r=16.90
standard deviation of absolute error in r = 8.66

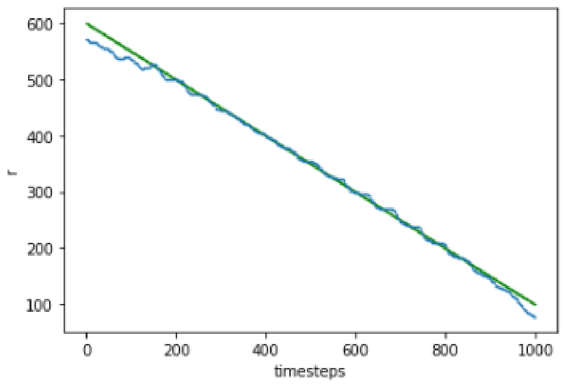
(a)



r=240
mean absolute error in Θ = 1.82degrees/0.031 radians
standard deviation of absolute error in Θ = 1.26

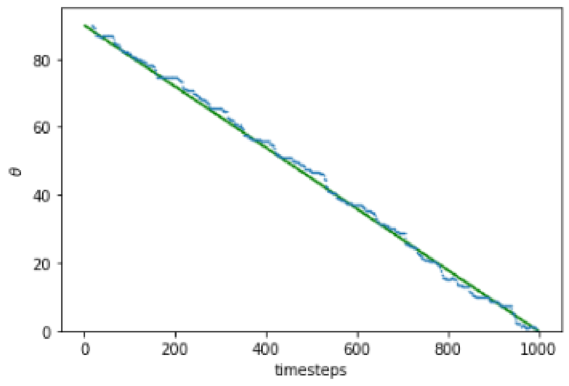
(b)

Figure 4.3: Baseline: (a)Translating line with speed=0.5units/timestep (b)Rotating line with speed=0.09units/timestep



mean absolute error in r = 6.89
standard deviation of absolute error in r =7.41
latency = 6.82

(a)



mean absolute error in Θ = 1.19
standard deviation of absolute error in Θ =0.80
latency = 2.58

(b)

Figure 4.4: Latency: (a) Reverse translation of baseline (b) Reverse rotation of baseline

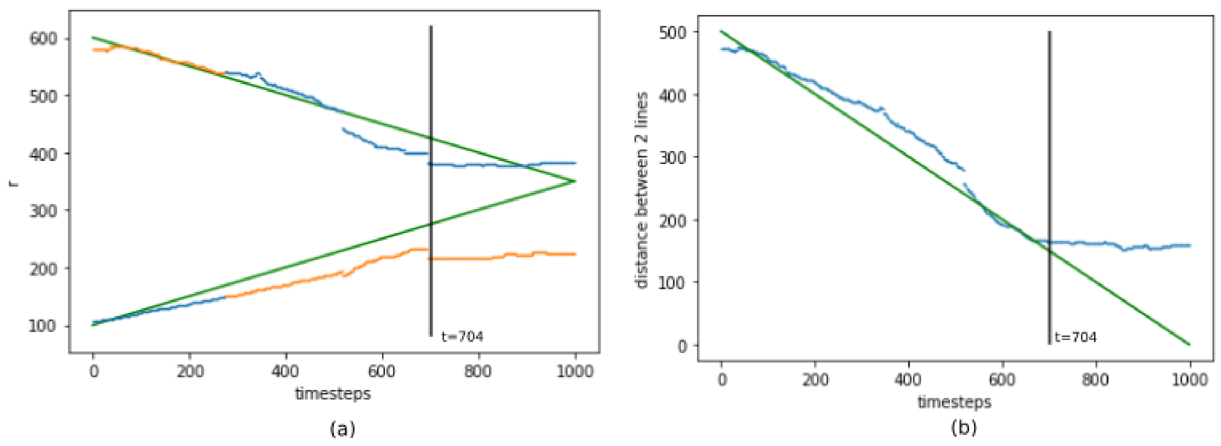


Figure 4.5: Hough interferences: (a) Prediction of two moving lines towards each other starting at r=100,r=600 with speed=0.25 (b)Prediction of the distances between two moving lines

