Learning Autonomous Flight Controllers with Spiking Neural Networks

Huanneng Qiu

A thesis submitted for the degree of
Doctor of Philosophy

School of Engineering & Information Technology
The University of New South Wales Canberra
The Australian Defence Force Academy

Oct, 2021

THE UNIVERSITY OF NEW SOUTH WALES
Thesis/Dissertation Sheet

Surname or Family name: **Qiu** First name: **Huanneng** Other name/s:

Abbreviation for degree as given in the University calendar: **PhD**

School: **School of Engineering & Information Technology** Faculty: **UNSW Canberra**

Title: Learning Autonomous Flight Controllers with Spiking Neural Networks

Abstract

The ability of a robot to adapt in-mission to achieve an assigned goal is highly desirable. This thesis project places an emphasis on employing learning-based intelligent control methodologies to the development and implementation of an autonomous unmanned aerial vehicle (UAV). Flight control is carried out by evolving spiking neural networks (SNNs) with Hebbian plasticity. The proposed implementation is capable of learning and self-adaptation to model variations and uncertainties when the controller learned in simulation is deployed on a physical platform.

Controller development for small multicopters often relies on simulations as an intermediate step, providing cheap, parallelisable, observable and reproducible optimisation with no risk of damage to hardware. Although model-based approaches have been widely utilised in the process of development, loss of performance can be observed on the target platform due to simplification of system dynamics in simulation (e.g., aerodynamics, servo dynamics, sensor uncertainties). Ignorance of these effects in simulation can significantly deteriorate performance when the controller is deployed. Previous approaches often require mathematical or simulation models with a high level of accuracy which can be difficult to obtain. This thesis, on the other hand, attempts to cross the reality gap between a low-fidelity simulation and the real platform. This is done using synaptic plasticity to adapt the SNN controller evolved in simulation to the actual UAV dynamics.

The primary contribution of this work is the implementation of a procedural methodology for SNN control that integrates bioinspired learning mechanisms with artificial evolution, with an SNN library package (i.e. `eSpinn`) developed by the author. Distinct from existing SNN simulators that mainly focus on large-scale neuron interactions and learning mechanisms from a neuroscience perspective, the `eSpinn` library draws particular attention to embedded implementations on hardware that is applicable for problems in the robotic domain. This C++ software package is not only able to support simulations in the MATLAB and Python environment, allowing rapid prototyping and validation in simulation; but also capable of seamless transition between simulation and deployment on the embedded platforms.

This work implements a modified version of the NEAT neuroevolution algorithm and leverages the power of evolutionary computation to discover functional controller compositions and optimise plasticity mechanisms for online adaptation. With the `eSpinn` software package the development of spiking neurocontrollers for all degrees of freedom of the UAV is demonstrated in simulation. Plastic height control is carried out on a physical hexacopter platform. Through a set of experiments it is shown that the evolved plastic controller can maintain its functionality by self-adapting to model changes and uncertainties that take place after evolutionary training, and consequently exhibit better performance than its non-plastic counterpart.

🎓 Welcome to the Research Alumni Portal, Huanneng Qiu!

You will be able to download the finalised version of all thesis submissions that were processed in GRIS here.

Please ensure to include the **completed declaration** (from the Declarations tab), your **completed Inclusion of Publications Statement** (from the Inclusion of Publications Statement tab) in the final version of your thesis that you submit to the Library.

Information on how to submit the final copies of your thesis to the Library is available in the completion email sent to you by the GRS.

Thesis submission for the degree of Doctor of Philosophy

Thesis Title and Abstract | **Declarations** | **Inclusion of Publications Statement**

Corrected Thesis and Responses

ORIGINALITY STATEMENT

☑ I hereby declare that this submission is my own work and to the best of my knowledge it contains no materials previously published or written by another person, or substantial proportions of material which have been accepted for the award of any other degree or diploma at UNSW or any other educational institution, except where due acknowledgement is made in the thesis. Any contribution made to the research by others, with whom I have worked at UNSW or elsewhere, is explicitly acknowledged in the thesis. I also declare that the intellectual content of this thesis is the product of my own work, except to the extent that assistance from others in the project's design and conception or in style, presentation and linguistic expression is acknowledged.

COPYRIGHT STATEMENT

☑ I hereby grant the University of New South Wales or its agents a non-exclusive licence to archive and to make available (including to members of the public) my thesis or dissertation in whole or part in the University libraries in all forms of media, now or here after known. I acknowledge that I retain all intellectual property rights which subsist in my thesis or dissertation, such as copyright and patent rights, subject to applicable law. I also retain the right to use all or part of my thesis or dissertation in future works (such as articles or books).

For any substantial portions of copyright material used in this thesis, written permission for use has been obtained, or the copyright material is removed from the final public version of the thesis.

AUTHENTICITY STATEMENT

☑ I certify that the Library deposit digital copy is a direct equivalent of the final officially approved version of my thesis.

🎓 Welcome to the Research Alumni Portal, Huanneng Qiu!

You will be able to download the finalised version of all thesis submissions that were processed in GRIS here.

Please ensure to include the **completed declaration** (from the Declarations tab), your **completed Inclusion of Publications Statement** (from the Inclusion of Publications Statement tab) in the final version of your thesis that you submit to the Library.

Information on how to submit the final copies of your thesis to the Library is available in the completion email sent to you by the GRS.

Thesis submission for the degree of Doctor of Philosophy

Thesis Title and Abstract | **Declarations** | **Inclusion of Publications Statement**

Corrected Thesis and Responses

UNSW is supportive of candidates publishing their research results during their candidature as detailed in the UNSW Thesis Examination Procedure.

Publications can be used in the candidate's thesis in lieu of a Chapter provided:

- The candidate contributed **greater than 50%** of the content in the publication and are the "primary author", i.e. they were responsible primarily for the planning, execution and preparation of the work for publication.
- The candidate has obtained approval to include the publication in their thesis in lieu of a Chapter from their Supervisor and Postgraduate Coordinator.
- The publication is not subject to any obligations or contractual agreements with a third party that would constrain its inclusion in the thesis.

☑ The candidate has declared that **some of the work described in their thesis has been published and has been documented in the relevant Chapters with acknowledgement**.

