# Neuromorphic Robotics for Drones

ROY BEN HAYUN, The Open University of Israel, Israel

Neuromorphic computing allows building computers and algorithms inspired by the architecture of the brain. Instead of using the classical Von Neuman architecture it uses electronic neurons which are sending spikes to each other for communication, computation and learning similar to how biological neuronal networks communicate, compute and learn.
The challenges in robotics make Neurorobotics - the application of euromorphic computing in robotics - a promising paradigm for building algorithms that will be executed in robots very efficiently in terms of performance, energy consumption, and very importantly learn as they move and operate.
In this work we will discuss how neuromorphic computing can address the challenges of Neurorobotics specifically in the field of drones.

Additional Key Words and Phrases: neuromorphic computing, spiking neural networks, robotics, drones

האוניברסיטה הפתוחה
THE OPEN UNIVERSITY OF ISRAEL
الجامعة المفتوحة

Prepared under the supervision of:
Dr. Elishai Ezra Tsur, The Open University of Israel

Author's address: Roy Ben Hayun, The Open University of Israel, Israel, roy.ben.hayun@iot-arch.com.

## 1 INTRODUCTION – NEUROMORPHIC ROBOTICS IN DRONES

Neuromorphic computing is a promising paradigm for designing algorithms and neuromorphic hardware inspired by biological neural networks which are known to solve complex problems with limited energy from insects to humans allowing perception, planning, control and learning.
The inherent parallelism and event-based nature of neuromorphic computing make it a promising paradigm for building neural networks architectures that will power future autonomous AI solutions, and specifically for robotics and drones in the context of this work.
In the field of neuromorphic engineering, Spiking Neural Networks (SNNs), are the brain-inspired software models used in neuromorphic computing. They are comprised of interconnected nodes linked through adjustable weighted lines, creating an organized network of layers through which spiking signals are propagated to make decisions in response to learned patterns over time. In parallel to the development of SNN based algorithms, distinctive neuromorphic hardware processors emerged, fundamentally different from the Von Neumann architecture. Biologically realistic neuronal models are realized directly in hardware, with co-location of memory and computing units comprised of neurons and synapses. The event-based asynchronous architecture of neuromorphic hardware leverages the asynchronous and event-based nature of SNNs to achieve orders of magnitude gains in power and performance over conventional architectures with an additional key benefit of on-chip plasticity enabling continuous learning in real-time. To name a few of these emerging neuromorphic processors: Neurogrid by Stanford University, TrueNorth by IBM, SpiNNacker by university of Manchester; and Loihi by Intel [6] which is the main neuro-chip used in the studied articles of this work (see appendix A [8] for technical information on Loihi).
Neuromorphic computing innovative architectural approach is therefore a great promise in powering autonomous AI solutions that require energy efficiency and continuous learning and is already in use in a variety of areas including vision, healthcare and large-scale AI applications. Particularly, the emergence of neuromorphic processors like Intel's Loihi yields great promises for SNN applications in robotics, and even more particularly in super-constrained systems such as drones.

### 1.1 Robotics

Before discussing neuromorphic robotics for drones, let us discuss what are the key challenges in the field of robotics in general, and how neuromorphic computing advances this field.
Core reason why neuromorphic computing is a good match specifically for robotics of all other optional fields is related to both precision and energy. The human brain for example excels in areas such as pattern recognition, cognition and learning. This is where neuromorphic computing shines as well. On the other hand, the human brain and neuromorphic computing are both not as good in solving problems requiring very high precision. Accordingly, the problems addressed in the field of robotics are located within the bounds of precision and energy where neuromorphic computing efficiency offers an attractive solution for autonomous robots that are energy constrained needing a good tradeoff between fit-for-purpose precision and excellent energy efficiency, and very importantly - learn as they move.
Robotic tasks can be very varied according to the robotic application. The ability to control a motor in order to achieve a goal state is a key ability of any robotic system. Robotics typically require perception coupled with motor control with efficient power and latency. Neuromorphic solutions for motor control were indeed suggested and realized in neuromorphic hardware i.e., control of a 3 DOF robotic arm in a 3D task space using the Neurogrid, neuromorphic head pose estimation experimented with Loihi in real-time on the iCub humanoid robot.

If we look further at mobile ground robotics tasks, they require a challenging combination of perception, vision and decision making together with the motor control. Mobile ground robots are also required to be capable to adapt to changing environmental conditions and possibly perform different tasks. Therefore, motor control, robust and fast perception, decision making and learning, all become critical components. Neuromorphic platforms were used in proof-of-concept experiments for variety of mobile ground robotic tasks such as robot navigation, path planning, and Simultaneous Localization and Mapping (SLAM). As an example, a neuromorphic approach to mapless navigation was experimented with a Turtlebot 2 and Loihi where after training, the robot successfully navigated in the real world showing significant energy efficiency and even higher rate of successful navigation to the goal. A mobile ground robot integrated with TrueNorth demonstrated ability to plan different trajectories according to either energy conservation or reducing travel distance traveled.
Other than conventional computing and processors, the field of robotic includes also Artificial Neural Networks (ANNs) and Deep Neural Networks (DNNs). These paradigms transformed the field of computer vision and are now increasingly used in robotics to support perception and learning, SLAM planning, and end-to-end control. However, ANNs and DNNs have their own drawbacks and challenges in their use in robotics. First, DNNs require substantial computing power often with GPUs to run in real time while mobile robots have stricter constraints on latency, energy, and payload than typical image processing applications. Secondly, a large amount of prepared data is required for the ANNs training procedure and then overfitting may come after the long training. And since the characteristics of a specific robot or task do not translate directly to another robot, it may be that even after long training the results may actually lead to lesser performance in the case of noise and disturbances, making adaptation in real-time and on-chip learning greatly missing. Third, these networks tend to grow while simple and smaller neural networks are preferable in robotics compared to non-robotic vision and image applications due to the low dimensionality of input and output space.
As much as robotics became advanced and effective, mobile robots are still limited in how well they can duplicate a very wide range of human behaviors, such as responding to changing environments. Mobile robotics need more efficient solutions also allowing them to adapt quickly in new environments and challenges. Similar to how biological neural networks allow perception, planning, control and learning with limited energy and limited local sensing only. This is where neuromorphic computing excels and brings a great promise of allowing robots to do not just repetitive tasks in pre-defined environment but also more expressive tasks and more loosely defined task, while adapting quickly in new stochastic and noisy environments, moving and operating safely in unknown complex and stochastic environments.

Let us now proceed to discuss neuromorphic robotics for drones.

### 1.2 Neuromorphic robotics for drones

Before describing drone's tasks, challenges and how neuromorphic computing fits, let us first describe what is a drone and its main components in the context of this work.
A drone in general may be various forms of an Unmanned Aerial Vehicle (UAV) either fixed wing, rotorcraft, quadcopter or else, which may operate under remote control by a human operator or autonomously. In the context of this work we will discuss Micro Air Vehicles (MAVs) which are small scale quadcopters powered by four rotors, one at each of the frame's corners and controlled remotely by a human operator. The quadcopter receives the operators command over a radio link via a radio receiver which is connected to the main computing unit on the drone - the Flight Controller (FC). The FC is comprised of the control SW running on an embedded processor i.e.,

Betaflight on CortexM processor, ArduPilot on CortextA etc. The FC is connected to the sensors on board of the MAV such as GPS, IMU, vision sensors, camera and else. The FC is responsible for aggregating the sensors data, sending the telemetry to an operator, receiving commands or making autonomous decisions to control the MAV. Speed and direction of the drone is controlled by rotation of the motors managed by Electronic Speed Controllers (ESCs) connected to the FC. The FC sends the commands to the ESCs via a serial protocol e.g., DShot and the ESCs control the motors rotation, varying the speed and direction i.e., by PWM signals to brushless motors.

Fig. 1. Example reference drone (with RC and FPV goggles)

### 1.3 State of the art and challenges in drones

There has been tremendous growth of the usage and number of drones in the last years within many industries. To name just a few - drones are being used in agriculture, construction, defense, homeland security, sports and entertainment. Their usage has expanded and converged. Drones are being used to fly over houses and forests to identify fire and point on fire fighters location, to follow and picture a bicycle rider, monitor facility pipes to identify leaks, scout pest infected plants and spray farm trees, carry military missions for defense and offense, deliver pizza or deliver life saving supplies to a missing traveler. They have indeed expanded and converged into our life. Yet innovation is happening at such fast pace that we can safely assume we are just at the beginning of their evolution, and many more innovative usages and technological advances will be developed in the coming years.

However, there are many challenges with drones and some inherent weaknesses. In some aspects they have additional challenges and constraints above those of ground robotics. These are some of the key challenges of drones:

- Energy constraints: a drone has to carry its energy source which brings a critical tradeoff between computing, payload and energy consumption
- Latency – to control a drone or for the drone to operate safely, a good and robust latency is critical
- Multiple operations - drones can perform very specific tasks within certain range of conditions. Multiple operations require additional sub systems with more sensors, payload, computational resources etc
- Limited autonomy – complete autonomy is a complex challenge that requires adaptive decision making with multiple sensors and sensory redundancy with significant computational resources
- Adaptivity – a human in the loop is still required. A human operator who can adapt to changes is still a need

To fulfil the promise of neuromorphic computing to address these challenges, first need to overcome other few considerable challenges. The two most important barriers towards robotic

applications are the reality gap between simulation and real world; and neuromorphic hardware integration with conventional computing frameworks. These engineering challenges apply not just to drones and are general to robotics. Additionally, we need to develop a much more pervasive neuronal approach with much more algorithms and computing primitives to be able to solve a variety of robotic tasks. The challenges for using neuromorphic robotics for drones, compared to a fixed robotic arm or a ground vehicle, grows even further considering the added complexity of controlling a 6-DoF UAV. As such, to the best of my knowledge at the time of writing this work, there is no report of any commercialized neuromorphic application embedded on any real world UAV or MAV. Not yet.

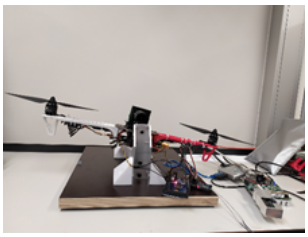
Interestingly enough, with only 100K neurons consuming limited energy and with limited sensing, flying insects can take off and land, orient themselves in an environment and find paths to their goals in unknown complex and stochastic environments, form associations, adapt and learn. Hoverflies, wasps and dragonflies perform highly demanding maneuvers, with accurate body control, precise roll and pitch movements and stabilization of their gaze. Fruit flies excel in navigation and flying in stochastic environments, avoid obstacles, perform complex maneuvers and achieve smooth landing. Biological systems from insects to animals and humans, indeed inspire robotics in their ability to perceive their surroundings with limited local sensing, move with under-actuated motor systems, operate in complex and dynamic environments solving complex perception, planning, and control problems with energy constrained biological neural networks.

This is where neuromorphic computing comes as a great promise for robotics in drones to deliver a low latency, low-power, responsive, adaptive AI at the edge. The low-power, asynchronous event-based nature and inherent parallelism of neuromorphic hardware make neuromorphic computing a promising paradigm for autonomous drones. The utmost accuracy is not mandatory and can be sacrificed for energy save, more flight time or autonomous control.

The promise of Neuromorphic Computing to drones is therefore to create smart algorithms and learning systems that will expose cognitive capabilities in real time with improved energy consumption and latency, allowing to design robust and efficient autonomous flight systems for autonomous drones.

Let us now discuss the studied articles which represent a mix of interesting capabilities applied to drones using neuromorphic computing and hardware.

## 2 EVENT-BASED PID CONTROLLER FULLY REALIZED IN NEUROMORPHIC HARDWARE: A ONE DOF STUDY



In this article [1] the authors present integration improvements and design improvements of a neuromorphic PID controller which they already implemented in a previous work. The PID controller, running on Intel's Loihi, controls the angle rotation of a drone constrained on a single axis i.e., rotating between -20° to +20° on the Y axis, according to a given desired angle.
Let us first discuss what is a PID controller, then elaborate on the motivation and the goals of this work, continue by reviewing what were the improvements applied and finally summarize the results and conclusions.

### 2.1 Background about PID

Proportional-Integral-Derivative controller, a PID controller, is a simple and common linear control loop mechanism widely used in control systems. The feedback loop is sensing the output of the plant - a drone in our case - measuring it against a reference signal and converts the error into a command so that the system will make adjustments accordingly. More specifically, the controller calculates an error value e(t) between a desired setpoint (SP) and a measured process variable (PV) and gives a correction action based on three P-I-D terms. As time progresses the error is driven to zero.
For example, if we have a robotic arm moving from 45 ° to 90 ° the error is the difference between the angles and the control variable is the torque we should apply to the motor.
In figure 2 can see a block diagram and the formula of PID:

$$e(t) = r(t) - y(t),$$
$$u(t) = K_p e(t) + K_i \int e(t)dt + K_d \frac{de(t)}{dt}$$

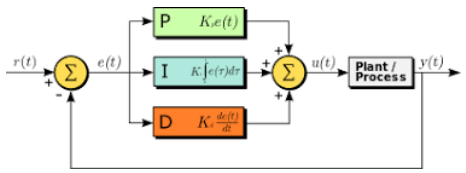



Fig. 2. PID forumula and block diagram

In the formula, e(t) is the difference between desired angle and actual angle, u(t) is the control value, torque in our case. The Kp, Ki, Kd in the formula are called PID Gains. Changing the gains allows the adjusting of how sensitive the system is to each of the P-I-D paths.

- The P term is the error scaled by Kp. It acts on the error to reduce it.
- The I term continuously sums the error over time and serves to prevent a steady state error.
- The D term is the rate of change of the error scaled by Kd. It acts on the derivative of the error, to regulate the rate of change i.e., the faster the error changes, the larger D path becomes.