Chapter 5

Conclusion

5.1 Summary

In this thesis, we identified lines as low-level features that act as building blocks on top of which other neuromorphic embedded vision algorithms can be designed. We have created a line-tracking algorithm with neuromorphic systems like event cameras and neuromorphic hardware. We approach line tracking by tracking infinitely long lines(referred to as lines) as well as tracking line segments. The line tracking algorithm provides us with the distance from the origin and angle to the perpendicular to the origin (r, θ). The line segment tracking algorithm provides us with (r, θ) along with the start and end point estimate of the line segment. We use the detect and track approach for tracking lines and line segments,i.e. we implement line detection at every timestep to enable line tracking. Our line segment detection is an extension of line detection. Line segments form better low-level information than lines do.

We have exploited the novel paradigm of sparse event camera visual inputs by using SNNs. The line tracking algorithm is a Hough Transform implemented with SNNs. We treat each event independently from the other; each spiking neuron processes events independently. The benefits are twofold: It reduces the influence of noisy events contributing towards line tracking and allows sparse computation on GPUs and SpiNNaker. The benefits of using SNNs optimise the line tracking for real-time neuromorphic setup. The algorithm utilises lesser computational resources than traditional implementations of Hough transforms since the voting phase of Hough Transforms is dealt with by using spiking neurons. We show that the general perception of having a bigger hough space than the image space can be limited by carefully including population coding techniques. Previous research(Seifozzakerini, Yau, Zhao, et al. 2016)(Seifozzakerini, Yau, Mao, and Nejati 2018) have implemented Spiking Hough transforms; this thesis tried to push the boundaries towards real-time embedded neuromorphic use case scenarios by reducing the number of neurons required for computation without compromising on the quality of the incoming information. We have analysed that the tracking quality is within reasonable limits when the input dimensions are scaled up or down. We showed the feasibility of implementing the infinitely long line detection algorithm on SpiNNaker and GPUs, two different accelerating hardware. It shows that our algorithm is adaptable to any neuromorphic *computer*. Our goal is to do well as an algorithm that is best suited for the neuromorphic paradigm. Building on the foundation of line detection, we show promising results for line segment detection with preliminary qualitative tests.

5.2 Future directions

This thesis provides ideas to adapt traditional hough transforms into the neuromorphic paradigm. SNN adaptations of Hough transform can also be used to detect and track parametric shapes and templates. Further extensions of our algorithm can be done to make line and line segment detection more robust. Currently, clustering is performed on a CPU with KMeans. This requires that the user provide the number of clusters to be found. But, using other clustering techniques that don't require cluster number inputs can be helpful in identifying all the lines in a scene. Also, using a neuromorphic clustering algorithm can ensure that all the computation happens on parallelised hardware, reducing the total computation time even further. Inhibitory and excitatory connections can be used for neuromorphic clustering algorithm design. More experiments and analyses need to be performed that can verify the feasibility of the line segment detection extension.

This thesis paves the path forward for many on-edge application scenarios with neuromorphic computers and event-based cameras. One of the straightforward applications of the line-tracking algorithm is lane keeping for autonomous vehicles and horizon tracking for flying drones. Other on-edge applications include object detection, collision avoidance, etc. For instance, when an autonomous car is driving around, and we require it to avoid lamp posts and trees, it can do so but detecting these objects as lines initially and finding which of the detected lines move consistently. Consistency of moving lines that are close to each other hints at an object in view. We can further think about extensions that have lines as their building block. One of the possible discomforts of line tracking through detection is when the object is out of view for a few seconds or is occluded. In such scenarios, we can implement a Hough-Hough Transform. In this idea, we consider that the vehicle is moving at a constant speed, and the environment is stationary. The line detection at every timestep forms a line in the Readout space over time. Hence, if we project these line predictions to a new Hough space similar to how we project events to the current Hough space, we have a Hough-Hough space that encodes trajectories. So, whenever an object is out of sight, we predict the trajectory it should have been moving by inferring from the Hough-Hough space.