A short statement on where this work appears in the thesis and how this work is acknowledged within chapter/s:

This thesis is partially comprised of the following publications that I contributed as the first author:

[1] H. Qiu, M. Garratt, D. Howard, and S. Anavatti. Evolving spiking neurocontrollers for UAVs. In 2020 IEEE Symposium Series on Computational Intelligence (SSCI), Dec 2020, pp. 1928-1935.
[2] H. Qiu, M. Garratt, D. Howard, and S. Anavatti. Towards crossing the reality gap with evolved plastic neurocontrollers. In Proceedings of the 2020 Genetic and Evolutionary Computation Conference, GECCO 20. New York, NY, USA: Association for Computing Machinery, 2020, pp. 130-138.
[3] H. Qiu, M. Garratt, D. Howard, and S. Anavatti. Evolving spiking neural networks for nonlinear control problems. In 2018 IEEE Symposium Series on Computational Intelligence (SSCI), Nov 2018, pp. 1367-1373.

Chapter 3 contains some results from Paper [1] and [3]. Chapter 5 contains some results from Paper [1]. Chapter 6 contains results from Paper [2].

Acknowledgement of the other authors has been made at the beginning of the chapters as well as in the Acknowledgement section.

Candidate's Declaration

I declare that I have complied with the Thesis Examination Procedure.

Acknowledgements

This PhD research has been a long journey. As I look back, I find I am in debt to a lot of people, without whom I would not have been able to reach this step.

I would first like to thank my supervisor, Prof Matthew Garratt for his active guidance and supports throughout this study. His enthusiasm and engagement in his work have impressed me a lot. He has also supported me in every way I needed. Especially, he has put tremendous efforts during the write-up of this thesis. I could not find a better supervisor. Additionally, I would like to thank him for providing his hexacopter simulation model used in this work, as well as helicopter models that have been helpful in my understanding of UAV dynamics.

I would also like to thank my secondary supervisor, Dr Sreenatha Anavatti for his care, encouragement and advice on my work. My wife said the moment she met him, she knew he is a kind man – well, he is also a wise man and I have learned a lot from discussions with him.

I would also like to thank my secondary supervisor, Dr David Howard for his constant, invaluable guidance in this project. He has done so much to help establish my research, and has put a lot of efforts in revising my publications. In particular, Dr Howard also provided me a C++ SNN program at the beginning of this project.

Further, I would like to thank my Master's supervisor, A/Prof Yuanxian Ou, who guided me into the field of robotics. He is an old-fashioned engineer of merits and has always been in the front line solving problems with his students. His dedication has made me want to be an engineer as well.

I want to thank my friends and colleagues – Vu, Ayad, Praveen and others for offering me helps in my research and my life. I also want to thank my Chinese fellows – Qiuxiang, Senyang, Yang Shuo, Zhu Yi and a lot others for being great companions, making my life more colourful. I am also thankful to the staff at SEIT UNSW for sustaining a comfortable and enjoyable working environment for us.

I would like to express my love to my parents, my mother-in-law for their

patience and cares. Especially, my mother-in-law has tried her best to take care of the family as she lives with us now. My love also goes to both my grandmothers, who passed away while I was abroad. I also want to thank my friends back in China for their supports and cares. Not too many have friends like them.

Finally and most importantly, my sincere love always goes to my wife Zhenqi, who has sacrificed her life and career to join me, who has been looking after the home, and who has always been my joy.

This research is funded by UNSW Canberra, and also partly supported by Commonwealth through the "Australian Government Research Training Program Scholarship".

June, 2021

Abstract

The ability of a robot to adapt in-mission to achieve an assigned goal is highly desirable. This thesis project places an emphasis on employing learning-based intelligent control methodologies to the development and implementation of an autonomous unmanned aerial vehicle (UAV). Flight control is carried out by evolving spiking neural networks (SNNs) with Hebbian plasticity. The proposed implementation is capable of learning and self-adaptation to model variations and uncertainties when the controller learned in simulation is deployed on a physical platform.

Controller development for small multicopters often relies on simulations as an intermediate step, providing cheap, parallelisable, observable and reproducible optimisation with no risk of damage to hardware. Although model-based approaches have been widely utilised in the process of development, loss of performance can be observed on the target platform due to simplification of system dynamics in simulation (e.g., aerodynamics, servo dynamics, sensor uncertainties). Ignorance of these effects in simulation can significantly deteriorate performance when the controller is deployed. Previous approaches often require mathematical or simulation models with a high level of accuracy which can be difficult to obtain. This thesis, on the other hand, attempts to cross the reality gap between a low-fidelity simulation and the real platform. This is done using synaptic plasticity to adapt the SNN controller evolved in simulation to the actual UAV dynamics.

The primary contribution of this work is the implementation of a procedural methodology for SNN control that integrates bioinspired learning mechanisms with artificial evolution, with an SNN library package (i.e. `eSpinn`) developed by the author. Distinct from existing SNN simulators that mainly focus on large-scale neuron interactions and learning mechanisms from a neuroscience perspective, the `eSpinn` library draws particular attention to embedded implementations on hardware that is applicable for problems in the robotic domain. This C++ software package is not only able to support simulations in the MATLAB and Python environment, allowing rapid prototyping and validation in simulation; but also capable of seamless transition

between simulation and deployment on the embedded platforms.

This work implements a modified version of the NEAT neuroevolution algorithm and leverages the power of evolutionary computation to discover functional controller compositions and optimise plasticity mechanisms for online adaptation. With the `eSpinn` software package the development of spiking neurocontrollers for all degrees of freedom of the UAV is demonstrated in simulation. Plastic height control is carried out on a physical hexacopter platform. Through a set of experiments it is shown that the evolved plastic controller can maintain its functionality by self-adapting to model changes and uncertainties that take place after evolutionary training, and consequently exhibit better performance than its non-plastic counterpart.

List of Publications

Peer-reviewed publications from this thesis research are listed in a reverse chronological order:

1. **H. Qiu**, M. Garratt, D. Howard, and S. Anavatti. Evolving spiking neurocontrollers for UAVs. In *2020 IEEE Symposium Series on Computational Intelligence (SSCI)*, Dec 2020, pp. 1928–1935. https://doi.org/10.1109/SSCI47803.2020.9308275

2. **H. Qiu**, M. Garratt, D. Howard, and S. Anavatti. Towards crossing the reality gap with evolved plastic neurocontrollers. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference*, GECCO '20. New York, NY, USA: Association for Computing Machinery, 2020, pp. 130–138. https://doi.org/10.1145/3377930.3389843

3. **H. Qiu**, M. Garratt, D. Howard, and S. Anavatti. Evolving spiking neural networks for nonlinear control problems. In *2018 IEEE Symposium Series on Computational Intelligence (SSCI)*, Nov 2018, pp. 1367–1373. https://doi.org/10.1109/SSCI.2018.8628848

Contents

List of Figures

List of Tables

Language Convention

The spelling in this thesis is mostly in the British/Australian style. However, there are also some terms that have been originally or widely used in the American spelling. For consistency purpose I will continue to use their original forms. For instance, the author of the `Boost Serialization`[1] software package has chosen *serialization* over *serialisation*. Therefore, in this thesis, *serialization* is used consistently.