The PID Gains can be tuned to a particular plant according to requirements.
This is the PID loop in pseudo code:

$eint \leftarrow 0$
$eprev \leftarrow 0$
**repeat** every dt seconds
  $e \leftarrow desired - readSensor()$
  $edot \leftarrow (e - eprev)/dt$
  $eint \leftarrow eint + e \times dt$
  $u \leftarrow K_p \times e + K_i \times eint + K_d \times edot$
  $eprev \leftarrow e$
  $sendControlSignal$
**until**

For example, every 1ms the feedback loop will sum the three P-I-D terms (difference of desired SP and sensory feedback, the derivative error and accumulated error) to produce a control signal. In the case of the robotic arm, every 1ms it will produce the torque to the motor based on actual angle difference.
Quantitative comparison of PID performance can use the following parameters:

- Overshoot: the amount that the process variable overshoots the setpoint, expressed as a percentage of the setpoint value
- Rise time: the amount of time the dynamic system takes to go from 10% to 90% of the steady-state value
- Settling time: the amount of time the dynamic system takes to enter and remain within a specified error band from the setpoint

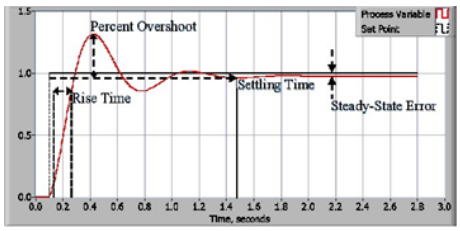



Fig. 3. PID performance parameters (source: https://www.ni.com/en-il/innovations/white-papers/06/pid-theory-explained.html)

## 2.2 Motivation and goals

For Spiking Neuronal Networks running on neuromorphic hardware to full-fill their potential of cognitive capabilities consuming low-power and low-latency, the neuronal computing architectures would be most efficient when all of perception, decision making, and motor control will be tightly integrated into a unified neuronal architecture running on neuromorphic hardware. At the same time, while there are various neuronal architectures for capabilities such as perception, neuronal motor controllers was somewhat less advanced in this field. The motivation of the authors therefore is to aim toward integrated neuronal capabilities, starting from motor control.
The goal set by this motivation is to improve a previous implementation of a neuromorphic PID controller running Intel's neuromorphic research chip Loihi to control a drone constrained to rotate on a single axis.

To understand what were the improvements goals, we should first understand what was the baseline for the improvements which is described in a previous article on previous work, then we will examine the previous work's conclusions as they were fundamental for setting the goals of this work.

**2.2.1 Goals set by baseline from previous work before improvements.** In the previous work [2] the authors chose to implement a PID controller motivated by the same thinking that edge AI cognitive capabilities also have to integrate with the motor control.
The UAV setup consists of a dual motor UAV mounted on a test bench, constrained to 1 DOF. The neuromorphic hardware running the SNN was Intel's Loihi, hosted in a Kapoho Bay device accessed through a USB interface. For obtaining the UAV's angle and angular velocity, the integrated IMU of a DAVIS240C sensor was used (Note: using the event camera sensor was out of the scope). The IMU and ESCs were interfaced using a Raspberri Pi. A $UP^2$ board was interfacing between the robotic environment and the Kapoho Bay. See figure 4.

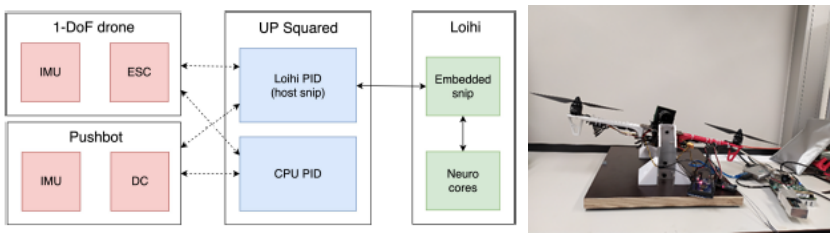



Fig. 4. SW and HW setup of the constrained UAV

To realize the algebra of the PID controller with neuromorphic computing, the design of the SNN includes two neuronal populations types:

- 1D N size Place-Code populations: represent numerical input and output values. These 1D populations do linear mapping of intervals using Place-Code encoding (rather than Rate-Coding). Every neuron in a population represents a numerical value ranging within the input limits i.e., values within the range [-lim, lim] are mapped to a single spiking neuron with index [0, N-1] in the 1D population.
- 2D NxN populations: realizing addition or subtraction of two 1D populations. The 2D populations receive the inputs of two 1D populations and send the arithmetic result to a third 1D output population constrained to same size as the input.

Throughout the network a One-Spike behavior was used in order to synchronize the neuronal computation with the UAV peripherals in 27ms time steps. Meaning, only one neuron in each population fires at any time-step. In 1D input populations only a single neuron fires in every time-step, in 2D populations a neuron can fire only if received two spikes in one time-step, and in the 1D output populations only a single neuron receives spike.
In figure 5 can see a diagram illustrating how the SNN-PID network works:

- Setpoint and sensory feedback are received in populations R and Y respectively (both 1D)
- Population EM (2D) calculates the error by subtracting the control signal from the sensory feedback
- Population IM calculates the integrated error signal by adding current error with the last integrated error

- Population DM calculates the derivative of the error by subtracting the one time-step delayed error signal from current error signal

In the SNN output population conversion between Place-Coding to Rate-Coding and back to Place-Coding is used. The PID outputs are converted to Rate-Coding for scaling with PID gains and future learning, then converted back to Place-Coding which is easier for conversion to motor commands.

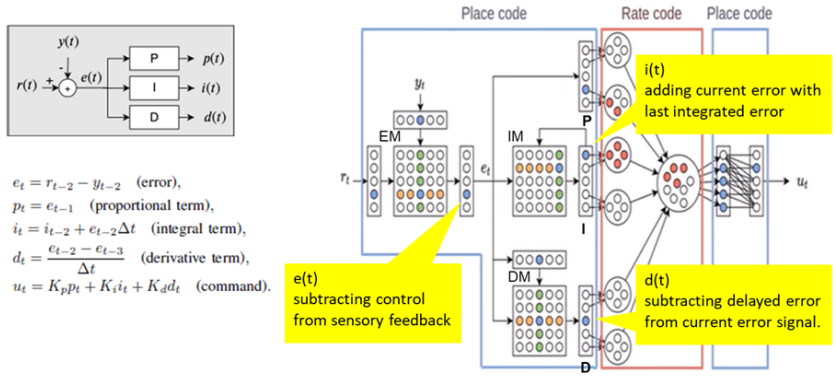



Fig. 5. SNN-PID network design in previous work

The SNN design was used twice, by two nested PID loops, in the drone setup:

- Outer PID: outputs target angular velocity based on target vs actual angular position
- Inner PID: outputs target torque based on target angular velocity from the Outer PID output vs actual angular velocity

The Outer PID controller receives sensory feedback from the IMU in population Y, eventually leading to output target angular velocity in output population U. This output is passed to the R input of the Inner PID with the actual angular velocity in population Y, eventually leading to the target angular acceleration in its output U population.
To run the two nested PID controllers, the neurons populations were segmented to run on 54 out of the 128 neuro-cores on one Loihi chip in 12 time steps.
Video clip of the PID controller in action can be viewed on https://youtu.be/lnGQqz7MM8w.

Please note that plasticity and on-chip learning was not applied in this work and was out of scope. It will still be discussed in the personal conclusions.

Now that we reviewed the background on what is PID control, let us go back to discuss the motivation and goals.

**2.2.2 Back to the goals in this work.** Now that we elaborated on the previous work, let us review the conclusion of the authors and how they led to the goals of this work.
Testing the PID controller was done by giving it a desired angle (SP) and comparing its behavior with a CPU based PID. However, the testing results indicated there are additional challenges in the setup and design before a robust and rigorous testing can be made. Specifically:

- Sensors and host CPU delays: the overall setup adds considerable delays due to the latency of the data flow between sensors to the host CPU and back to the UAV

- Long time-step duration: the time-step on Loihi can be as fast as 10$\mu$s. By optimizing IO the time-steps could be improved considerably
- Number of time-steps: the SNN uses 12 time steps of 27ms to flow through the nested PID controllers.

Figure 6 illustrates the outcome of these challenges.

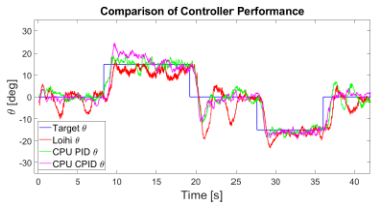



Fig. 6. SNN-PID vs CPU-PID comparison

The blue trace is the desired SP angle given to the UAV as a control signal. The red trace shows the performance of the SNN-PID running on Loihi. The green trace shows the performance of a conventional PID controller on CPU. The magenta trace shows the performance of another CPU-PID constrained to use same value resolution as the SNN-PID.
These plots together with additional quantitative measurements of PID overshoot, rise-time and settling time, indicated that the SNN-PID compares reasonably well to CPU-PID. However, it also led the authors to think that more optimizations must be made before any additional progress or any rigorous testing can be made. Which leads us to the improvements and motivation in the main article.

The goals in this work were therefore to improve the following:

- Improve the SNN: the previous limited resolution of value representation, led the I-path to fast saturation and was causing oscillations. The goal was to make it more robust.
- Simplify the SNN: to simplify the network, the SNN with two nested PID control loops was to be redesigned.
- Improve the IO: the goal was to decrease time step duration and the cycle duration of the control loop.

Achieving these goals was meant to serve the purpose of increasing control frequency and improving overall performance.

### 2.3 Improvements methods

Resuming from the previous work, the constrained UAV setup remained similar: a 1-DoF constrained UAV, an IMU inside a DAVIS240C sensor to measure angle and angular velocity, a Navio FC running on the Raspberry Pi controlling the ESCs of the two motors. A $UP^2$ board was interfacing between the robotic parts and Loihi hosted on a Kapoho Bay device over USB.
The design of the SNN was also similar. Input parameters are encoded in a 1D population, 63 neurons each, using Place-Code and updated once in every time-step. Arithmetic operations are performed by 2D populations that take two 1D populations input and send the addition or subtraction result to another 1D output population.
For the first goal of making the SNN more robust, the I-term integration module was the key target for improvements. In the previous setup when trying to optimize the time steps duration, due to certain limitations (low-resolution value representations) the signals saturated quickly which led to oscillations. Therefore, the authors implemented the error signal integration based on a path

integration network developed for neuromorphic SLAM [5] and head pose [4] in ground robotics. Figure 7 shows the improved integration module, which is using four auxiliary neuron populations. Two counter single-neuron populations for integration of positive and negative error; and two additional 1D neurons populations for shift up or shift down of the 1D output population. The refactored design of the integration module is not entirely straight forward and includes a handful of interesting biologically inspired techniques for computation using spikes: usage of excitatory and inhibitory connections, counter neurons with no-leak and high-threshold accumulating current over time for integration, shift populations changing position of an active neuron in the I population, splitting a population to neurons with either negative and positive values by different indices, different synaptic weights according to indices, synchronization of firing neurons.

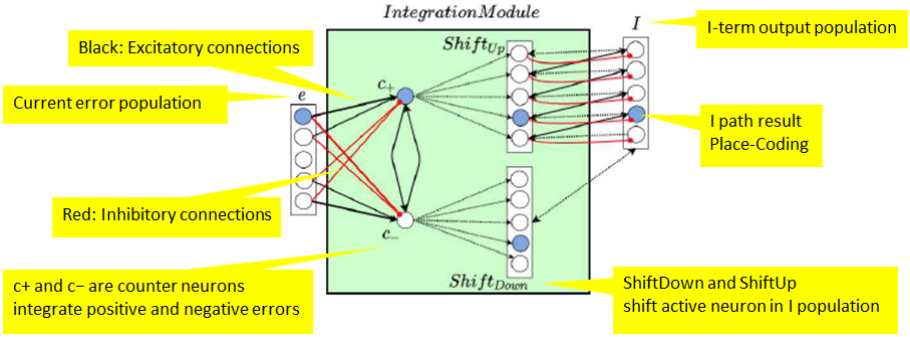



Fig. 7. The improved I-term integration module

The end result of the improved integration module is overcoming the limitation of 63 neurons low-resolution value representation in input populations which was causing oscillations, making the I term more robust as needed.
For the second goal of simplifying the SNN, instead of having two PID loops as in the previous article setup, this setup included a single PID loop. By that cutting down both the size of the neuronal architecture and also the number of steps required to calculate the control commands.
IO interfaces were also improved and delays were minimized. Eventually after the integration and SNN design improvements, the PID loop provided a new result at 1kHz frequency with latency of 10ms with only 6 time steps for the SNN execution, running over only 38 neuro-cores. This was a significant improvement from the previous work's setup in which velocity was sampled at 20Hz, needing 12 time-steps of 27ms each to go through two nested PID controllers taking around 320ms, segmented over 54 neuro-cores. The overall improvements led to a more efficient SNN, consuming less neuronal resources and applicable for more rigorous testing.

### 2.4 Results and conclusions

Once integration and design improvements were applied, it was time to continue more rigorously the measurements initiated in the previous article, of both performance and power usage.
The SNN-based PID controller running on Loihi was compared to a conventional PID controller with the constrained drone. PID gains for both controllers were tuned by hand.
The parameters used for quantitative comparison of the SNN-PID to the SW based PID were:

- Overshoot: percentage of measured angle moves further past the target 20° angle.
- Rise time: time it takes the drone to reach 90% of the target 20° angle
- Settling time: time it takes the drone to enter and remain within 20% of the target 20° angle

The table and figure in 8 show the quantitative comparison and behavior comparison respectively. Both indicate the controllers operate similarly well.

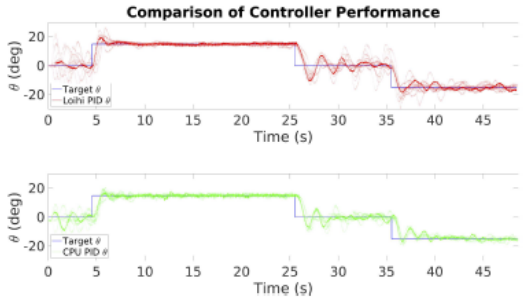



Fig. 8. comparison of SNN and CPU controllers performance

| Controller | Overshoot (%) | | Rise time (s) | | Settling time (s) | |
|---|---|---|---|---|---|---|
| *Step* | *+20°* | *-20°* | *+20°* | *-20°* | *+20°* | *-20°* |
| **SNN PID** | | | | | | |
| **Median** | 41.4 | 35.9 | 0.46 | 0.41 | 1.4 | 8.8 |
| $\pm\frac{IQR}{2}$ | *±13.8* | *±18.8* | *±0.11* | *±0.11* | *±0.65* | *±3.34* |
| **CPU PID** | | | | | | |
| **Median** | 24.0 | 20.1 | 0.53 | 0.49 | 1.3 | 4.3 |
| $\pm\frac{IQR}{2}$ | *±7.3* | *±6.4* | *±0.13* | *±0.07* | *±1.64* | *±4.4* |

Fig. 9. performance of SNN and CPU controllers

A basic power measurement was performed indicating that an average of 25mW is consumed by the SNN. However, to set right expectations the power measurements results are not indicative yet on real cases from few reasons:

- CPU HW can include more optimizations then in this setup. Hence any comparison would not be indicative.
- Kapoho Bay does not support power measurements. Hence another Loihi board, Nahuku, was used.
- SNN-PID is a limited scale neuromorphic application. Hence advantages of neuromorphic computing are not apparent yet as with a larger scale neuromorphic system including more cognitive capabilities

Therefore, power measurements were done only on a best-effort basis by adapting and scaling on Intel's Nahuku Board.

### 2.5 Summary

The authors presented an improved version of a SNN-PID controller on Loihi controlling a constrained 1 DoF drone in real time. Redesign of the SNN and improved IO allowed improving the latency and frequency of the controller indicating performance of the SNN-PID is comparable to those of a CPU-PID, paving the way to a more tightly integrated and a wider neuromorphic system encompassing perception, planning and motor control realized and running integrated on the same neuro-chip in the future.