Another interesting application of line detection/ tracking using event camera inputs is to identify the lens distortion and correction necessary in event cameras. This can be done by displaying a single line on a huge monitor with known parameters. We use DVS to capture this line and check the deviation from the actual parameters. The difference can be used as a correction factor for the event camera lens.

Appendix A

Appendix

A.1 Generalised Hough Transform

The Generalised Hough Transform (GHT) was proposed to identify curves and shapes whose parameters are unknown(S. Rahmdel et al. 2015). The most important thing to remember is that this method requires knowledge of the edge points and the *edge direction* extracted from traditional images. *Edge direction* is defined as the unit vector in the direction of the edge that is normal to the direction of maximum intensity change. So, it is not suitable for the neuromorphic approach proposed in this thesis. But, the knowledge of the capabilities of hough transforms can help to explore other neuromorphic adaptations.

Implementing GHT traditionally is done in two phases:

The first phase is the offline phase; we require an ideal template of shape to be tracked, and a fixed reference point in the shape. We go through each edge point and ask, "Where is the reference point w.r.t to that point?". This gives us the distance of the point from the reference point r_i and the angle with the origin θ_i. Then we group together r_i and θ_i of all the edge points that share the same edge direction ϕ_k. This grouping is done by constructing a ϕ table. Each row of the ϕ table represents one edge direction ϕ_k and contains all possible $(r_i, \theta_i)_k$ groups. The ϕ table has discretised intervals of all possible edge directions for an image. ϕ table $= \{(r_i, \theta_i)_k\}$.

In the second phase, we have an image where we need to find our shape of interest. We compute the edge direction for all the points. We create an accumulator array of the same shape as the image dimensions. For a single input point with ϕ_k edge direction, we look up the ϕ table to find all possible distances and angles $(r_i, \theta_i)_k$ an ideal template point part of the shape would have been at. So, each input point votes for a point as if it were part of the template shape; which it would consider as the reference point. So, for a point with edge direction ϕ_k, we increment the accumulator array using the equations (voting process):

$$x_c = x_i \pm r_i cos(\theta_i)$$
$$y_c = y_i \pm r_i sin(\theta_i)$$

where (x_c, y_c) are estimates of the cartesian position of the reference point. We repeat the voting process for all input edge points.

Bibliography

AI and compute (2023). en-US. URL: https://openai.com/research/ai-and-compute (visited on 03/03/2023).

Bachiller-Burgos, Pilar, Luis J. Manso, and Pablo Bustos (May 2017). "A variant of the Hough Transform for the combined detection of corners, segments, and polylines". In: *EURASIP Journal on Image and Video Processing* 2017.1, p. 32. ISSN: 1687-5281. DOI: 10.1186/s13640-017-0180-7. URL: https://doi.org/10.1186/s13640-017-0180-7 (visited on 02/21/2023).

Blakemore, Colin and Elisabeth A. Tobin (Sept. 1972). "Lateral inhibition between orientation detectors in the cat's visual cortex". en. In: *Experimental Brain Research* 15.4, pp. 439–440. ISSN: 1432-1106. DOI: 10.1007/BF00234129. URL: https://doi.org/10.1007/BF00234129 (visited on 03/22/2023).

Chen, Guang et al. (July 2020). "Event-Based Neuromorphic Vision for Autonomous Driving: A Paradigm Shift for Bio-Inspired Visual Sensing and Perception". In: *IEEE Signal Processing Magazine* 37.4. Conference Name: IEEE Signal Processing Magazine, pp. 34–49. ISSN: 1558-0792. DOI: 10.1109/MSP.2020.2985815.

Czerwinski, R.N., D.L. Jones, and W.D. O'Brien (Feb. 1999). "Detection of lines and boundaries in speckle images-application to medical ultrasound". In: *IEEE Transactions on Medical Imaging* 18.2. Conference Name: IEEE Transactions on Medical Imaging, pp. 126–136. ISSN: 1558-254X. DOI: 10.1109/42.759114.