In addition, the prefix *neuro* is usually joined to a word it describes, conventionally without a hyphen. Therefore, in this thesis, *neurocontroller, neuroevolution* and *neuromorphic* are preferred over *neuro-controller, neuro-evolution* and *neuro-morphic*.

[1] `https://www.boost.org/doc/libs/1_58_0/libs/serialization/doc/index.html`

Abbreviations

ADC	Analog-to-Digital Converter
AGI	Artificial General Intelligence
AI	Artificial Intelligence
ANN	Artificial Neural Network
API	Application Programming Interface
BLDC	Brushless Direct Current
CI	Computational Intelligence
CNN	Convolutional Neural Network
CPU	Central Processing Unit
DA	Dopamine
DL	Deep Learning
EA	Evolutionary Algorithm
EC	Evolutionary Computing
ER	Evolutionary Robotics
ESC	Electronic Speed Controller
FLC	Fuzzy Logic Control
FPGA	Field Programmable Gate Array
GA	Genetic Algorithm
GCS	Ground Control Station
IMU	Inertial Measurement Unit
LIF	Leaky Integrate-and-Fire
LQG	Linear-Quadratic-Gaussian
LQR	Linear-Quadratic Regulator
LSTM	Long Short-Term Memory
LTD	Long-Term Depression
LTI	Linear Time-Invariant
LTP	Long-Term Potentiation
MAV	Micro Air Vehicle
MCS	Motion Capture System
MIMO	Multiple-Input Multiple-Output
ML	Machine Learning

MLP	Multi-Layer Perceptron
MoNEAT	Modified NeuroEvolution of Augmenting Topologies
MOSFET	Metal-Oxide-Semiconductor Field-Effect Transistor
MPC	Model Predictive Control
MSE	Mean Square Error
NE	Neuroevolution
NEAT	NeuroEvolution of Augmenting Topologies
OS	Operating System
PID	Proportional-Integral-Derivative
PSNN	Plastic Spiking Neural Network
PWM	Pulse Width Modulation
RC	Radio Controller
RL	Reinforcement Learning
RNN	Recurrent Neural Network
ROS	Robot Operating System
SDK	Software Development Kit
SL	Supervised Learning
SNN	Spiking Neural Network
SNR	Signal-to-Noise Ratio
SoC	System on a Chip
SSH	Secure Shell
STDP	Spiking-Timing Dependent Plasticity
TD	Temporal Difference
UART	Universal Asynchronous Receiver-Transmitter
UAV	Unmanned Aerial Vehicle
USB	Universal Serial Bus

Nomenclature

a, b, c, d	Coefficients of Izhikevich model
a_z	Vertical acceleration
A	Rotor disc area
$\boldsymbol{B}$	Rotation matrix between body-fixed and earth-based coordinates
b_h	Bias in hover throttle estimation
b	Bias in vertical acceleration estimation
C	Capacitance of LIF model
C_d	Drag coefficient
D_x	Fuselage drag in the x-axis direction
D_y	Fuselage drag in the y-axis direction
D_z	Fuselage drag in the z-axis direction
e^n	Normalised error
e_h	Error in height
e_x	Error in the x-axis direction
e_y	Error in the y-axis direction
e_z	Error in the z-axis direction
e_θ	Error in pitch
e_ϕ	Error in roll
e_ψ	Error in yaw
$\bar{e}$	Averaged error
$\|e\|$	Absolute error
f	Fitness
$\boldsymbol{F}$	Net force acting upon the drone
F_x	Force acting in x-axis direction
F_y	Force acting in y-axis direction
F_z	Force acting in z-axis direction
g	Gravitational acceleration
h_r	Hover throttle command
$\tilde{h_r}$	Estimated hover throttle command
I	Input current
$\boldsymbol{I}$	Inertia matrix

I_x	Moment of inertia about x-axis
I_y	Moment of inertia about y-axis
I_z	Moment of inertia about z-axis
J	Cost function for solving a LQR problem
k_h	Coefficient in hover throttle estimation
k_t	Throttle gain in vertical acceleration estimation
k_v	Velocity gain in vertical acceleration estimation
K_d	Derivative gain for a PID controller
K_i	Integral gain for a PID controller
K_p	Proportional gain for a PID controller
K_T	Ratio of rotor command to rotor speed
L	Rolling moment
m	Hexacopter mass
M	Pitching moment
$\boldsymbol{M}$	Net moment of force acting upon the drone
N	Yawing moment
N	Rotor torque
p	Angular rate about x-axis
P	Rotor power
P_i	Induced rotor power
P_0	Profile drag power
q	Angular rate about y-axis
q_0, q_1, q_2, q_3	Quaternion parameters
$\boldsymbol{Q}$	Control signal mixing matrix
r	Angular rate about z-axis
r	Setpoint
R	Resistance of LIF model
R	Radius of rotor disc
S_x	Equivalent flat plate area of fuselage in x-axis direction
S_y	Equivalent flat plate area of fuselage in y-axis direction
S_z	Equivalent flat plate area of fuselage in z-axis direction
t_i	Timestep
t_r	Throttle command, in the range of [0, 1.0]
T	Rotor thrust
u	Recovery variable of Izhikevich model
u	Velocity in body-fixed x-axis
$\boldsymbol{U}$	Vector of movement control command
v	Membrane potential
v	Velocity in body-fixed y-axis
v_{th}	Threshold of membrane potential
v_x	Velocity in x-axis direction

v_y	Velocity in y-axis direction
v_z	Vertical velocity
V_b	Battery voltage
$\bar{V_b}$	Averaged battery voltage
$\tilde{V_b}$	Measured battery voltage
w	Velocity in body-fixed z-axis
w	Connection weight of a neural network
$\boldsymbol{W}$	Vector of rotor speed command
w_{ij}	Connection weight from neuron j to i
X	X position in earth-based coordinate
Y	Y position in earth-based coordinate
Z	Z position in earth-based coordinate
γ	Reward signal
Δw	Change in weight
κ	Correction factor of induced power
η	PWM duty cycle
θ	Pitch angle
θ_r	Pitch command, in the range of [-1.0, 1.0]
ρ	Air density
τ	Decay time constant
ϕ	Roll angle
ϕ_r	Roll command, in the range of [-1.0, 1.0]
ψ	Yaw angle
ψ_r	Yaw command, in the range of [-1.0, 1.0]
$\boldsymbol{\omega}$	Vector of angular velocity
Ω	Angular velocity of rotor
$\boldsymbol{\Omega}$	Vector of angular velocity