## 2.6 Personal thoughts

**2.6.1 Why PID.** PID is not a high level cognitive task. Implementing PID with SNN running on a neuro-chip, may seem like an overkill or not using the right tool for the right task. Similar question is why choose motor control and not choose as goal a higher level cognitive task. However, after reading the articles the answer seems that there are two reasons for SNN-PID:

- PID: because building knowledge and technology requires starting from basic building blocks
- Motor control: because it is part of the integrated embedded system.

Therefore, PID is an intermediate step, motor control is an integral part of robotic system. This together answers both questions. The motivation remains to build a larger neuronal architecture covering several cognitive capabilities running on the same neuromorphic hardware and by that to fully exploit the benefits of neuromorphic computing.

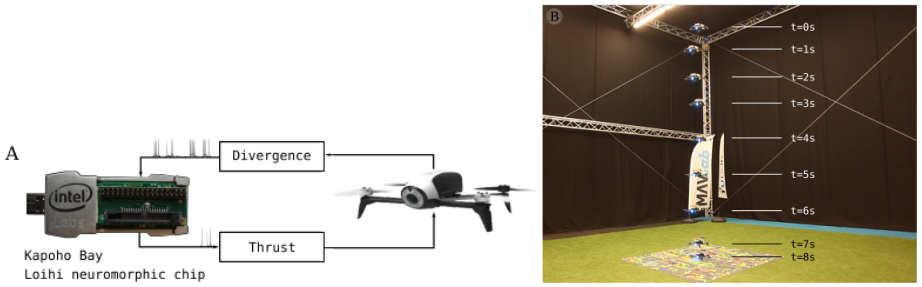
**2.6.2 A very big hidden challenge.** Integration is critical. The integration of neuromorphic computing with conventional computing in the context of robotic environment, is far from being trivial. The integration brings many engineering challenges that must be addressed or otherwise any neuromorphic solution to a highly cognitive challenge cannot not be fully realized. In my opinion an evidence of the importance of the integration challenge can be seen when comparing the two plots from the previous work [6] with the current work [8]. When examining the result of the previous work it shows that a non-optimized rudimentary integration will lead any real world drone to a very flaky and unstable behavior. This in my opinion emphasizes the critical challenge of integration between the neuromorphic and conventional worlds.

**2.6.3 Thoughts about on-chip learning.** Learning is greatly missed. Key benefit of neuromorphic computing is plasticity enabling continuous learning in real-time, which was not applied in this work. The authors indicate that future work will aim at real time PID tuning in an on-chip learning process. To enable such future adaptive learning of the PID gains, preparatory work of converting Sparse-Coding to Rate-Coding for scaling with PID gains and back to Place-Code, was done in the output layer. The authors did realize learning in one of their earlier related articles [3] on motor control with a mobile ground robot using another neuro-chip, ROLLS. In my opinion the plasticity part is a major aspect of neuromorphic computing which is not part of the work, but it is understandably missing when considering the focus on the major integration challenges.

**2.6.4 Inspiration from ground robotics.** The Integration module was taken and inspired from ground robotics. Proving a direct link between different robotic areas applies also in neuromorphic computing. Showing actual transition of knowledge and implementations between the two areas. It is important that neuromorphic solutions can be shared and reused between areas, that knowledge can flow and inspiration from one area can drive progress in another.

**2.6.5 SNN can contain lots of know-how tricks.** There are many tricks in the SNN implementation. These techniques are not trivial and require advanced knowledge, expertise and know-how. Additionally, it is interesting to see how these techniques e.g., Place-Code, neurons with ordered indices in a population, are realized with different neuronal engineering framework.

## 3 NEUROMORPHIC CONTROL FOR OPTIC-FLOW-BASED LANDINGS OF MAVS USING THE LOIHI PROCESSOR




Two major challenges in neuromorphic computing to achieve autonomy, a complex challenge in itself, are the reality-gap between SNN simulation and the neuromorphic hardware; and the related reality-gap between simulation and real world. In this work [7] the authors embed intel's Loihi neuro-chip on a drone to control autonomous landing using a downward looking camera and a simple and efficient strategy called Optic Flow Divergence, used by flying insects. The SNN computes the thrust command based on the divergence of the ventral optic flow field.
The methodology presented in the article includes progressing in steps starting from training the SNN in a simulated environment, adjusting for differences between simulation and the real neuro-chip, testing and measuring intermediate performance on an offline neuro-chip, then finally deploying and testing on a real drone in a real test environment.
The reality-gap between simulation and the real world is bridged using the evolutionary training approach. The SNN is trained to optimize its weights and neuron's parameters using an evolutionary algorithm in a randomized simulated environment before being deployed in a real drone in a real test environment.
To summarize the article we will first give background on Optic Flow Divergence and Evolutionary Algorithms, then elaborate on the article's motivation and goals, after which we will proceed to discuss the article's methods, results and conclusions.

### 3.1 Background on Optic Flow Divergence

Orchestrating a safe landing is one of the greatest challenges for drones and for flying insects. For smooth landing it is essential to control deceleration, to ensure flight speed decreases to a value close to zero at touchdown [9] [10]. It is possible to compute such landing by measuring speed and distance to target in order to reduce speed, yet this approach is computationally expensive.
Alternatively, insects perceive motion through light-sensitive cells and networks of interconnected neurons that react to brightness changes in the scene, resulting in sparse and asynchronous spikes. Honey bees for example, rely on patterns of visual motion and use Optic Flow Divergence to relate the body's own movements to those of the environment. When performing a landing on a horizontal surface, objects at different distances move at different speeds on the retina of the eye, and the bee holds constant the divergence of Optic Flow. It ensures a smooth approach by maintaining a constant rate of expansion of the Optical Flow field allowing it to land on difficult surfaces such as flower leaves in great accuracy, consuming very little energy.

This surprisingly elegant strategy for landings does not require measuring distance to the surface or speed of approach. It only requires tracking the rate of expansion of the image of the surface and ensures a safe landing and a gentle touchdown. Importantly, this strategy is energy efficient and appropriate for their simple nervous systems.

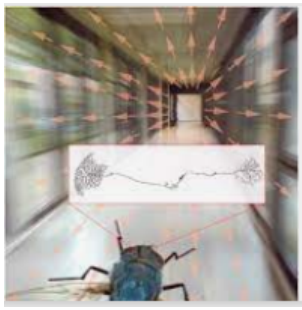


Fig. 10. Optic flow field for a fly flying straight ahead along a corridor

To estimate divergence for landing a drone with a camera directed over a static planar surface we can use the relative temporal variation in the distance between tracked image points. In the image below can see that when descending to land on the surface, image points move outwards. The current point in green moved outwards relative to the previous tracked point in orange:

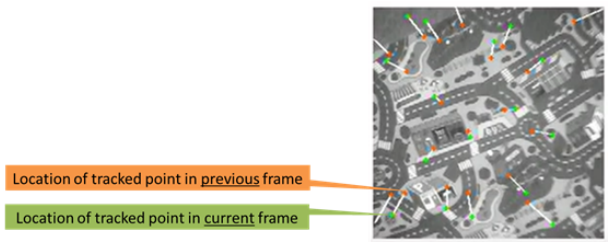



Fig. 11. relative motion of tracked points in optic flow

Divergence corresponds to the ratio of vertical velocity and height above the surface: D=$\frac{V}{h}$ When averaging over many point pairs we get an efficient and reliable estimate called Size Divergence:

$$\hat{D}(t) = \frac{1}{N_D} \sum_{i=1}^{N_D} \frac{1}{\Delta t} \frac{l_i(t) - l_i(t - \Delta t)}{l_i(t - \Delta t)}$$

Fig. 12. size divergence

In the image below can see the overview of the system in the article. A drone with downward-facing camera is performing autonomous landings based on Divergence of the Optical Flow field. As the drone descends, its field-of-view narrows on the center of the landing surface, and distances between tracked points increase proportionally to D.

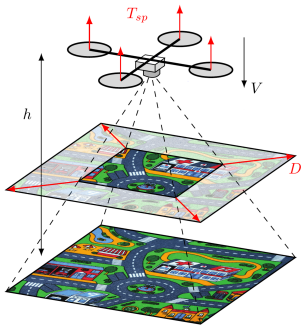



Fig. 13. neuro-chip regulates thrust based on the divergence of optical flow field

### 3.2 Background about evolutionary training

Evolutionary algorithms are population-based, stochastic search methods inspired by the principles of Darwinian evolution [11]. Candidate solutions to the problem play the role of individuals in a population, and a fitness function determines the score of the solutions. Mutations and crossovers between generations introduce new candidate solutions to the pool and advance towards evolved optimized solutions. A simplistic example illustrating an evolutionary algorithm:

generate initial population of individuals
**repeat** until termination
evaluate fitness of each individual
select the fittest individuals
breed new individuals for next generations
**until**

Evolutionary algorithms can be used in several ways. They can be an alternative, can complement other commonly used learning algorithms. In neural networks the characteristics of the networks can be evolved according to a performance criterion. Also, evolution can be coupled with Hebbian learning.
In their previous work [8] the authors used evolutionary training of an SNN to compute the drone's thrust command based on the divergence of the ventral optic flow field to perform autonomous landing with the following evolutionary algorithm: evolution starts with randomly initialized population of individuals. Individuals are tested in a stochastic simulation environment including noise, jitter and domain randomization e.g., wind, through several parameterized scenarios. Their fitness is evaluated by time to land, final height, final vertical velocity, total spike rate, with extra punishment for those who fail to land. Selection is carried out, weights and neuron's parameters are mutated for next generation offsprings. At all time, a hall of fame is maintained, which holds the pan-generational Pareto front. Hall of fame individuals are tested in additional landings and best performing individuals go for real-world tests. See below the evolutionary training algorithm in more details:

Evolution starts with 4 randomly initialized populations of 100 individuals
**repeat**
each individual tested in 4 landings in randomized evironment

individuals fitness evaluated by time to land, final height, final vertical velocity, total spike rate
selection of fittest individuals
mutation of weights and neuron's parameters for offsprings
**until** 400 generations
hall-of-fame individuals tested on 250 landing
best performing hall-of-fame individuals go for real-world tests

The final selected individual was tested on a real drone in a real test environment and compared with the performance of other controllers - ANN based controller and proportional controller. Overall, in this previous work the authors demonstrated that an SNN evolved in a simulated environment was capable of controlling real-world landings while keeping network spike rate minimal.

### 3.3 Motivation and goals

After providing background about Optic Flow Divergence and Evolutionary training, let us elaborate on the motivation and goals of this article. As stated earlier, two major challenges are two most important barriers towards neuromorphic robotics and specifically in drone:

- the hardware integration in a reliable real-time framework
- the reality gap between SNN simulation and neuromorphic hardware

The first challenge was discussed in details in 2. The second challenge is the motivation for this article - bridging the gap between SNN simulation and SNN on a neuro-chip, and bridging the gap between simulation and real-world.
As described in 3.2, the authors already successfully transferred an SNN from a highly abstracted simulation to the real world, performing fast and safe landings while keeping network spike rate minimal. However, while the previous work scope was transition of SNN from simulation to a real drone in a real world environment, the scope did not include transition from simulation to a real neuro-chip. In their previous work the SNN was implemented with TinySNN (C implementation) and deployed on a conventional Cortex-A processor. Not on a neuro-chip.
A secondary goal is to keep the spike rate low: in the previous work the SNN was trained and tested with either 20, 1, or 0 neurons in the hidden layer. The hidden layer of the real-world tested SNN consisted of only a single spiking neuron equally capable of smooth landings while using a fraction of spike rate. Coupled with landing performance, the spike rate was also measured with motivation to minimize required energy consumption.
In this work the goal was to take another extra step and bridge the gap between simulation and an actual neuro-chip on a drone. Doing so while keeping low spike rate.

### 3.4 Methods

**3.4.1 the SNN neuron model, layers and evolutionary training.** In the simulated environment the SNN was realized using PySNN based on PyTorch using the classical LIF model with the following formula 14:

$$v_i(t) = \tau_{v_i} \cdot v_i(t - \Delta t) + u_i(t)$$
$$u_i(t) = \sum_j w_{ij} s_j(t)$$

Fig. 14. simulated LIF model

At each time step input currents are multiplied by their weights and summed and the membrane potential decays by a factor. A spike is emitted if the membrane potential is larger then the threshold. In the previous work the authors tested several configurations of the SNN topology. In this work the SNN had three layers - 20, 10 and 5 LIF neurons for the input, inner and output layers respectively. A total of 35 LIF neurons:

- The input layer has 20 neurons using Place-Coding to represent the Optic Flow Divergence
- The hidden layer has 10 neurons which seems like a balanced compromise between smoothness of landing and keeping spike rate low (in previous work hidden layer was tested with 20, 1 or 0 neurons)
- The output layer has 5 neuron whose spikes are decoded to thrust setpoint

The mutated network parameters in the evolutionary algorithm are $\omega, \theta, \delta, \alpha, \tau$. See below image of the SNN layers:

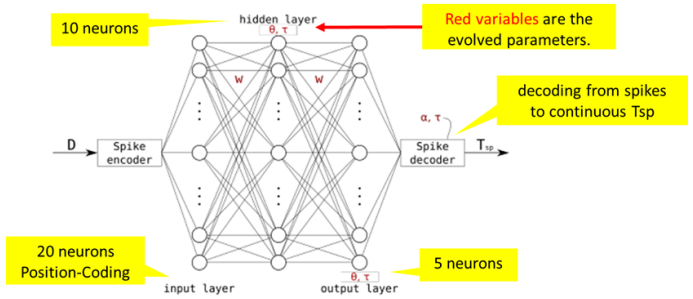



Fig. 15. trained SNN with three layers

Conversion of the output layer spikes to Thrust Setpoint is the weighted average of a thrust vector q and spike trace Xi(t) calculated using the formula in 16:

$$\mathbf{q} = \{q : q = -0.4 + \tfrac{1}{5}n, n \in \{0 \,..\, 4\}\}$$

$$X_i(t) = \tau_{x_i} \cdot X_i(t - \Delta t) + \alpha_{x_i} \cdot s_i(t)$$

$$T_{sp}(t) = \frac{\sum_i q_i \cdot X_i(t)}{\sum_i X_i(t)}$$

Fig. 16. conversion formula from spikes to thrust setpoint

The evolutionary training in this work contained few changes and adjustments to the evolutionary algorithm in the previous work 3.2 for better fitting i.e., number of simulated landings and starting altitude, varying sources of noise and delays, number of populations and number of individuals in each. Fitness of each individual is calculated from accumulated divergence error formular 17:

$$\sum_t |\hat{D}(t) - D_{sp}|$$

Fig. 17. accumulated divergence error

**3.4.2 HW and setup.** Let us now describe the drone setup and its integration with Loihi. The drone, a Parrot Bebop 2, is flashed with a custom version of the Paparazzi autopilot FC. The Optitrack motion capture system provides the FC with position data to control the X and Y axis motion. Divergence was estimated on the conventional processor based on the camera and IMU. See figure 18.