Davies, Mike et al. (Jan. 2018). "Loihi: A Neuromorphic Manycore Processor with On-Chip Learning". In: *IEEE Micro* 38.1. Conference Name: IEEE Micro, pp. 82–99. ISSN: 1937-4143. DOI: 10.1109/MM.2018.112130359.

DeBole, Michael V. et al. (May 2019). "TrueNorth: Accelerating From Zero to 64 Million Neurons in 10 Years". In: *Computer* 52.5. Conference Name: Computer, pp. 20–29. ISSN: 1558-0814. DOI: 10.1109/MC.2019.2903009.

Duda, Richard O. and Peter E. Hart (Jan. 1972). "Use of the Hough transformation to detect lines and curves in pictures". In: *Communications of the ACM* 15.1, pp. 11–15. ISSN: 0001-0782. DOI: 10.1145/361237.361242. URL: https://doi.org/10.1145/361237.361242 (visited on 03/10/2023).

Furber, Steve B. et al. (Dec. 2013). "Overview of the SpiNNaker System Architecture". In: *IEEE Transactions on Computers* 62.12. Conference Name: IEEE Transactions on Computers, pp. 2454–2467. ISSN: 1557-9956. DOI: 10.1109/TC.2012.142.

Gallego, Guillermo et al. (Jan. 2022). "Event-based Vision: A Survey". en. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 44.1. arXiv: 1904.08405, pp. 154–180. ISSN: 0162-8828, 2160-9292, 1939-3539. DOI: 10.1109/TPAMI.2020.3008413. URL: http://arxiv.org/abs/1904.08405 (visited on 03/14/2022).

Grübl, Andreas et al. (Nov. 2020). "Verification and Design Methods for the BrainScaleS Neuromorphic Hardware System". en. In: *Journal of Signal Processing Systems* 92.11, pp. 1277–1292. ISSN: 1939-8115. DOI: 10.1007/s11265-020-01558-7. URL: https://doi.org/10.1007/s11265-020-01558-7 (visited on 04/13/2023).

Hendy, Hagar and Cory Merkel (Jan. 2022). "Review of spike-based neuromorphic computing for brain-inspired vision: biology, algorithms, and hardware". In: *Journal of Electronic Imaging* 31.1. Publisher: SPIE, p. 010901. ISSN: 1017-9909, 1560-229X. DOI: 10.1117/1.JEI.31.1.010901. URL: https://www.spiedigitallibrary.org/journals/journal-of-electronic-imaging/volume-31/issue-1/010901/Review-of-spike-based-neuromorphic-computing-for-brain-inspired-vision/10.1117/1.JEI.31.1.010901.full (visited on 02/07/2022).

Illingworth, J. and J. Kittler (Oct. 1988). "A survey of the hough transform". en. In: *Computer Vision, Graphics, and Image Processing* 44.1, pp. 87–116. ISSN: 0734-189X. DOI: 10.1016/S0734-189X(88)80033-1. URL: https://www.sciencedirect.com/science/article/pii/S0734189X88800331 (visited on 03/08/2023).

Ivanov, Dmitry et al. (2022). "Neuromorphic artificial intelligence systems". In: *Frontiers in Neuroscience* 16. ISSN: 1662-453X. URL: https://www.frontiersin.org/articles/10.3389/fnins.2022.959626 (visited on 03/28/2023).

Lakshmi, Annamalai, Anirban Chakraborty, and Chetan S. Thakur (2019). "Neuromorphic vision: From sensors to event-based algorithms". en. In: *WIREs Data Mining and Knowledge Discovery* 9.4. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/widm.1310, e1310. ISSN: 1942-4795. DOI: 10.1002/widm.1310. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/widm.1310 (visited on 11/04/2022).

Law, Moore's (n.d.). "Unconventional HPC Architectures". en. In: ().