Chapter 1

Introduction

1.1 Aim

This thesis project contributes to the field of autonomous unmanned aerial vehicles (UAVs), i.e. aerial platforms that are capable of self-operation to achieve an assigned mission under (partially) uncertain environmental conditions [1, 2]. The aim of this thesis is to present a procedural methodology for the design and implementation for UAV control using plastic spiking neural networks (SNNs). Artificial evolution is exploited in this work for the training of functional network compositions based on a simplified simulated model. Hebbian synaptic plasticity developed offline is utilised for controller adaptation to model variations from simulation to on-board the platform.

A complete pathway for the control of a hexacopter UAV is proposed, which comprises the following three stages:

- identification and modelling of the vehicle dynamics in simulation in which only basic knowledge of the plant is required;
- controller development via artificial evolution based on the identified model with regard to the network structure and configurations, as well as bioinspired synaptic learning rules;
- controller deployment on the target platform, together with an algorithm which facilitates further online adaptation to cross the gap between different platform representations.

Through a set of experiments it has been demonstrated that the evolved plastic neurocontroller can maintain its functionality by self-adapting to model changes and uncertainties that take place after evolutionary training, and consequently exhibit better performance than its non-plastic counterpart.

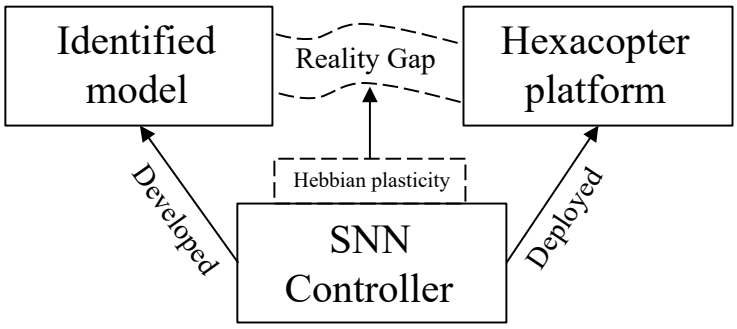


Figure 1.1: Development of flight control using plastic SNNs.

1.2 Motivation

1.2.1 Unmanned Aerial Vehicles

UAVs, or more conversationally known as "drones," are a group of aerial robotic platforms that have been widely employed in public services and commercial and agricultural applications [3, 4], such as professional video recording, disaster assistance as well as farming and pest control. They have also been extensively used in research studies for developing and testing advanced control techniques due to the complexity of the system dynamics and aerodynamics [5, 6].

Generally there are two types of UAVs: fixed-wing UAVs (FUAVs) and rotary-wing UAVs (RUAVs) including single-rotor helicopters and multi-rotor aircraft (i.e. multicopters) [7]. These two types have distinct aerodynamics and specifications in terms of flight endurance, speed, payload and manoeuvrability, and thus have been applied in divergent flight scenarios for different purposes. Fixed-wing aircraft are more power-efficient and can cover longer flight distance as the vertical lift is produced by the forward thrust coupled with the aerodynamics of the wings. On the other hand, vertical lift in an RUAV is generated by the rotations of blades which cause a 'downwash' of air in the opposite direction to the generated thrust.

Compared with FUAVs, RUAVs usually have smaller payload, but can operate in a wider range of environments than fixed-wing aircraft because of their VTOL (vertical take-off and landing) characteristics that allows them to hover and take off in confined spaces. RUAVs are also able to cruise at a lower speed and perform more aggressive manoeuvres than fixed-wing UAVs. A multicopter platform (i.e. hexacopter) is chosen in this work to conduct the experiments.

Multicopters are the most popular type of drones nowadays because of their affordability and modular structure which is easy for manufacturing

and maintenance. The multiple-rotor architecture is more able to provide robustness than helicopters but requires more energy. Multicopters (and helicopters) are naturally less stable than FUAVs, and therefore require a well-tuned flight controller to achieve safe operations [8].

1.2.2 Spiking Neural Networks: Towards Neuroscientific Intelligence

Machine Intelligence? Neuroscientific Intelligence?

During the past decades since computers were invented, humans have been sharing fascination about future robotics where artificial general intelligence (AGI)[1] is realised. Although the development of artificial neural networks (ANNs) and their recent revival with deep learning technologies have significantly advanced the research in the machine learning (ML) domain, we are still very far away from what we have imagined. The fundamental problem is we have not yet fully understood how human (or animal) brains learn and generalise. Specifically, how is information represented and transmitted in biophysical neural systems? How is knowledge gained and developed from interactions with the environment?

From a neuroscientific point of view, information transmission in biophysical neural systems is in the form of timed discrete electrical pulses called *spikes* [9]. *Synaptic plasticity* plays an essential role in the learning activities where cognition builds up meaningfully [10]. An interestingly distinct characteristic of biological brains is that learning in these organisms are considered via local learning rules based on interactions between neighbouring neurons. Whilst the prevailing ANN models utilise a supervised learning paradigm where massive labelled data (i.e. *big data*) is required for the training process; formation and removal, strengthening and weakening of synapses in biophysical models are through synaptic plasticity such as long-term potentiation and depression (LTP and LTD) [11, 12]. Electrophysiological experiments have shown such mechanisms are modulated by Dopamine (DA) as a delayed reward signal [13].

Spiking Neural Networks

This thesis does not attempt to elaborate the theories and phenomenological modelling of biophysical neurons and synapses, or give answers to questions such as how learning is perceived in biophysical brains. Instead, a plausible way of developing autonomous robotics is presented here, by employing

[1] `https://en.wikipedia.org/wiki/Artificial_general_intelligence`

computational spiking neural network (SNN) [14] models with bioinspired learning mechanisms, with the hope of making a contribution to the development of brain-like computing in robotic applications [15].