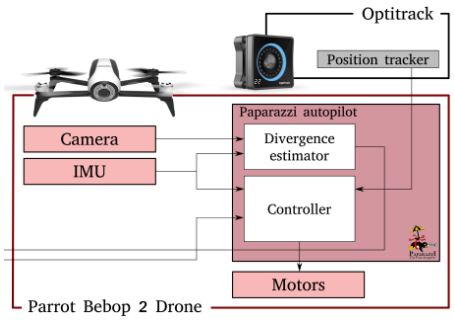



Fig. 18. drone, optitrack and flight controller

Figure 19 illustrates the integration of Loihi and the drone. A Kapoho Bay device containing conventional X86 processors for spike management and two Loihi neuro-chips was embedded on the drone. A $UP^2$ board with a conventional processor running Linux and ROS was interfacing between the Kapoho Bay device and the drone. Spike encoding and decoding was performed in the ROS environment on the $UP^2$ board and exchanged with Kapoho Bay. The divergence and its derivative are calculated by a conventional processor and fed as input encoded to generated spikes transmitted to the Kapoho Bay X86 processors, from which the spikes were sent to the Loihi neuro-cores to be processed by the SNN. The SNN output spikes were sent to the $UP^2$ board in the same way, to regulate the thrust setpoint and to compute the thrust command in the ROS environment.

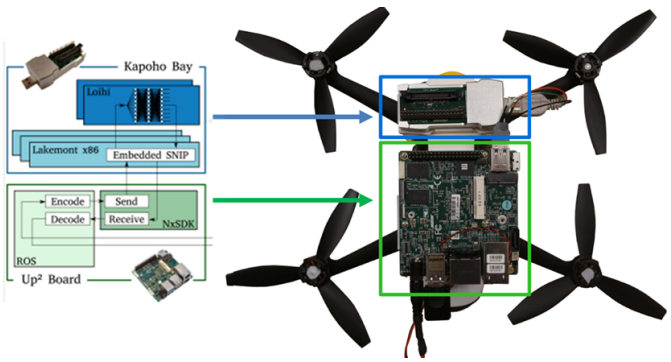



Fig. 19. Loihi and $UP^2$ in drone setup

Divergence error and thrust setpoint were exchanged between the $UP^2$ board and the drone using UART protocol.

**3.4.3 From simulation to on-chip SNN.** Comparing the simulation LIF model equations (see 14) with the Loihi LIF model equations (see 20) reveals differences in the neuronal model that need to be bridged in the transition from simulation to a neuro-chip. Some of these differences are synaptic current decay, membrane voltage bias and subtle differences in behavior calculations such as resolution. Overall, the neurons implemented on Loihi follow a variation of the CUBA LIF model (detailed explanation in section 2.1 of Loihi spec [6]).

$$v_i(t) = \tau_{v_i} \cdot v_i(t-1) + u_i(t) + b_i$$
$$u_i(t) = \tau_{u_i} \cdot u_i(t-1) + 2^{6+\beta} \cdot \sum_j w_{ij} s_j(t)$$

Fig. 20. Loihi CUBA LIF model

Additionally, in Loihi each neurons layer is only updated once per forward pass which is not the case in the simulations environment. Several adaptations were applied in the simulation environment to close the gaps and to closely mimic the Loihi neurons such as limiting firing threshold, enforcing integer resolution, delay buffers between layers.
In the next section 3.5 a methodology to bridge the other related reality-gap, between simulation and real-world, is also discussed.

## 3.5 Results

**3.5.1 Pre-flight simulations.** The selected Hall-of-Fame individual was put through a set of 100 pre-flight simulated landings from different heights in randomized simulated environments with an offline Loihi device. Recordings were taken from the simulated SNN on PySNN of the divergence error inputs, output thrust setpoint and the spikes sequences in both hidden and output layers. These recordings were used to inject the same inputs to the SNN on Loihi and comparing the recorded results with two metrics, infill percentage and matching score (21).

$$\text{Infill} = \frac{\sum_{t=0}^{T-1} \sum_{i=0}^{N_n-1} s_i(t)}{T \cdot N_n} \qquad \text{Match} = 1 - \frac{\sum_{t=0}^{T-1} \sum_{i=0}^{N_n-1} |s_i^{PySNN}(t) - s_i^{Loihi}(t)|}{T \cdot N_n}$$

Fig. 21. InFill and Matching scores

Very close matching score results between PySNN and Loihi were observed in both hidden and output layer, and InFill results were also found statistically not different.
This intermediate pre-flight testing allowed evaluation of bridging the reality gap between simulated SNN on PySNN and SNN on Loihi, before transition to real-world testing. The comparison results (22) showed there are no significant behavior differences between the simulated SNN on PySNN and SNN on Loihi indicating the reality gap can be bridged using the evolutionary approach.

| | **Hidden layer** ($N = 100$) | | | **Output layer** ($N = 100$) | | |
|---|---|---|---|---|---|---|
| | PySNN | Loihi | *Match* | PySNN | Loihi | *Match* |
| *mean* | 6.0% | 6.0% | 99.8% | 10.7% | 10.6% | 99.7% |
| *med.* | 6.0% | 6.0% | 99.9% | 10.6% | 10.5% | 99.9% |
| *s.d.* | 0.5% | 0.5% | 0.3% | 1.1% | 1.1% | 0.7% |
| *min* | 5.0% | 5.0% | 97.9% | 8.3% | 8.3% | 95.7% |
| *max* | 7.5% | 7.5% | 100.0% | 13.8% | 13.8% | 100.0% |

Fig. 22. comparison of InFill and Matching scores

An additional quantitative comparison of the resulting Thrust Setpoint sequences proved equivalence between the PySNN simulation and the Loihi implementation.

**3.5.2 Real landing in real environment.** Following the pre-flight simulations, 20 real world landings were performed with the selected individual on Loihi integrated with the real drone in real test environment. All landings completed successfully and smoothly within 6-8 seconds. As before, input, output and spikes were recorded to compare. Other then few subtle differences e.g., timing of a braking thrust, once again the results compared well without any significant differences.
Video clips of the landing experiments can be viewed on https://youtu.be/LGkjlG-OjaU.

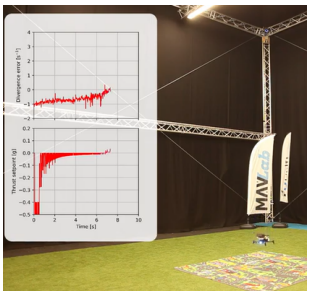


Fig. 23. experiments with Loihi in real test environment

### 3.6 Summary

An SNN with only 3 layers and 35 neurons for autonomous landing using Optic Flow Divergence, was trained using an evolutionary algorithm in an abstracted simulation environment after which a selected individual was tested in a real drone in a real test environment. The real-world landing tests demonstrated robust and consistently successful landings. According to the authors these results indicate the reality gap between simulated environments and real-world with on-chip SNNs can be achieved using an evolutionary approach.

### 3.7 Personal thoughts

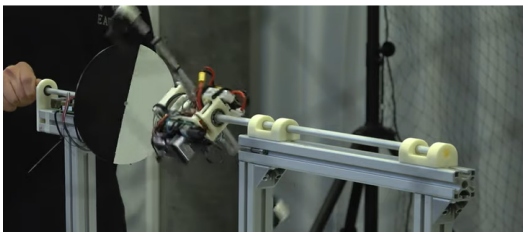
This interesting article brings food for thought in several directions. One direction, about online learning, was the most interesting to me personally.

**3.7.1 Personal thoughts about online learning in this article.** The primary topic of this article is the challenge of reality gap between simulation and the real world. The authors present an interesting approach of evolutionary training which is well worth investigating and indeed shows good results in facing the challenge. However, for robotic applications to adapt to stochastic environments and new tasks, and a distinctive property of neuromorphic hardware that enables continual learning in real-time is on-chip plasticity. Specifically for the neuro-chip used in this work, Loihi has support for several features and synaptic plasticity allowing an online programmable synaptic learning process. Therefore, any sort of reference to on-chip online learning could have been extremely interesting.
The article has both direct relevance to online learning and technical feasibility to either test, refer, compare or relate to online learning. However, the work did not use the opportunity neither in the main discussion of the article nor mention online learning as an open question. As a reader it leaves many open questions. Foremost, how would a similar SNN implementation using Loihi's plasticity compare to the evolutionary trained SNN.

Focusing on evolutionary training is still a very interesting approach. While it may seem that not choosing on-chip learning but rather use evolutionary training is a wrong direction or some sort of rebellious act, in my personal opinion it is both very important and interesting to explore such alternative direction. Combining online learning with pre-flight evolutionary training as two complementary mechanism, would have been extremely interesting.

**3.7.2 Other personal thoughts.** ANNs are served well by Von Neumann CPUs with GPUs. SNNs on the other hand are poorly served by these conventional architectures. Comparing this work with SNN on Loihi to the previous work with SNN with TinySNN on a CortexA processor, highlights the need not only for SNN based algorithms but also for the dedicated neuromorphic hardware in order to bring neuromorphic engineering to a whole new paradigm.
Another personal thought is on the importance of having a good simulation and on-target development facilities and tools. A well defined methodology to bridge the reality gap is also very important. Similarly related methodologies and tools have proven extremely valuable in other fields such as mobile and embedded development and more methodologies and tools advancements will undoubtedly be required for progress in the field of neuromorphic computing.
A minor and less important note it that landing was done on a flat surface in a test lab. It is interesting to think about using the SNN for landing on unbalanced surface with disturbances e.g., wind.
Last thought and over ambitious at this stage: in a perfect world it would have been interesting to see this work progressing towards complementing evolutionary training with online learning using Loihi's plasticity, replacing the Optic Flow usage of the Optitrack system with an Event Camera (discussed in the next article), and extending the thrust commands from Z axis only to include also pitch and roll for X and Y axis, so the drone is able to land on more challenging conditions and surfaces.

## 4 EVENT-DRIVEN VISION AND CONTROL FOR UAVS ON A NEUROMORPHIC CHIP



In this article [12] the authors explore and demonstrate how neuromorphic vision algorithm processing a sparse stream of events from an Event-Camera can be implemented with SNN on Intel's Loihi. The authors explore how the integration between neuromorphic vision and neuromorphic motor control can lead to much faster control rates and lower latency in high speed vision-based control. Another neuromorphic controller, an SNN-PID controller [2], is integrated on the same neuromorphic chip to receive the vision events, with an additional capability of on-chip learning using neuronal plasticity.
Let us first discuss what is an Event-Camera and the relevant vision algorithm (Hough Transform) used in this work, then what were the motivation and goals of the authors, what was the design of the neuromorphic controllers, how it was tested and what were the results.

### 4.1 Background about Event Cameras

Event cameras are bio-inspired vision sensors that output asynchronous events encoding per-pixel brightness changes in the image plane. This method of encoding is inspired by the biological vision. Each pixel reports an event with a timestamp, polarity of change and location only when the intensity changes beyond a certain threshold. Event cameras measure intensity changes in sub millisecond latency and high temporal resolution in order of $10\mu s$ with very high-dynamic range capturing both very dark and very bright areas. Event cameras reduce the sensed information from entire image frames to sparse events, making the vision sensing computationally efficient. As such, event cameras do not output redundant data. Unchanged pixels and entire areas do not appear repeatedly as in standard cameras but only when a change occurs, an event at a certain pixel is sent.
Traditional frame-based cameras fail to provide such low perception latency due to limited frame rate, they also suffer from motion blur limiting the speed of movements they can capture. These properties of event cameras compared to standard frame-based camera make them suitable for high-speed applications where traditional cameras would fail - a promising candidate to enable high speed vision-based control and solving problems such as odometry, object detection, feature tracking and SLAM.
There are also challenges involved with neuromorphic-vision-driven control related to state estimation. Classic image processing and algorithms are typically applied to image frames and not directly to event stream. Neither conventional vision algorithms nor Convolutional Neural Networks (CNN) typically used for visual perception are directly applicable to sparse events. Thus, special event-based vision algorithms have been developed to extract task-relevant information from the stream of events. However, accumulating the events comes with the cost of increased latency and motion blur. Another challenge with event cameras is the reduced data in stationary scenes where there is little change in the scene and the the event camera generates smaller number of events, making state estimation harder or not possible.
In figure 24 can see an illustration of the difference between standard cameras and event cameras.

The standard camera outputs a whole image frame every interval, outputs frames in stationary scenes and may suffer from blur in high speed changes. The event camera outputs sparse events in high temporal resolution indicating on polarity changes only (red and blue dots represent positive and negative events respectively) and does not output redundant data but informative pixels or no events at all.

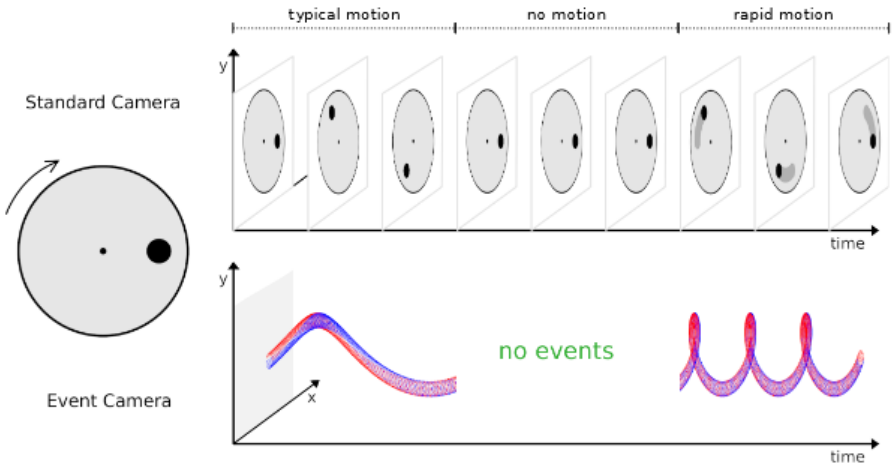



Fig. 24. comparing event camera output to frame camera output

Combining event cameras with standard cameras is a viable option. The frame camera can provide high quality images with high spatial resolution, the event camera can provide changes between frames at high temporal resolutions.
UAVs require fast and efficient perception and control and it has been shown that visual processing with event-based cameras can lead to up to three orders of magnitude faster than conventional processing with image-based vision sensors, while also consuming a few mW of power. For example, it has been demonstrated how an event-based camera can enable tracking of a simple visual pattern with a constrained UAV at an angular velocity of >1000° per second.