Lenz, Gregor (July 2021). "Neuromorphic algorithms and hardware for event-based processing". en. PhD thesis. Sorbonne Université. URL: https://tel.archives-ouvertes.fr/tel-03474197 (visited on 02/03/2022).

Li, Zhengrong et al. (Nov. 2008). "Knowledge-based power line detection for UAV surveillance and inspection systems". In: *2008 23rd International Conference Image and Vision Computing New Zealand*. ISSN: 2151-2205, pp. 1–6. DOI: 10.1109/IVCNZ.2008.4762118.

Mahowald, Michelle A. (1992). "VLSI analogs of neuronal visual processing: a synthesis of form and function". en. phd. California Institute of Technology. DOI: 10.7907/4bdw-fg34. URL: https://resolver.caltech.edu/CaltechTHESIS:09122011-094355148 (visited on 03/23/2023).

Moradi, Saber et al. (Feb. 2018). "A Scalable Multicore Architecture With Heterogeneous Memory Structures for Dynamic Neuromorphic Asynchronous Processors (DYNAPs)". In: *IEEE Transactions on Biomedical Circuits and Systems* 12.1. Conference Name: IEEE Transactions on Biomedical Circuits and Systems, pp. 106–122. ISSN: 1940-9990. DOI: 10.1109/TBCAS.2017.2759700.

Mukhopadhyay, Priyanka and Bidyut B. Chaudhuri (Mar. 2015). "A survey of Hough Transform". en. In: *Pattern Recognition* 48.3, pp. 993–1010. ISSN: 0031-3203. DOI: 10.1016/j.patcog.2014.08.027. URL: https://www.sciencedirect.com/science/article/pii/S0031320314003446 (visited on 09/26/2022).

Nawrocki, Robert A., Richard M. Voyles, and Sean E. Shaheen (Oct. 2016). “A Mini Review of Neuromorphic Architectures and Implementations”. In: *IEEE Transactions on Electron Devices* 63.10. Conference Name: IEEE Transactions on Electron Devices, pp. 3819–3829. ISSN: 1557-9646. DOI: 10.1109/TED.2016.2598413.

Pedersen, Jens Egholm and Jörg Conradt (Dec. 2022). *AEStream: Accelerated event-based processing with coroutines.* arXiv:2212.10719 [cs]. DOI: 10.48550/arXiv.2212.10719. URL: http://arxiv.org/abs/2212.10719 (visited on 04/01/2023).

Posch, Christoph et al. (Oct. 2014). “Retinomorphic Event-Based Vision Sensors: Bioinspired Cameras With Spiking Output”. In: *Proceedings of the IEEE* 102.10. Conference Name: Proceedings of the IEEE, pp. 1470–1484. ISSN: 1558-2256. DOI: 10.1109/JPROC.2014.2346153.

Reinagel, Pamela (Aug. 2001). “How do visual neurons respond in the real world?” en. In: *Current Opinion in Neurobiology* 11.4, pp. 437–442. ISSN: 0959-4388. DOI: 10.1016/S0959-4388(00)00231-2. URL: https://www.sciencedirect.com/science/article/pii/S0959438800002312 (visited on 04/13/2023).

Research Groups: APT - Advanced Processor Technologies (School of Computer Science - The University of Manchester) (2023). URL: http://apt.cs.manchester.ac.uk/projects/SpiNNaker/project/Access/ (visited on 04/13/2023).

Rolfs, Martin (Oct. 2009). “Microsaccades: Small steps on a long way”. en. In: *Vision Research* 49.20, pp. 2415–2441. ISSN: 0042-6989. DOI: 10.1016/j.visres.2009.08.010. URL: https://www.sciencedirect.com/science/article/pii/S0042698909003691 (visited on 04/13/2023).