Spiking neural networks, often referred to as "the third generation of artificial neural networks" [14], are computational systems inspired by the structure of biological brains. Unlike the current, widely used, neuron models that carry out computation via summation of continuous-valued input signals, information transmission in SNNs takes the form of discrete temporal spikes generated during a potential integration process. Each neuron has a *membrane potential* as a measurement of neuronal excitement; receiving multiple incoming spikes within a given time window raises the membrane potential above some threshold, causing an outgoing spike being sent to any forward-connected neurons.

Compared with discrete machine learning applications [16, 17], SNNs in robotic control usually feature with online learning functionality and favour smaller networks in size due to the requirement of real-time computing [18, 19, 20]. Recent SNN studies on robotic control problems [21] have emerged in this area. Such implementations are more advantageous than conventional neural network systems as: i) they are able to yield more powerful computation compared with non-spiking neural systems [14] due to their spatiotemporal dynamics, which can learn to interact internally over *time* and *space* [22]; ii) biological learning rules with SNNs can provide additional learning and adaptation to the system [23, 24, 25], which has become significant in robotic applications where labelled data cannot be obtained. iii) their implementations are power inexpensive on neuromorphic chips because of their event-driven sparsity [26]. This makes them especially suitable to applications where payload weight and power budgets are very tight such as UAVs or space applications.

1.3 Approach

The development of flight control often relies on simulations to provide observable and reproducible optimisation with no risk of damage to hardware. However the system dynamics in simulations are usually simplified as some of the characteristics cannot be easily modelled. A form of learning in autonomous robotics, therefore, is the ability to maintain the robot's functionality when the controller learned in simulation is deployed on the physical platform. In this work, online adaptation is proposed to *cross the reality gap* [2, 27, 28] by utilising Hebbian synaptic plasticity [29] in SNNs.

Controller development in this work is divided into three steps. First a

lower-order simulation model is constructed from a system identification process in which only the essential parts of the system dynamics are captured. An improved neuroevolution algorithm (MoNEAT) is then used to evolve effective network topology and weight configurations in simulation. Hebbian plasticity is also optimised via artificial evolution (*evolution of learning* [30]), in which evolution takes place in the rules that govern synaptic self-organisation instead of in the synapses themselves. The resulting controller solution is finally transferred to a hexacopter platform, together with the algorithm which facilitates further online adaptation. A C++ SNN software package (`eSpinn`) is developed, providing seamless operations from simulations to embedded hardware.

1.4 Contributions

This thesis draws attention to the implementation of a procedural methodology for learning-based SNN flight control that integrates bioinspired learning mechanisms with artificial evolution. The primary contributions of this thesis include:

- the development of a C++ SNN software package (i.e. `eSpinn`), with a particular focus on embedded implementations;
- the improvement of a popular neuroevolution algorithm (i.e. MoNEAT) with an additional mutation operation to discover favourable network topologies more efficiently.
- the construction of a hexacopter platform and the development of controller source code in the ROS environment;
- an incremental evolutionary approach to developing spiking neurocontrollers for flight control in six degrees of freedom; and
- an adaptive approach for hexacopter height control that utilises Hebbian synaptic plasticity for controller online adaptation from simulation to the target platform.

1.5 Thesis Layout

This thesis contains eight chapters. Organisation of the rest of this thesis is as follows. In Chapter 2 a detailed literature review of the related work

on UAV control, SNN learning and its applications to robotic control is presented. In Chapter 3 an SNN machine learning package developed by the author (i.e. `eSpinn`) is introduced. An improved neuroevolution algorithm (i.e. MoNEAT) is proposed for the training of SNNs, with several examples that demonstrate the functionality of this library. Chapter 4 describes the experimental platform and systems used in this thesis, with an introduction to the controller instantiation on the embedded hardware that integrates `eSpinn` with the robot operating system (ROS). Chapter 5 provides a detailed system description of a simulated hexacopter model that is utilised to validate and analyse the proposed control algorithms. An incremental approach to training SNN controllers for the full control of the hexacopter model is given as well. In Chapter 6 the reality gap problem is addressed. A novel approach is proposed for hexacopter height control by leveraging the power of artificial evolution to optimise Hebbian plasticity for controller online adaptation. In Chapter 7 this method is applied to the hexacopter platform described in Chapter 4. Finally, conclusions and discussions are provided in Chapter 8.

Chapter 2

Literature Review

2.1 Introduction

This thesis place an emphasis on evolving plastic spiking neural networks (EP-SNNs) as autonomous controllers for unmanned aerial vehicles (UAVs), which involves multiple fields of research (i.e., adaptive and intelligent robotics, neural networks and evolutionary computation).

There are a variety of active research topics with regard to robotics (or UAVs), ranging from the higher level mission or trajectory planning [31, 32, 33], to collision avoidance [34, 35], to sensing and mapping [36, 37], and finally to the lowest control of the robot [5, 6]. This thesis work is on the development of UAV flight controllers and therefore in this chapter the literature review is confined to the *control*-related work only.

In this chapter, a short clarification of the primary disciplines will first be provided in Section 2.2. Then a detailed review of the related areas will be elaborated. Previous work on system identification is first detailed in Section 2.3. An overview of flight control is then described in Section 2.4. A study on the neural network theory is provided in Section 2.5, followed by a discussion on SNN learning methods in Section 2.6. Finally, Section 2.7 discusses the benefits of SNN-based control approaches and reviews related research in robotic applications.

2.2 Autonomous Robotics

2.2.1 Autonomous vs. Intelligent

The words *autonomous* and *intelligent* (as in autonomous/intelligent control, robotics) have become extensively used in nowadays' research community, yet

there seem to be disputed views towards the definitions of these terms. Some literature has used the phrase "intelligent control" to describe learning-based AI-related control approaches [38, 5], such as neural networks, fuzzy logic, evolutionary computing, reinforcement learning, etc, as a term opposite to conventional control [39]. On the other hand, there are also some literature stating that "autonomous" and "intelligent" are the terms to describe the degree of autonomy [40]. As per [41], the autonomy level of a robotic system is distinguished as automatic, autonomous and intelligent. An autonomous system is capable of decision making and adaptation to achieve its preset goal, while an intelligent system has the ability of reasoning and discovering its own goals and thus has a higher level of autonomy compared with autonomous agents.

From a practical view, I agree most of the existing research on robotics does not involve agents with general artificial intelligence (AGI). However, there factually exist modern control methodologies that are called intelligent control approaches. Therefore, in this thesis, I use

- the term *autonomous* to describe the robots and their behaviours if they are capable of learning and adaption during the interaction with the environment;
- the term *intelligent* to describe the control techniques (neural networks, fuzzy logic, etc) that these robots utilise.