**4.1.1 Hough Transform.** Hough transform (HT) [13] is a feature extraction technique used in image analysis, computer vision, and pattern recognition. HT finds imperfect instances of objects or features by a voting procedure carried out in a parameter space, from which elected candidates are obtained as local maxima in an accumulator space.
In detection of a straight line for example, we can show any line in the Cartesian space by two parameters: slope and y-axis intersection. Any line in Cartesian space can be transformed to a point in the Hough parameter space. These parameters are quantized in intervals to create appropriate accumulator bins. For every pixel in Cartesian space, if it satisfies the specific line equation, the corresponding bin accumulator is incremented. Bins exceeding a certain threshold are found as representing straight lines in Cartesian space.
Let us see a Hough transform procedure for line detection in Cartesian space as shown in figure 25-A. Every line L can be uniquely defined by two parameters including the normal distance r from the origin and angle $\theta$ between the normal vector r and x-axis. Assuming $\hat{r} = (\cos\theta, \sin\theta)$, the equation $r = x\cos\theta + y\sin\theta$ transforms every point from Cartesian coordinate to parameter space $(r, \theta)$. In other words, every point in Cartesian space is transformed to a sinusoidal curve in

parameter space defined by the equation.
See figure 25 below for Hough transform procedure of line detection:

- A: lines can be uniquely defined by r and $\theta$
- B: 6 different points on an imperfect line in Cartesian space with parameters r=90 and $\theta$=30°
- C: six sinusoidal curves in parameter space correspond to the 6 points, coincide at (90, 30°)
- D: a bright peak point is visible in parameter space which defines the line characteristic in Cartesian space.

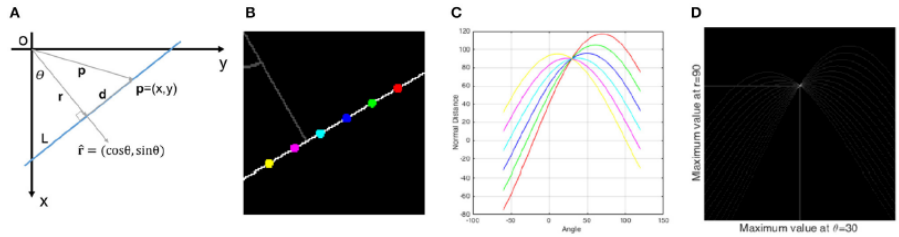



Fig. 25. HT illustration

**4.1.2 Hough Transform and event-camera input.** The previous section described HT in frame-based input. An efficient version of the Hough transform for event-based input was described and used in [14] for State Estimation of roll angle and angular velocity of a fixed drone with respect to a rotating disc. As can see in figure 26 the event-based algorithm tracks a constant number of events N=80 to give an accurate state estimate with little effect from noise. The window size is capped to 3ms to minimize the accumulation of old events. Every 1ms new incoming events update the Hough parameters space and old events are removed.

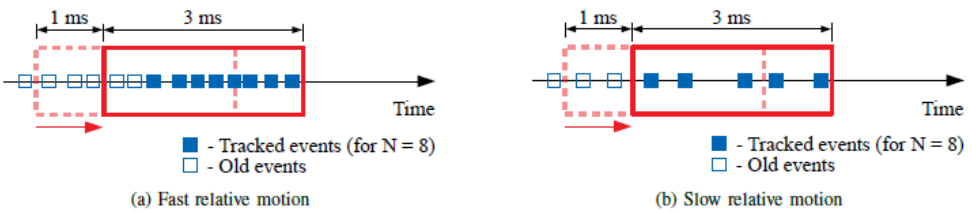



Fig. 26. HT for event-based input

The (r, $\theta$) Hough parameters space accumulator is discretized into bins of size 5° by 5 pixels. The $\theta$ value of the bin with maximum count is eventually considered to be the attitude of the disk with respect to the drone.

### 4.2 Motivation and goals

Drones are very agile and unlike most other mobile robots they can traverse extremely complex environments at high speeds. However, autonomous drones today are far from capabilities of human pilots in terms of speed, versatility and robustness. To date only expert human pilots can fly at extremely high speeds in stochastic environments where even a tiny error can lead to a crash. Speed is particularly important because to fly faster drones need faster sensors and faster algorithms.

The authors explore and demonstrate in this article several interesting capabilities implemented, integrated and combined to achieve fast control rates and low latency in high speed vision-driven drone with neuromorphic control. The first capability is a neuromorphic visual line-tracking task with SNN implementing event-based Hough Transform running on neuromorphic hardware. The inputs for the SNN algorithm are sparse events from an event camera. The second capability is integrating the neuromorphic visual processing with a neuromorphic motor control. Integrating the two capabilities and ensuring it can work in high speed scenarios is an interesting challenge in itself. The third capability is enhancing the neuromorphic motor control with on-chip learning using neuronal plasticity.

Overall, the three capabilities combined should achieve a goal of enabling autonomous high-speed flight with neuronal sensing and control, at low-power with improved latency and processing rate compared to conventional computing.

There are a number of considerable challenges in this work. The event camera should generate events with high enough temporal resolution to track a fast rotating line. Seamless integration between the neuromorphic vision and motor control and the drone's conventional modules should provide fast and accurate inputs to the close-loop controller SNN, which should output robust and low-latency motor commands, allowing the drone to rotate according to the high speed line rotation. Additionally, the behavior should be able to adapt dynamically using on-chip learning.

### 4.3 Methods

The drone's neuromorphic system can be described in terms of the hardware setup and the neuromorphic computing sub-system. The latter is comprised of neuromorphic vision processing SNN and neuromorphic motor control processing SNN, tightly integrated. The motor control is further split into a closed-loop PD controller SNN and an additional adaptive-learning-path SNN using on-chip learning.
Let us now describe first the hardware setup and then the three SNNs deployed on the neuro-chip.

**4.3.1 HW and setup.** At the core of the overall drone setup is Intel's Loihi neuromorphic chip interfaced directly with a neuromorphic event camera and indirectly with the drone through a $UP^2$ board.

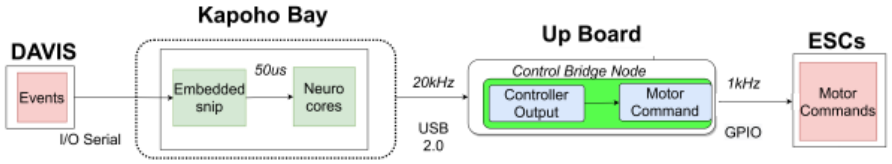



Fig. 27. drone HW setup

A black-and-white disk serves as the reference pattern, with the line between the black and white halves marking the horizon to be tracked. The disk can be rotated by hand or with motor, and the drone is tracking the rotating horizon.
The event camera, DAVIS 240C, sends an event stream directly to a Kapoho Bay device via direct AER interface in the case of the SNN implementation on Loihi (alternatively, in the case of comparison with a CPU event-based algorithm, to a $UP^2$ board ). For ground truth measurement, the drone and disk are both equipped with CUI AMT22 Modular Absolute Encoders, measuring actual roll angle and velocity with accuracy of 0.2° and a precision of 0.1° respectively.

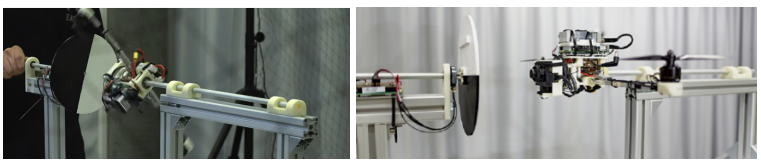


Fig. 28. physical setup of the disc and drone

All neuromorphic computations are hosted on the Loihi neuromorphic cores. The DVS events are input to the neuronal cores of a neuromorphic chip using an address event representation (AER) interface, and processed directly by the visual processing SNN. The motor control SNN computes the required thrust which is sent every 1ms through the $UP^2$ board to the Flight Controller.

**4.3.2 SNN Hough-Transform implementation.** The authors implemented an SNN on Loihi which generates state estimates directly from incoming DVS event stream, as illustrated in figure 29. This SNN is a neuromorphic version of the event-based Hough Transform described in article [13].

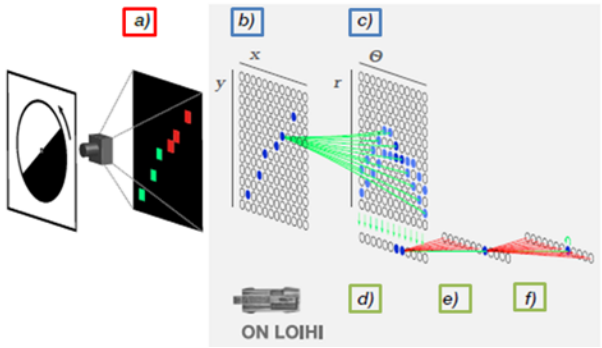



Fig. 29. SNN for visual processing of event camera

The implemented SNN algorithm works as follows:

- In layer (a) the DVS camera event-stream is down-sampled on the x86 co-processor on Loihi. From the X86 CPUs, the DVS events are injected into the neuronal cores in small batches at fixed time-steps
- On the Loihi neuro-cores, neuronal populations (b) and (c) represent events in Cartesian space and in the Hough parameters space, respectively (25). The connections matrix between neurons in population (b) to neurons in population (c) is computed according to the equation r = x cos$\theta$ + y sin$\theta$ in 4.1.1.
  - The continuous values of r and $\theta$ are discretized to bins. A population size of 90 neurons results in a resolution of 2° per spiking neuron in the range of ±90°
  - In order to accumulate events the decay time and threshold of neurons in population (c) are parameterized so that at-least 20 events lie on same line within 3 time-steps to generate a spike
- In neurons population (d) the $\theta$ from the Hough parameters space is read out to a respective "angle" neuron. These "angle" neurons are connected to neurons population (e) for cleaning up activity of several neurons firing simultaneously
- The last of the three angle neurons population (f) acts as a memory for stationary periods in which the disc does not rotate.

The overall SNN visual processing time is 250$\mu$s corresponding to 5 time steps, one per each neurons population layer (b, c, d, e, f).
In short, for visual processing, a neuromorphic version of the Hough-Transform was implemented and deployed on Loihi.

**4.3.3 SNN-PD implementation.** The SNN used to control the drone's position is a shallow version of the simpler and earlier version of the PID controller in [2] which was also used as basis for the article in section 2. Two inputs, angular error and its derivative, are provided to the SNN directly by the event-based visual module. The I term was omitted for simplicity of comparing with a CPU control loop. Improvements were required to ensure the SNN-PID can handle the fast rotation speed of the disc. First, by utilizing more efficient allocation of the neurons populations on the chip, the error and motor command populations size increased from 63 to 361 neurons allowing to achieve more precise control and a larger range of commands. As a reminder, the populations use Place-Coding, therefore by increasing the population size allows representing either finer resolutions or wider ranges of values. Second, by using a more efficient integration with the $UP^2$

board (Direct Memory Access instead of communication protocol), the SNN output was obtained at a three times faster rate.
The state variable of the control loop is the difference in orientation between the drone and the line on the disk. The spikes in the output population (U) encode the difference using Place-Coding, which is converted to thrust in software using, equations in figure 30.

$$\mathbf{u} = \frac{idx}{N} \cdot (T_{max} - T_{min}) + T_{min}$$

A. Thrust difference represented in Place-Coding

$$\mathbf{u} = K_P \cdot \theta + K_D \cdot \dot{\theta}$$
$$\mathbf{T} = c_T \pm (\frac{u}{2} + b_T)$$

B. controller output converted to thrust

Fig. 30. controller output conversion to thrust

Online learning was realised not within the SNN-PD controller but through an adaptation pathway which runs in parallel, described in the next section 4.3.4.
In short, for neuromorphic motor control, a simplified version of the previous PID controller was optimized and tightly integrated allowing it to fit the fast rotation of the disk.

**4.3.4 On-chip learning with Loihi.** On-chip learning on Loihi enables adaption of the neuromorphic controller by using synaptic plasticity rules. The authors have chosen the option of adding an adaptation pathway which runs in parallel to the full SNN PD controller as a feed-forward layer that connects the input population $\theta$ to the PD controller's output population U. The adaptive path monitors the error signal over time and compensates for extended periods of large steady-state errors by increasing or decreasing the adaptive term according to if the monitored accumulated error exceeds a threshold.

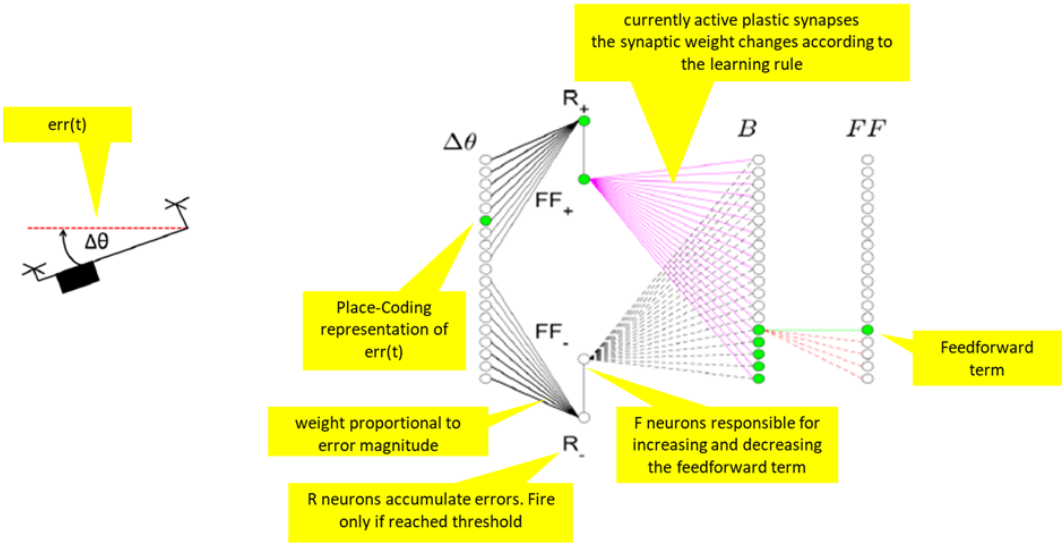



Fig. 31. Adaptive term SNN with on-chip learning

The neuronal population representing the err(t) in the adaptive path is connected to accumulating neurons ±R that fire only when reaching a threshold. Online learning realised by local synaptic

plasticity occurs between the ±FF neurons and the B population where changes to the synaptic weights occur according to the ±R neurons activity and the learning rule in figure 32, to compensate for large steady-state errors.

$$\Delta W = W \pm \delta_R(t)$$

Fig. 32. Adaptive pathway learning rule

### 4.4 Experiments and results

The drone setup was tested in two aspects:

- compare the neuromorphic controller to an event-driven CPU controller in how it performs in a similar extreme control scenario of a tracking the disk at high speed
- quantify the ability of the on-chip plasticity to adapt control parameters dynamically

**4.4.1 Testing high-speed control.** Here the ability of the drone to track the horizon rotating at different speeds was tested and analyzed for performance and with RMSE.