S. Rahmdel, Payam et al. (2015). “A Review of Hough Transform and Line Segment Detection Approaches:” en. In: *Proceedings of the 10th International Conference on Computer Vision Theory and Applications.* Berlin, Germany: SCITEPRESS - Science, pp. 411–418. ISBN: 978-989-758-089-5 978-989-758-090-1 978-989-758-091-8. DOI: 10.5220/0005268904110418. URL: http://www.scitepress.org/DigitalLibrary/Link.aspx?doi=10.5220/0005268904110418 (visited on 02/20/2023).

Sayim, Bilge and Patrick Cavanagh (Oct. 2011). “What Line Drawings Reveal About the Visual Brain”. In: *Frontiers in Human Neuroscience* 5, p. 118. ISSN: 1662-5161. DOI: 10.3389/fnhum.2011.00118. URL: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3203412/ (visited on 11/02/2022).

Scheffer, Louis K. (Mar. 2021). “The Physical Design of Biological Systems - Insights from the Fly Brain”. In: *Proceedings of the 2021 International Symposium on Physical Design.* ISPD ’21. New York, NY, USA: Association for Computing Machinery, pp. 101–108. ISBN: 978-1-4503-8300-4. DOI: 10.1145/3439706.3446898. URL: https://dl.acm.org/doi/10.1145/3439706.3446898 (visited on 04/13/2023).

Seifozzakerini, Sajjad, Wei-Yun Yau, and Kezhi Mao (Aug. 2017). “Effect of Inhibitory Window on Event-Based Hough Transform for Multiple Lines Detection”. In: *Proceedings of the International Conference on Advances in Image Processing.* ICAIP 2017. New York, NY, USA: Association for Computing Machinery, pp. 39–44. ISBN: 978-1-4503-5295-6. DOI: 10.1145/3133264.3133286. URL: https://doi.org/10.1145/3133264.3133286 (visited on 11/25/2022).

Seifozzakerini, Sajjad, Wei-Yun Yau, Kezhi Mao, and Hossein Nejati (2018). “Hough Transform Implementation For Event-Based Systems: Concepts and Challenges”. In:

Frontiers in Computational Neuroscience 12. ISSN: 1662-5188. URL: `https://www.frontiersin.org/articles/10.3389/fncom.2018.00103` (visited on 11/03/2022).

Seifozzakerini, Sajjad, Wei-Yun Yau, Bo Zhao, et al. (2016). "Event-Based Hough Transform in a Spiking Neural Network for Multiple Line Detection and Tracking Using a Dynamic Vision Sensor". en. In: *Procedings of the British Machine Vision Conference 2016*. York, UK: British Machine Vision Association, pp. 94.1–94.12. ISBN: 978-1-901725-59-9. DOI: `10.5244/C.30.94`. URL: `http://www.bmva.org/bmvc/2016/papers/paper094/index.html` (visited on 11/25/2022).

Table of contents — Neuronal Dynamics online book (2023). URL: `https://neuronaldynamics.epfl.ch/online/index.html` (visited on 04/13/2023).

Willmore, Ben D. B., James A. Mazer, and Jack L. Gallant (June 2011). "Sparse coding in striate and extrastriate visual cortex". In: *Journal of Neurophysiology* 105.6, pp. 2907–2919. ISSN: 0022-3077. DOI: `10.1152/jn.00594.2010`. URL: `https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3118756/` (visited on 04/08/2023).

Yan, Shi-Xian et al. (Nov. 2019). "Microscopic Object Recognition and Localization Based on Multi-Feature Fusion for In-Situ Measurement In Vivo". en. In: *Algorithms* 12.11. Number: 11 Publisher: Multidisciplinary Digital Publishing Institute, p. 238. ISSN: 1999-4893. DOI: `10.3390/a12110238`. URL: `https://www.mdpi.com/1999-4893/12/11/238` (visited on 04/13/2023).

Zhou, Ying et al. (Mar. 2021). "Image-based onsite object recognition for automatic crane lifting tasks". en. In: *Automation in Construction* 123, p. 103527. ISSN: 0926-5805. DOI: `10.1016/j.autcon.2020.103527`. URL: `https://www.sciencedirect.com/science/article/pii/S0926580520311079` (visited on 04/13/2023).