2.2.2 Autonomous UAVs: from a Control Perspective

The underlying fundamentals of a control system incorporates: i) the modelling of the system dynamics, and ii) the control law applied to the system. For a discrete-time control system like robotics, the system can be well described as [42]:

$$\begin{aligned} \boldsymbol{x}(t+1) &= f(\boldsymbol{x}(t), \boldsymbol{u}(t)) \\ \boldsymbol{y}(t) &= h(\boldsymbol{x}(t), \boldsymbol{u}(t)) \end{aligned} \tag{2.1}$$

where $\boldsymbol{x}(t)$ is a vector of state variables, $\boldsymbol{u}(t)$ is a vector of control signals and $\boldsymbol{y}(t)$ is a vector of state measurements. The modelling of the system is represented by the functions f and h, which are generally nonlinear mappings in a UAV application.

The control law $\boldsymbol{u}(t)$ relies on the feedback measurements and can be formulated as:

$$\boldsymbol{u}(t) = g(\boldsymbol{y}(t)) \tag{2.2}$$

From a control perspective, an autonomous robot possesses the ability of learning and self-adaptation, generating a self-regulating control policy from

the interactions with its environment, which enable it to perform a task under uncertain conditions [2]. The autonomy of the system is reflected in

- that the control law is developed by a learning process other than explicit formulation;
- that the control system is able to adapt to uncertainties and improves variation attenuation.

There are a number of fields that are directly related to this research, e.g., neurorobotics [43], evolutionary robotics [44, 30], where robotic research questions are emerging with the focus on interfacing with neuroscience and computational intelligence that otherwise could not be addressed with conventional approaches.

UAV platforms, especially small rotary wing and flapping wing micro air vehicles (MAVs), are particularly challenging platforms for developing and testing such control techniques [5]. Specifically, the two aspects of UAV control (i.e. system modelling and controller development) are both complex problems. In the following sections, these problems will be elaborated.

2.3 System Identification

Although pure-hardware approaches in UAV control have been demonstrated as successful [45, 46, 47], extensive engineering efforts are required to safely conduct the learning process in hardware. As such, learning-based control development for UAVs invariably relies on simulations as an intermediate step [48, 19, 49], providing cheap, parallelisable, observable and reproducible optimisation with no risk of damage to hardware. This is especially the case for evolutionary learning [44], where initial populations are random and the learning process itself is stochastic.

It is not uncommon to derive UAV models mathematically from first principles [50, 8, 51]. However, such models are ill-suited to capturing every aspect of the system dynamics, because some of them cannot easily be modelled analytically, e.g., actuator kinematic nonlinearities, servo dynamics, etc [52]. Ignoring these effects can significantly deteriorate the performance of the designed controller when being deployed onto the physical platform. Loss of performance is likely to arise because the real platform has somewhat different dynamics – which is known as the *reality gap* [53]. Previous approaches often require simulation models with a high level of accuracy. A common practice is to apply a *system identification* process to obtain an "identified" simulated model that reproduces the exact dynamics of the real plant. This

is essentially a data-fitting process in which the exact dynamics are modelled from the measured plant's input and output data, such that the response of the system can be predicted when given the input vector. Such implementations have been successful amongst previous research [54, 55, 52, 56, 57].

Data collection is the first and perhaps the foremost line of work in the system ID process. It is usually accomplished with a range of on-board sensors, e.g., inertial-measurement units (IMUs), global positioning system (GPS) sensors or motion capture systems (MCS). In this project we use a VICON MCS which uses infrared cameras to track the pose of the UAV (refer to Section 4.3 in Chapter 4). There are a number of issues to be noted with regard to data collection. One is that the flight data can be noisy, due not only to the structural vibration effects on the on-board sensors during flights, but also to the uncertainties of electronic measurements. Common filtering methods in signal restoration from on-board sensors include the extended Kalman filter (EKF) [58] and the moving-average filter [59].

On the other hand, model structure is the most difficult aspect to determine, as it requires *a priori* knowledge of the system dynamics [56]. A variety of approaches have been carried out for system identification [60]. In recent years, learning-based black-box approaches have gained popularity in UAV systems, such as fuzzy logic [61], neural networks using the Levenberg-Marquardt (LM) algorithm [62, 52, 63] and deep learning with ReLU [57], as they are convenient ways to capture the system dynamics with only basic knowledge of the system.

2.4 UAV Flight Control

Existing flight controllers can be mainly categorised into three groups [64]: i) classic linear control (i.e. PID controllers); ii) modern model-based control approaches including linear control (e.g. LQR controllers) and nonlinear control; and iii) learning-based intelligent control approaches.

2.4.1 Proportional-Integral-Derivative Control

The proportional-integral-derivative (PID) controller [65, 66] is the most extensively utilised control paradigm in the engineering world. PIDs are linear feedback controllers in which the output u is a correction signal based on the amount of the error e between the desired setpoint r and the measured value of the system output y:

$$e = r - y \tag{2.3}$$

As its name indicates, a PID controller consists of the proportional, integral and derivative terms of the error value and the output (u) is the sum of these three components, as formulated in Eq. 2.4. In some simpler cases, use can also be made of proportional (P), proportional+integral (PI) and proportional+derivative (PD) controllers, in which the integral or derivative terms of Eq. 2.4 are omitted as appropriate.

$$u(t) = K_p e_y(t) + K_i \int e_y(t)dt + K_d \frac{d}{dt} e_y(t) \tag{2.4}$$

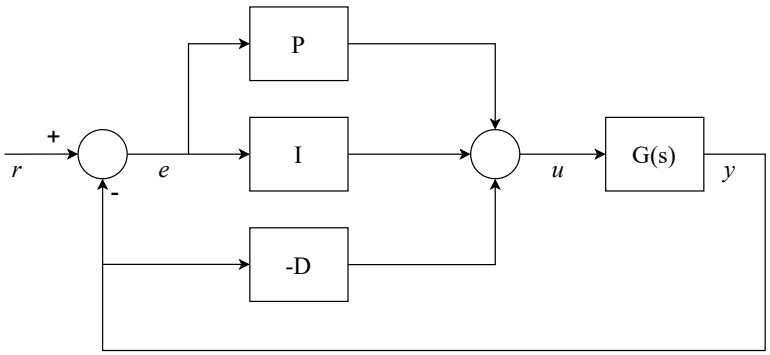


Figure 2.1: A feedback control system using PID control

PID control has prevailed in UAV applications [67, 68] and has been widely used as a benchmark [69, 70, 71, 72] because it is easily understandable and straightforward to implement, yet generally delivers a fairly satisfactory performance. The design and gain tuning can be carried out without knowing the exact dynamics of the system, but by an iterative trial-and-error process.