The below figure 33 illustrates how both neuromorphic controller and CPU controller can track the line when the disk with the visual pattern was turned manually at low speed.

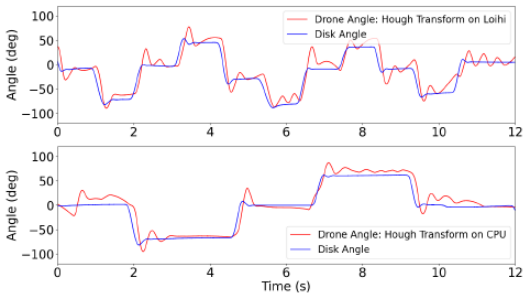



Fig. 33. Line tracking comparison between Loihi and CPU

For comparison in high-speed rotation the RMSE between the reading on the disk and the drone Absolute Encoders was calculated, while using a motor to spin the disk. As expected, as a result of the neuromorphic visual processing running 20-times faster than the CPU processing and allowing more accurate estimate at high rotation speeds, the neuromorphic controller tracked at higher speeds (upto 1200 deg/sec) with a lower RMSE than the CPU controller. See figure 34

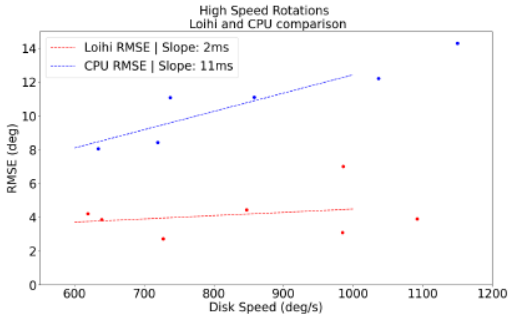



Fig. 34. RMSE when tracking high-speed rotating disc

Video clip of the rotating drone in action can be viewed on https://youtu.be/EcZ_Nq6Z89U.

**4.4.2 Testing on-chip adaptive control.** For exploring and measuring the ability of the on-chip learning to autonomously adjust the controller parameters dynamically, an additional weight of 125g was attached to only one arm at one side of the drone, representing a strong disturbance. (In this experiment the PD controller received inputs from the encoder for ensuring the learning results are isolated from the vision module).
The CPU controller without adaptation shows a persistent steady state error of roughly 20°, with an increasing average error. On the other hand, the neuromorphic controller with the adaptive learning path compensates for the steady state error and maintains same error as without the disturbance.

In figure 35 can see how the non-adaptive CPU controller is affected by the disturbance, while the adaptive neuromorphic controller compensates for the disturbance and maintains a much smaller RMSE.

| Controller | RMSE (deg) | | | | | |
|---|---|---|---|---|---|---|
| *Step* | *+20°* | *-20°* | *+30°* | *-30°* | *+40°* | *-40°* |
| **No Added Weight** | | | | | | |
| Adaptive SNN PD | 6.9 | 12.3 | 8.2 | 13.2 | 11.4 | 13.9 |
| Non-adaptive CPU PD | 6.9 | 7.7 | 8.5 | 9.2 | 9.4 | 11.5 |
| **With Weight** | | | | | | |
| Adaptive SNN PD | 6.9 | 9.9 | 8.6 | 10.8 | 8.5 | 16.0 |
| Non-adaptive CPU PD | 22.3 | 21.9 | 19.6 | 20.3 | 20.6 | 19.7 |

Fig. 35. Adaptive SNN vs CPU controller accuracy with additional weight

In figure 36 can also see how the non-adaptive CPU controller is affected by the disturbance maintaining a constant error from the target angle, while the adaptive neuromorphic controller compensates for the disturbance and corrects the disturbance.

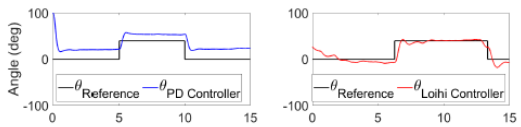



Fig. 36. CPU controller (left) vs Adaptive SNN (right) performance with additional weight

## 4.5 Summary

The authors have shown how neuromorphic vision control can be implemented on Intel's Loihi integrated with an event-based vision sensor. The neuromorphic vision-control was integrated with a neuromorphic motor-control, on the same neuro-chip. Additionally, on-chip learning on Intel's Loihi was tested in compensating for external disturbances. The controllers leverage the benefits of neuromorphic hardware and outperform a state-of-the-art conventional CPU based high-speed event-driven controller.

## 4.6 Personal thoughts

In my personal opinion this article is the most interesting and challenging among the four articles in this work. It is a step forward in atleast three aspects. First, as described in summarized article 2 there are many engineering challenges and bottlenecks that must be addressed. This work uses tight

integration for communication between the neuro-chip and conventional CPU, efficiently deploying separate neuromorphic modules on the same neuro-chip, using AER for integrating neuromorphic modules. These are non-trivial optimizations yielding significant improvements. Second, the scope of this work is wider then a single neuromorphic module or building an isolated neuromorphic building block. The combination of both neuromorphic vision-control and neuromorphic motor-control is a step forward towards combining multiple neuromorphic modules on the same neural network or single neuro-chip, opening possibilities for more advanced applications combining together multiple cognitive capabilities of vision, control, perception and planning; leading to a highly autonomous intelligent system with AI on the edge. Lastly, the authors took the opportunity to explore also on-chip learning which is a distinct feature and benefit of neuromorphic computing augmenting the previous list of cognitive capabilities with online learning in contrast to offline training required by conventional ANN.

A possible step to progress this work forward could be a real world scenario as in summarized article 3. A non-constrained drone performing a task in a non-constrained real environment with dynamic disturbances.

# 5 DEEP NEUROMORPHIC CONTROLLER WITH DYNAMIC TOPOLOGY FOR AERIAL ROBOTS

In this article [15] the authors present a neuromorphic controller which is dynamically changing its weights and topology to handle switching between different tasks with uncertainties in environmental conditions. Let us first discuss what are the motivation and needs for controller with network topology adaptiveness and what are the study goals. Then discuss what are the adaptivity mechanisms and design of the controller, the setup of the drone, the tested scenarios and experiments carried.

## 5.1 Motivation and goals

Drones are required to be increasingly adaptive and to perform variety of operations in changing conditions to complete diverse missions. There are many possibilities for such variety of tasks and changes in conditions. An aerial robot may need to operate and switch between tasks arriving in sequence with conditions that may not be anticipated. Different missions may require the robot to conduct different set of tasks . A drone may perform a fixed routine sequence of tasks but with significant different environmental conditions that may change even during flight.
An autonomous drone built with a fixed model or a model with fixed computing capacity may fail either due to tasks switching or when environmental conditions change. Conventional learning approach could handle variations when the system is trained for similar tasks in a representative environment, however, conventional learning may result in overfitting leading to degradation or failure. There is a need for a learning approach that is neither fixed for particular tasks nor requires pre-flight tuning of the network.
The goals set by the authors are to formulate a flexible learning approach with flexible learning scheme with a set of mathematically sound laws and mechanics, operating within constrained computation limits and providing stability for the aerial robot. Therefore, the article aims to design a neuromorphic controller which is able to morph and adapt to new tasks, work in different operating conditions, without any pre-flight training or manual parameter tuning.
Two scenarios are considered for the scope - a circular trajectory with disturbances and altitude hovering with surface proximity influence. The goals are to be achieved using an online self reconstructed controller with adjustable weights, self adjustment of network width and expansion of network depth. Number of nodes in each layer may change and number of hidden of layers may change so that the drone is able to handle different tasks with different disturbances and noise without training from scratch, within computing limits, while maintaining flight and operation stability.

## 5.2 Methods

To formalize the dynamics and challenges of the drone operations, some assumptions, modeling and control aspects are first described.
The base assumptions for the control system is that a lower-level controller such as a PID controller in the inner-loop is handling commands for all three axis. The bespoke neuromorphic controller will be at the outer-loop for for improving the system stability, performance and adaptive to the dynamics.
The equations in figure 37 describe the nonlinear system dynamics along x-axis in the mathematical model for the control design. x(t) is the horizontal position of the drone, vx(t) is velocity, ux(t) is control input, dx(t) is the unknown disturbances, function f() is representing the acceleration due to disturbances and uncertainties.

$$\dot{x}(t) = v_x(t)$$
$$\ddot{x}(t) = \dot{v}_x(t) = u_x(t) + f(v_x(t), d_x(t))$$

Fig. 37. model of system dynamics along x-axis

The proposed controller takes the form of a Sliding Mode Controller (SMC) to make the model mathematically sound and keep the system stable. Ux is the control input, Us is a switching function, Unn is the output from the neuromorphic controller.

$$u_x = u_{\mathrm{s}} + u_{\mathrm{nn}}$$

Fig. 38. controller modeled with Sliding Mode Controller method

Figure 39 shows the controller inputs and output. The neuromorphic controller takes four inputs: The reference signal, and the plant error in the last three steps. The output is the control signal for the dedicated axis.

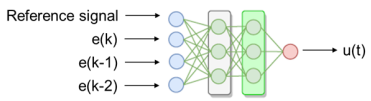



Fig. 39. neuromorphic controller inputs and output

The neuromorphic controller has the following three main features:

- adaptation of network weights
- growing and pruning of nodes within a layer
- expansion of the network with a new hidden layer

Let us now discuss the online reconstruction of the network allowing it to change its capacity in handling different tasks and unexpected disturbances.

**5.2.1 Online network reconstruction.** The weights of the network are updated based on minimizing the difference between Unn (controller output) and Ueq (estimated equivalent control) using the quadratic function in figure 40. The ability to update the network weights dynamically allows the neuromorphic controller to keep ongoing system stability.

$$J = \frac{1}{2}(u_{\mathrm{eq}} - u_{\mathrm{nn}})^2$$

Fig. 40. procedure for network weights update rule

As changes may be more drastic then those handled by weights adjusting only, the neuromorphic controller features an elastic network topology with dynamically adjustable number of nodes and hidden layers. The dynamic topology is self-organized on the fly while engaging in the control action.
For the adaptation, Bias and Variation are estimated in a time window. The node growing strategy is based on High Bias condition identified by the modified Statistical Process Control (SPC) approach.

Node growing is triggered in the high bias condition indicating on underfitting conditions when equation A in figure 41 is satisfied. Node growing triggering will add a new node into the network. Hidden node pruning is triggered if high variance condition is identified which indicates on overfitting conditions when equation B in figure 41 is satisfied. Node pruning triggering will results in removal of a node. The pruned node is the node with the lowest statistical contribution as per equation C in figure 41. Both adding and pruning of nodes occurs frequently or sparsly according to confidence levels estimation.

$$\mu_\beta^N + \sigma_\beta^N \geq \mu_\beta^{\min} + \pi\sigma_\beta^{\min} \qquad \mu_V^N + \sigma_V^N \geq \mu_V^{\min} + 2\chi\sigma_V^{\min} \qquad NC_i = ||W_i||_2$$

A. High Bias condition B. High Variance condition C. Node contribution

Fig. 41. adaptation conditions

As changes may be more drastic then these handled by weights adjusting only or even by node adding or pruning, the neuromorphic controller can also add a whole new layer to enhance the network capacity significantly. The controller uses the Concept Drift Detection Method to identify highly significant changes in dynamics of the drone, environmental disturbances or a new task. All of which require the network capacity to change significantly. Figure 42 shows the conditions for adding a whole layer when the mean bias gradient is increasing under saturated performance of the network. The closest hidden layer to the output layer is then copied and added with re-initialized parameters.

$$\nabla_\beta^E = \frac{\mu_\beta^E - \mu_\beta^{E-1}}{\Delta t} \qquad |\nabla_\beta^E - \nabla_\beta^{min}| < \frac{R}{\mu_V^E}\sqrt{\frac{1}{2E}\ln\frac{1}{\delta}}$$

A. Gradient mean bias B. Saturated performance

Fig. 42. Layer expansion conditions

Let us now explore how these mechanisms allow topology adaptiveness. Figure 43 illustrates a sequence of changes in the network. Initially, the network is created with a single hidden layer and single node. The inputs to the network are the reference signal and the last 3 steps error. In the second step, high-bias condition is identified by the condition in equation 41-A, and a new node is added to the hidden layer. Following in the third step, the network is expended with a new layer when significant change in the dynamics is identified by the condition in 42. This is followed by another node added in the fourth step after which high-bias is identified and pruning is activated in the fifth step when high-variance condition is satisfied.

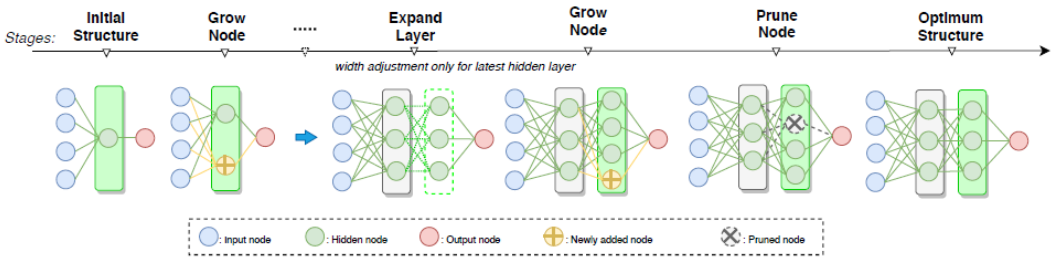



Fig. 43. examples flow of topology adaptiveness

### 5.3 Experiments and results

The scenarios considered in this work are:

- tracking a circular trajectory with disturbance like wind gusts
- tracking altitude with interaction effects generated due to surfaces proximity

To measure and evaluate the dynamic controller, two environments were used for testing:

- Gazebo simulation of tracking a circular trajectory
- Real-world tests of circular trajectory with wind gust disturbances.

(NOTE: the results for surface proximity were not included in the published article)

**5.3.1 Simulation results.** The Gazebo simulation starts with a first phase of hovering for 200s while the system learns initial parameters that remain to be used as the baseline for the next phase. In the second phase, two network configurations are tested:

- Fixed network with a fixed capacity in which the network does not change its topology
- Dynamic network configuration able to change its topology according to the rules described in 5.2.1.

The circular trajectory is defined with two different velocities. Slow trajectory in the first half of the circle, followed by a faster trajectory in the second half.
Figure 44 illustrates how the fixed capacity network configuration could not keep tracking the faster trajectory while the dynamic neuromorphic controller successfully tracked both slow and fast trajectories.