The open-source flight control project PX4 Autopilot [73] used in this thesis employs a cascaded PID control architecture. Flight control is organised in a nested layered paradigm, where the outputs of the controllers are passed through from the higher-level position controller, to the velocity controller, then to the attitude controller and finally to the innermost angular rate controller.

Since the empirical gain tuning of the PID controllers could be time-consuming, more recently there has also been a variety of work on the automatic gain scheduling of these controllers based on other advanced algorithms, such as neural networks [74], fuzzy logic [75, 76], artificial evolution [77] and reinforcement learning [78].

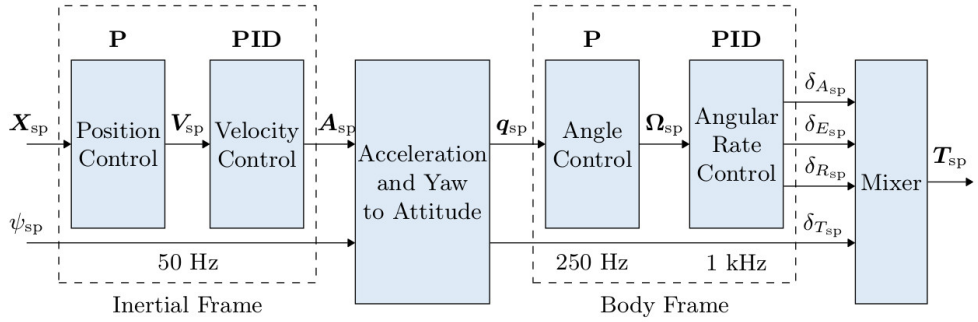


Figure 2.2: A cascaded PID control architecture in PX4 Autopilot, from `https://docs.px4.io/master/en/flight_stack/controller_diagrams.html`.

2.4.2 Model-Based Control Approaches

Linear-quadratic regulator

Another commonly used linear control approach is the linear-quadratic regulator (LQR) [79], which is a model-based optimal control approach in which the controller is developed by minimising a quadratic cost function. Given a linear time-invariant (LTI) model

$$\frac{dx}{dt} = Ax + Bu \tag{2.5}$$

The cost function is formulated by:

$$J = \int_0^\infty (x^T Q x + u^T R u) \tag{2.6}$$

where Q and R are weighting matrices of the state variables and the input vector respectively. Optimal control can be achieved by minimising J and the control output u turns out to be a linear state feedback with a constant gain matrix [80]:

$$u = -Kx \quad \text{where } K = -R^{-1}B^T P \tag{2.7}$$

in which P is the solution matrix to the algebraic Riccati equation (ARE) [42]:

$$PA + A^T P - PBR^{-1}B^T P + Q = 0 \tag{2.8}$$

Further, for the linear-quadratic-Gaussian (LQG) problem where the state measurements of the system are corrupted by Gaussian noise, the LQR controller can be designed together with a combination of a state estimator (e.g.

Kalman filter) [80]. This approach has been applied to the altitude control of an UAV [81].

The LQR or LQG control has been successfully implemented in several UAV applications [67, 82, 83, 70] and have shown to be tolerant against rotor failure [83] and wind disturbance [70]. It has also been used as a training reference for the design of a spiking neurocontroller for an insect-like robot [84] in a supervised learning scheme. Some more discussions can be found in Section 2.7.1.

Other Model-Based Control Approaches

H_∞ control Apart from LQR, H_∞ is another modern linear control method that has appeared in the literature [85, 86, 87, 88], where a model of the controlled system is also required [42].

Model-based nonlinear control approaches These set of control methods are generally developed with the nonlinear modelling of the aircraft obtained from first principles, with parameter identification in some cases [41]. Common methods include feedback linearisation control [89, 90], backstepping [91, 92], and model predictive control (MPC) [85, 93].

2.4.3 Intelligent Control Approaches

Conventional linear control methods (e.g. PID, LQR) are designed based on the linearisation of the controlled system and therefore have restricted performance in complex dynamic systems, while model-based approaches can be too complex and sometimes impractical to acquire the mathematical representation of the system, especially UAV applications that are operating under a number of uncertainties [5].

Intelligent control techniques are superior to conventional approaches because: i) the development does not rely on the mathematical model of the system; ii) the learning ability enables them to cope with model variations and uncertainties. A summary of intelligent flight control methods can be found in [5, 6]. Similar to the system identification process, learning-based intelligent flight controllers also include fuzzy logic [94, 95, 71, 96, 97, 98], neural network models [54, 48, 69, 52, 99, 49] and a combination of both approaches (i.e. neuro-fuzzy control) [100, 101, 102, 103].

Fuzzy Control

Fuzzy control (or sometimes referred to as fuzzy logic control) is a mathematical methodology where the control law is gained from a heuristic understanding of the controlled system [104], imitating the way that human experts aggregate information and develop a control strategy from practical experience during a control design process. Essentially a fuzzy controller is a decision maker that operates based on linguistic-like "if-then" rules. A fuzzy controller has four main components: i) a fuzzifier that employs membership functions to quantity and fuzzify (convert) the input signals to be interpreted by the inference; ii) an inference mechanism that determines which control rules should be applied given the fuzzified inputs; iii) a rule base that consists of a set of fuzzy rules formulated in the form of "if-then" statements; and iv) a defuzzification interface that decodes the fuzzy conclusions into numeric outputs that are passed to the controlled plant.

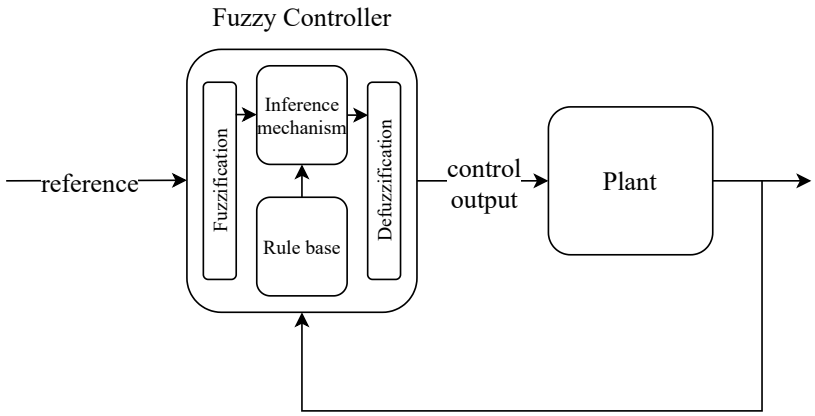


Figure 2.3: A fuzzy control architecture.