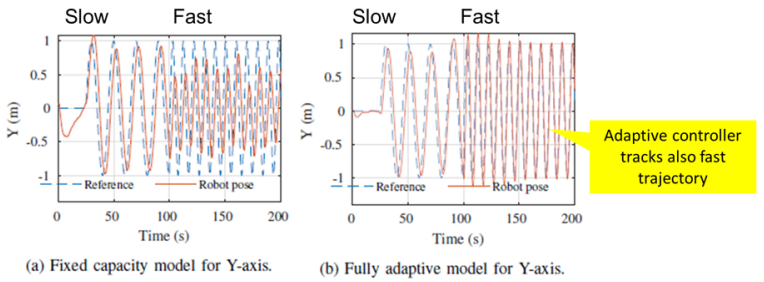
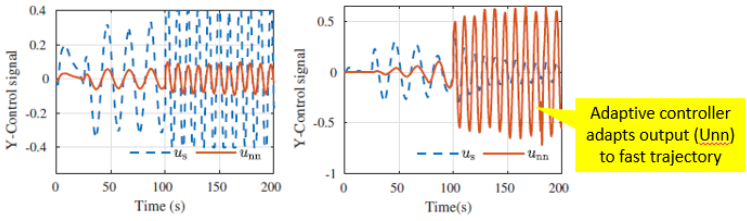



Fig. 44. tracking of slow and fast circular trajectories

In the next figure 45 can see the respective values of the switching function and network output. In the adaptive controller on the right side can see how the network output changes, which explains how the dynamic controller was able to track both low and fast trajectories illustrated in previous figure 44

Figure 46 illustrates how the dynamic controller mechanisms contributed to the network update change. The dynamic controller added and removed nodes in the hidden layer in order to adapt the networks capacity for handling the trajectories changes. Worth noting also that layer addition was not required to be used since the changes were not significant to be at a level of concept drift.

**5.3.2 Real-world testing results.** A set of real-world experiments were defined to test the two scenario. A circular trajectory with wind gust rejection was simulated using an industrial fan, shown in figure 47-A. Altitude tracking with interaction effects due to surfaces proximity was simulated using a transparent ceiling, shown in 47-B.
(reminder: scenario B results are not published in the article)




(a) Control signals of the fixed capacity (b) Control signals of the fully adaptive

Fig. 45. Control signals of the adaptive model

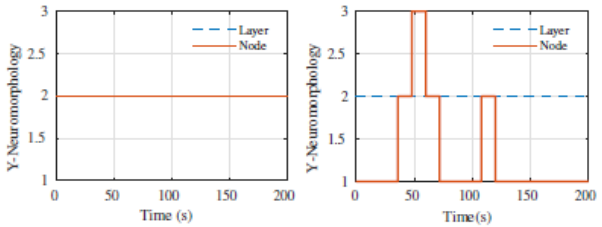



Fig. 46. adding of nodes in the hidden layer

The drone setup for the real-world was using conventional CPUs for running the Flight Controller, inner-loop PID controller and the outer-loop dynamic controller implemented with PyTorch.

Fig. 47. Scenario A - circular with wind, Scenario B - Surface interaction

The experimental results are given in figure 48 with statistical evaluations showing that also in the real world the fixed-capacity controller could not handle the variations and disturbances and its performance deteriorated as the tests progressed. On other hand, the dynamic controller was able to adapt to the disturbances and reduced the error.

In figure 49 can see the reason the dynamic controller was able to adapt to the disturbances. in two separate tests of the circular trajectory with wind gust the network identifies a significant change in the dynamics and applies layer expansion. The added layer increases the network capacity and the drone is successful in handling the wind gust disturbance.

Video clip of the real world experiments can be viewed on https://youtu.be/rWHoKYJBgBQ.

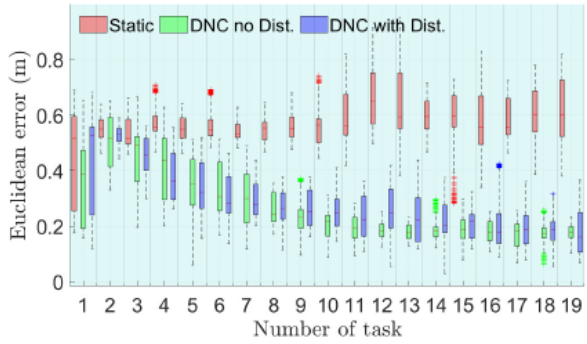



Fig. 48. circular trajectory tracking statistics

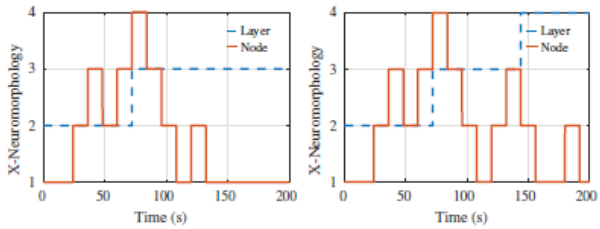



Fig. 49. layer expansion in two real world experiments

### 5.4 Summary

The article presents an approach for a neuromorphic controller with dynamic topology allowing it to adapt to different tasks, variations and uncertainties in the system dynamics and environmental conditions. The approach was evaluated using simulation and real-world tests indicating that by using weights adjustments, adding or removing nodes and adding layers the network was able to adapt to changing conditions. The results were compared to a fixed network model which was not able to adapt to the changes and therefore suffered from lower performance.

### 5.5 Personal thoughts

The primary personal thought about this article is about how the network topology concept could be applied to neuromorphic algorithms and neuromorphic hardware. The article scope is the theoretical development of network topology adaptiveness. The technical work in this article did not use specifically SNN, biological neuron model and neuromorphic computing. Nevertheless, apart from the direct relevance of this article to research in the field of autonomous drones with cognitive capabilities, in my opinion the article presents a very interesting approach that should be explored further with SNN running on neuro-chip. Same as the article [7] applied Evolutionary Algorithm for SNN and presented an interesting alternative or complementary method to online learning, it may very well be that the concept of a dynamically changing neuromorphic controller should be explored further to dynamically changing SNNs and address questions such as how would topology adaptiveness compare to Loihi online learning, can it complement and work in synergy and be used selectively for increasing capacity according to situations.
Additionally, same as conventional computing platforms today support dynamically loading and

unloading of code and modules e.g., shared libraries or instruction caches, as the field of neuromorphic computing advances there may become a need to perform operations that allow to replace executable code or perform operations that help to reduce resource consumption. A neuromorphic algorithm implementation that changes its resource utilization ad-hoc by growing or shrinking could be an interesting alternative for dynamic loading, unloading and shifting of executable code.

## 6 PRACTICAL WORK

For the practical part of the seminar I chose to implement a neuromorphic PID controller to regulate how a simulated drone reaches a setpoint altitude. The work was inspired by two of the articles discussed in this work - the one DoF neuromorphic PID controller [2] and the vision and control [4].
The next section describe the motivation and goal, the methods, results and future direction.

### 6.1 Motivation and Goals

The guiding principles and primary motivation were:

- building knowledge and technology requires starting from basic building blocks
- should start with a simple building block and aim towards a long term goal
- leveraging the NBE course completed last semester, for the practical work

Several simulators, frameworks and implementation goals were considered each having pros, cons and constraints. Eventually I chose to use Nengo which we learnt in the NBE course to follow a similar path led by Yulia Sandamirskaya in the first and third articles: start from a neuromorphic PID controller which is a basic building block for motor control as in her work in article 2, Keeping in mind a longer term goal of vision, motor and online learning on a neuromorphic chip on a real drone as in her work in article 4.

Therefore, the goal of the practical work is to build an SNN-PID controller with Nengo https://www.nengo.ai/ integrated with the popular Microsoft AirSim simulation https://https://microsoft.github.io/AirSim to control the velocity of a drone in order to reach a setpoint altitude.

### 6.2 Methods

The SNN-PID controller is implemented with three Nengo Ensembles (neurons populations), per each P-I-D term. The controller receives as input the error between current altitude and setpoint altitude, and outputs the required velocity. AirSim Python APIs were used for acquiring the drone altitude and for setting the drone flight velocity. The implementation overview is presented in figure 50. Short description of the flow and interaction between the main modules in figure 50:

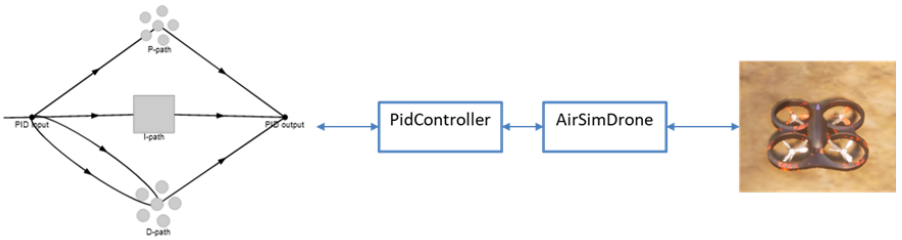



Fig. 50. practical work architecture overview

lift off to start position altitude
**repeat** until termination
    get current altitude from AirSim
    execute Nengo simulator for short period
    get neuromorphic PID controller output

set simulated drone vertical velocity
execute AirSim simulator for short period
**until**

Figure 51 shows the Nengo code implementing the SNN-PID.
The project code is available at https://github.com/roybenhayun/nbe-seminar.

```
# stim, in, out
stim = nengo.Node(get_stimulus) #[error, last_error]
error_input = nengo.Node(size_in=2, label="PID input")
pid_output = nengo.Node(output=node_output_func, size_in=1, size_out=1, label="PID output")
nengo.Connection(stim, error_input)

# P path
p_ensemble = nengo.Ensemble(n_neurons=n_neurons, dimensions=1, radius=radius, label="P-path")
nengo.Connection(error_input[0], p_ensemble, synapse=0.005)
nengo.Connection(p_ensemble, pid_output, transform=Kp, label="Kp")

# I path
integrator = nengo.networks.Integrator(tau_i, n_neurons=n_neurons, dimensions=1, label="I-path")
integrator.ensemble.radius = radius * 3
nengo.Connection(error_input[0], integrator.input)
nengo.Connection(integrator.output, pid_output, transform=Ki, label="Ki", synapse=0.1)

# D path
d_ensemble = nengo.Ensemble(n_neurons=n_neurons, dimensions=1, radius=radius, label="D-path")
nengo.Connection(error_input[0], d_ensemble, transform=1 / AIRSIM_SIMULATION_MS_PERIOD, synapse=0.5)
nengo.Connection(error_input[1], d_ensemble, transform=-1 / AIRSIM_SIMULATION_MS_PERIOD, synapse=0.5)
nengo.Connection(d_ensemble, pid_output, transform=Kd, label="Kd")
```

Fig. 51. Nengo Python implementation of SNN-PID

Tuning of the SNN was done using several mechanisms including changing number of neurons by inspecting tuning curves, adjusting ensemble's radius, scaling SNN input output to reduce ensemble's radius, adjusting synapse values. Figure 52 illustrates tuning curves and variations drop used to tune the P-term ensemble

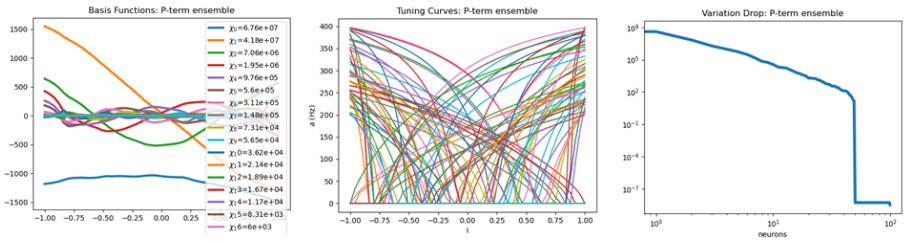



Fig. 52. SNN Tuning (P-term example): Basis Functions, Tuning Curves, Variation Drop

Testing the PID controller was done similar to article 2 by giving it a desired setpoint and comparing its behavior with a CPU based PID. (due to simulation constraints I used altitude setpoint and not angle as in the article). The Nengo-Gui simulator was found very helpful in debugging the neuromorphic model, allowing to inspect separate ensembles outputs in isolation

from the AirSim environment.
Tuning the PID gains of both CPU-PID and SNN-PID was done manually by testing and evaluating PID performance results.

### 6.3 Results

The test case for evaluating the neuromorphic PID controller was reaching a setpoint altitude. After the drone lift off, let it hover for few seconds to stabilize and then give it the setpoint altitude. Different combinations of PID configurations and SNN tuning configurations were evaluated and compared to a reference CPU-PID controller. Key takeaways from the evaluation and comparisons are related to variability, dynamics and comparability.
Variability of the SNN-PId behavior and performance relates to the non-deterministic and imperfect accuracy of neurons output and also to the wider spectrum of additional tunable parameters beyond PID Gains e.g., number of neurons, radius, synapses etc. In figure 55 can see an illustration for a primary reason to variability, the number of neurons in the ensembles. The image shows performance of the same SNN-PID configuration with 50, 150, 250 and 1000 neurons in each ensemble, respectively.

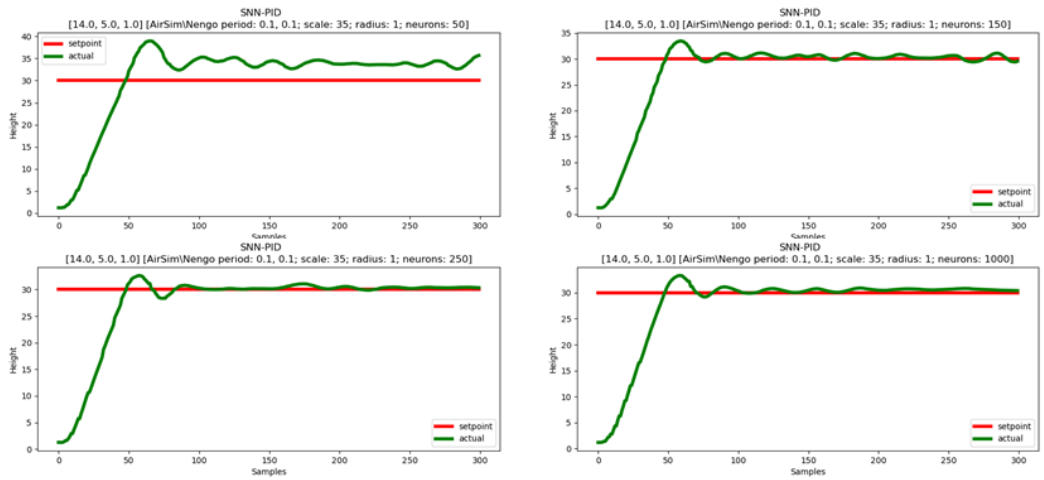



Fig. 53. SNN-PID with 50, 150, 250, 1000 neurons ensembles

As stated, the variability in behavior results also from the wider spectrum of additional tunable parameters beyond PID Gains. Figure 54 illustrates the variability of the same SNN-PID Gains with default synapse values, custom synapse values, wider radius and wider radius for the I-term specifically. Each parameter tuning produces variability of the SNN-PID behavior with the same PID Gains.

When comparing the SNN-PID and CPU-PID the dynamics and comparability are observed. The CPU-PID has a more deterministic and rigid performance while the SNN-PID has softer and non-deterministic dynamics. Overall can see in diagrams 55 that the SNN-PID and CPU-PID are comparable.

Challenges were observed in integration of the neuromorphic modules with the conventional environment, similar to the first three articles. These challenges can be categorized by required improved interfaces, latency issues, SNN runtime duration etc. As an example, in integration of Nengo and AirSim, the two environments can not run concurrently. AirSim has to be paused and

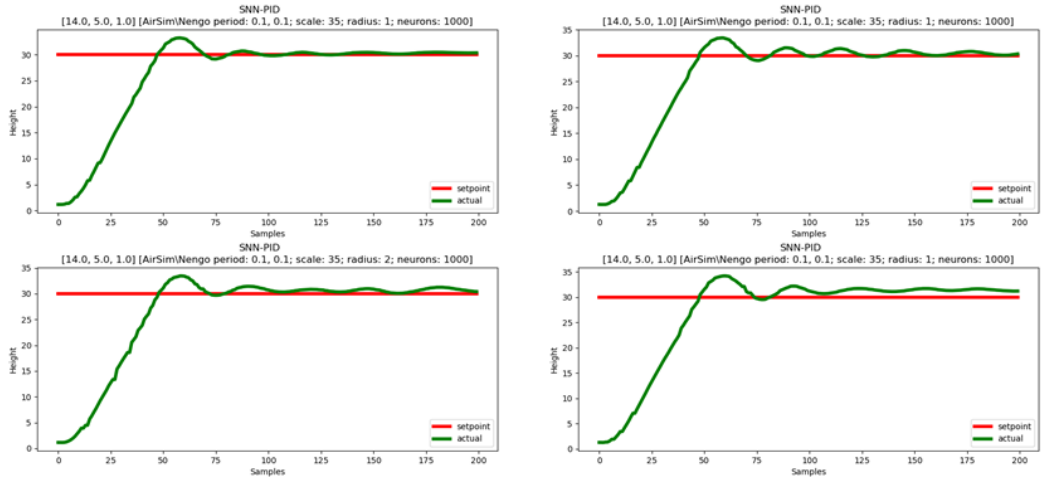



Fig. 54. SNN-PID with same PIG Gains and different radius, synapse values

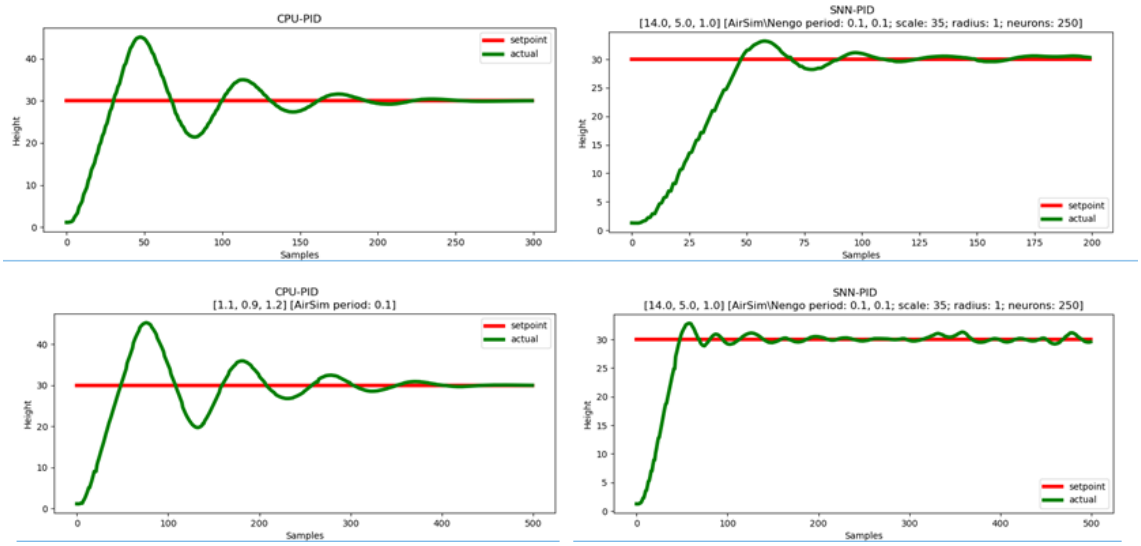



Fig. 55. CPU-PID (left) and SNN-PID (right) performance and dynamics comparison

resumed to allow Nengo execution. Also, Nengo simulator has very high latency far beyond a reasonable range required for a PID controller to operate on a real drone. Undoubtedly there will be considerable reality-gaps from real neuromorphic hardware if there was an attempt to run on a real neuro-chip. Some additional challenges are in the integration points between the conventional programming and the Nengo neuromorphic paradigm.

### 6.4 Summary and future direction

The practical work included building an SNN-PID controller using Nengo, controlling a simulated drone in Microsoft AirSim. Behavior tuning was evaluated and the performance was found to be comparable with a CPU-PID. A future direction is to combine motor control with vision and online

learning with a more capable and advanced drone simulation i.e., use Zurich university drone simulator Flightmare https://uzh-rpg.github.io/flightmare/ which could pave the way to Lava and Loihi integrated with a real drone. The future direction beyond this work is therefore to first add improvements and optimizations, and then to combine neuromorphic modules of motor control, vision and learning (hopefully running on Loihi integrated in a real drone)

## 7 SUMMARIZING THOUGHTS

This work covered a summary of four articles and a section of a practical project inspired by two of these articles. In the articles we have seen ongoing work of researchers and engineers working towards realizing neuromorphic robotics for drones. The first article presented an improved version of a SNN-PID controller on Loihi controlling a constrained 1 DoF drone in real time. Improvements and redesign were critical for the latency and frequency of the controller. The second article presented autonomous landing using Optic Flow Divergence inspired by insects with SNN trained using an evolutionary algorithm in an abstracted simulation environment after which it was tested in a real drone in a real test environment. The primary challenge discussed in this was the reality gap between simulation and the real world. The third article presented integration of neuromorphic vision and motor control with online learning to control a constrained drone to track a fast rotating disc, paving the way to a wider neuromorphic system encompassing multiple neuromorphic modules with advanced cognitive capabilities. The fourth article presented theoretical development of network topology adaptiveness which could potentially allow a neuromorphic controller high degree of adaptivity to different tasks and uncertainties in conditions.

To the best of my knowledge, at the time of writing this work there is currently no commercial or productized drone that uses neuromorphic computing. However, neuromorphic chips are still a relatively new technology in research phase, backed by strong industry players e.g., Intel's Loihi, and it is possible that drone manufacturers will begin to incorporate neuromorphic HW and algorithms into their designs in the upcoming future. Obviously, there are a plethora of research projects exploring the use of neuromorphic computing in robotics, autonomous systems and specifically in drones.

In my personal opinion there is a powerful potential future in neuromorphic robotics. Neuromorphic computing brings a promising paradigm for powering future autonomous drones making them easier to operate, perform a wider variety of operations, flying more safely and for longer duration. So it is possible that commercial drones using neuromorphic chips for some of their autonomous sub-systems may become a reality in the coming years.

There are also considerable challenges in neuromorphic robotics for drones. In my opinion, the two critical challenges beyond creating neuromorphic algorithms are the reality gap between simulation and real world and neuromorphic computing integration with conventional computing frameworks. To fulfil the promise of neuromorphic robotics for drone must overcome these barriers. Then, the long term goal of neuromorphic robotics for drones will be to combine multiple cognitive capabilities of vision, control, perception, planning, online learning and more, leading to a highly autonomous intelligent drones with powerful edge AI that can autonomously operate complex tasks in stochastic environments in real time, safely and robustly.

## 8 APPENDIX A - INTEL'S LOIHI

Loihi is a neuromorphic research processor [6] first introduced in 2018 by Intel comprised of 2 billion transistors and 33 MB of SRAM over its 128 neuromorphic cores and three x86 cores. The chip is fully digital, functionally deterministic and asynchronous. The 128 neuromorphic cores each contains 1,024 spiking neural units, for a total of 131,072 neurons and more than 130 million synapses.
Programming Loihi is done by a Python-based API that allows to define the SNN topology and parameters, program learning rules, and setup communication with host system.
Loihi demonstrates incredibly time and energy efficiency. Average energy per neuron update is between 52 pJ (inactive) to 81 pJ (active) taking only 5.3ns to 8.4ns respectively.
The neuromorphic computing field spans a range of different neuron models and abstraction levels. Loihi uses a variation of the CUBA LIF model with two internal state variables, the synaptic response current ui(t) and the membrane potential vi(t).

$$u_{i(t)} = \sum_{j \neq i} w_{i,j}(\alpha_u * \sigma_j)(t) + b_i \qquad \dot{v}_i(t) = -\frac{1}{\tau_v} v_i(t) + u_i(t) - \theta_i \sigma_i(t)$$

Fig. 56. Loihi CUBA LIF mode state variables

Loihi approximates the above dynamics using a fixed-size time-step model so that the dynamics evolve synchronously. The model includes computational primitives such as stochastic noise, configurable delays, configurable dendritic tree processing and more.

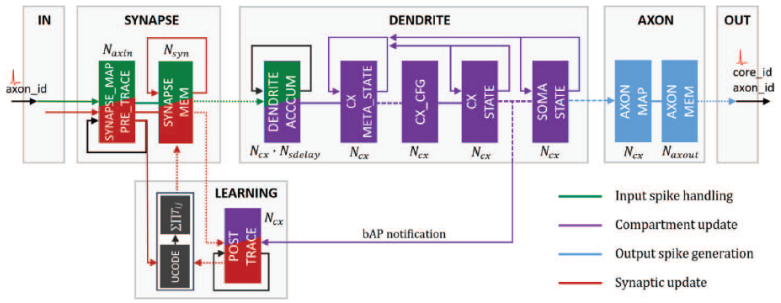



Fig. 57. Loihi core architecture in high level

In 57 can see the high level architecture of Loihi cores. The SYNAPSE unit processes all incoming spikes. The DENDRITE unit updates the state variables of the neurons. The AXON unit generates spike messages. The LEARNING unit updates synaptic weights using programmed learning rules. Learning in an SNN refers to adapting the synaptic weights for changing SNN dynamics to a desired one. Each Loihi core includes a programmable learning engine that can evolve synaptic state variables over time as a function of previous spike traces. The learning engine supports simple pairwise STDP rules and also much more complicated rules. To support development of learning rules, Loihi offers a variety of local information: pre-synaptic and post-synaptic neurons spike traces, state variables per synapse besides the weights, reward traces and spikes representing reward or punishment signals.
For more detailed information on Loihi please refer to [6].

## REFERENCES


[1] R. K. Stagsted, A. Vitale, A. Renner, L. B. Larsen, A. L. Christensen, and Y. Sandamirskaya, “Event-based pid controller fully realized in neuromorphic hardware: A one dof study.” in IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), 2020.

[2] R. K. Stagsted, A. Vitale, J. Binz, A. Renner, L. B. Larsen, and Y. Sandamirskaya, “Towards neuromorphic control: A spiking neural network based pid controller for uav.” in Robotics: Science and Systems, 2020.

[3] S. Glatz, J. Martel, R. Kreiser, N. Qiao, and Y. Sandamirskaya, “Adaptive motor control and learning in a spiking neural network realised on a mixed-signal neuromorphic processor,” Proceedings - IEEE International Conference on Robotics and Automation, vol. 2019-May, pp. 9631–9637, 2019.

[4] R. Kreiser, A. Renner, V. R. Leite, B. Serhan, C. Bartolozzi, A. Glover, and Y. Sandamirskaya, “An on-chip spiking neural network for estimation of the head pose of the icub robot,” Frontiers in Neuroscience, vol. 14, 2020.

[5] R. Kreiser, A. Renner, Y. Sandamirskaya, and P. Pienroj, “Pose estimation and map formation with spiking neural networks: towards neuromorphic slam,” in 2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), 2018, pp. 2159–2166.

[6] M. Davies, N. Srinivasa, T. Lin, G. Chinya, Y. Cao, S. H. Choday, G. Dimou, P. Joshi, N. Imam, S. Jain, Y. Liao, C. Lin, A. Lines, R. Liu, D. Mathaikutty, S. McCoy, A. Paul, J. Tse, G. Venkataramanan, Y. Weng, A. Wild, Y. Yang, and H. Wang, “Loihi: A Neuromorphic Manycore Processor with On-Chip Learning,” IEEE Micro, vol. 38, no. 1, pp. 82–99, 2018.

[7] Dupeyroux, Julien Hagenaars, Jesse Paredes-Vallés, Federico Croon, Guido. (2021). Neuromorphic control for optic-flow-based landing of MAVs using the Loihi processor. 96-102. 10.1109/ICRA48506.2021.9560937.

[8] J. J. Hagenaars, F. Paredes-Vall´es, S. M. Boht´e, and G. C. H. E. de Croon, “Evolved Neuromorphic Control for High Speed Divergence-based Landings of MAVs,” IEEE Robotics and Automation Letters, vol. 5, no. 4, pp. 6239–6246, 2020.

[9] Julien Serres, Franck Ruffier. Optic flow-based collision-free strategies: From insects to robots. Arthropod Structure and Development, 2017, 46 (5), pp.703 - 717. ⟨10.1016/j.asd.2017.06.003⟩. ⟨hal-01644523⟩

[10] E. Baird, N. Boeddeker, M. R. Ibbotson, and M. V. Srinivasan, “A universal strategy for visually guided landing,” Proceedings of the National Academy of Sciences, vol. 110, no. 46, pp. 18 686–18 691, 2013.

[11] Floreano, D., Dürr, P. and Mattiussi, C. Neuroevolution: from architectures to learning. Evol. Intel. 1, 47–62 (2008). https://doi.org/10.1007/s12065-007-0002-4

[12] A. Vitale, A. Renner, C. Nauer, D. Scaramuzza and Y. Sandamirskaya, "Event-driven Vision and Control for UAVs on a Neuromorphic Chip," 2021 IEEE International Conference on Robotics and Automation (ICRA), Xi’an, China, 2021, pp. 103-109, doi: 10.1109/ICRA48506.2021.9560881.

[13] S. Seifozzakerini, W.-Y. Yau, K. Mao, and H. Nejati, “Hough transform implementation for event-based systems: Concepts and challenges.” Frontiers in Computational Neuroscience, vol. 12, p. 103, Dec 2018.

[14] R. S. Dimitrova, M. Gehrig, D. Brescianini, and D. Scaramuzza, “Towards low-latency high-bandwidth control of quadrotors using event cameras,” in 2020 IEEE International Conference on Robotics and Automation (ICRA). IEEE, 2020, pp. 4294–4300.

[15] B. B. Kocer, M. A. Hady, H. Kandath, M. Pratama and M. Kovac, "Deep Neuromorphic Controller with Dynamic Topology for Aerial Robots," 2021 IEEE International Conference on Robotics and Automation (ICRA), Xi’an, China, 2021, pp. 110-116, doi: 10.1109/ICRA48506.2021.9561729.