Fuzzy control has the advantage that the solution is in the form of human-like reasoning that is well understandable, so that the experience of human experts can be cast into the design of the controller. However, this type of methods has no capability of generalisation [5]. Compared with neural network based control, it can only provide limited learning intelligence [6].

Neural Network Control

Early implementations of neurocontrollers for aerial vehicles date back to work such as that in [105], in which a feedforward neural network is trained using backpropagation to achieve stable hovering of a helicopter with inertial data inputs. Later full control of a quadcopter in four degrees of freedom

(DoF) (i.e. roll, pitch, yaw and altitude control) was developed [106]. The drone is capable of achieving vertical take off and landing, as well as sustaining a predefined attitude using a set of onboard sensors (a tilt sensor, a compass and a gyroscope).

More recently a large group of flight controllers have been designed based on neural networks. With the development of other AI techniques (e.g., artificial evolution [48, 69], reinforcement learning [54, 55, 99, 49]), neural network control has proven successful in a wide variety of UAV control applications. Since the focus of this work is on the development of spiking neurocontrollers, a review of these approaches will be elaborated in details in Section 2.7 after an introduction to the neural network theory in Section 2.5.

2.5 Neural Network Theory

2.5.1 An Overview

Artificial neural networks (ANNs) (or simply neural networks (NNs)) are computational systems inspired by the structure of human brains [107]. They are designed to process and analyse information similar to the way biological neural systems are stimulated.

A neural network is essentially a multi-input-multi-output (MIMO) system with a graph structure, in which the computing units (i.e. *neurons*) operating in parallel are linked by unidirectional edges called *connections*. An NN is usually organised in layers that are made up of neurons and can have different network topologies such as feedforward and recurrent structure. As illustrated in Fig. 2.4, a typical NN consists of an input layer, an output layer and in between layers which are called hidden layers. The input neurons are sensing units perceiving the environment. Signals propagate from the input layer and are passed through the hidden layers to the output layer, via weighted connections. The process to iteratively update the connection *weights* is called the *learning* or *training* process, such that the system can learn to exhibit desired behaviour and perform some certain tasks.

2.5.2 A Brief History

ANNs and the other artificial intelligence (AI) techniques have become unparalleled tools in modern life-changing applications. The development of ANNs has come to an era in which AI can solve problems that would otherwise be impossible for human brains.

However, taking a look back to history, the modern view of ANNs began

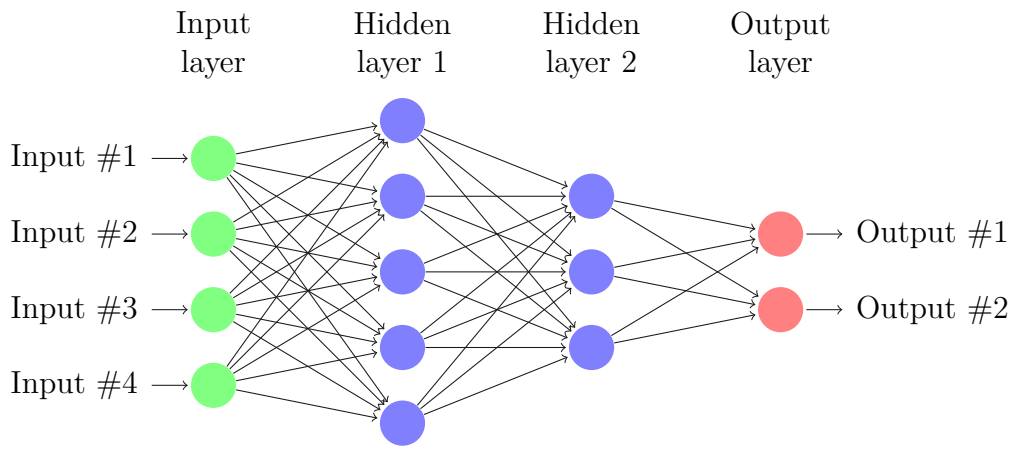


Figure 2.4: Diagram of a feedforward NN with four inputs, two outputs and two hidden layers, in which neurons are fully connected.

with simple concepts. The first generation of ANNs could date back to the 1940s, when the McCulloch-Pitts neuron was first proposed [108]. These units are also referred to as *perceptrons* and can only deal with binary inputs/outputs (which can only be 1 or 0). This basic perceptron network can barely solve a limited class of problems because it does not have much computation capability and is not able to learn by itself.

The second generation of ANNs were based on neuron models which comprises two components as shown in Eq. (2.9): a weighted sum of inputs and an *activation function* (also called *transfer function*) generating the output accordingly.

$$a = f(\boldsymbol{W}\boldsymbol{p} + b) \tag{2.9}$$

where a is the neuron output activated by the transfer function f; $\boldsymbol{p} = p_1, p_2, ..., p_n$ is the input vector of the neuron; $\boldsymbol{W}$ is the corresponding weight matrix and b is the bias. Both the inputs and outputs of these neurons are real-valued. These models have gained popularity with the backpropagation (BP) algorithm [109], which is a gradient descent method that updates connection weights backwards by the chain rule[1]. BP has become a key development for training multilayer perceptron networks (MLPs) and they are still widely used in a variety of applications nowadays.

Recent years have seen a revival of NN implementations in the machine learning (ML) domain, with the development of deep learning (DL) research [110]. DL has significantly improved the progress in image processing with weight sharing convolutional neural networks (CNNs) [111], and in sequential

[1] `https://en.wikipedia.org/wiki/Chain_rule`

data stream applications (e.g., text classification, speech recognition) with long short-term memory units (LSTMs) [112], in a way that traditional tools have never been able to achieve.

The current widely used ANN models follow a computation cycle of "multiply-accumulate-activate." While these models have shown exceptional performance in a variety of machine learning problems, they are highly abstracted from their biological counterparts in terms of information representation, transmission and computation paradigms.

In recent decades, experimental evidence has shown that the communication between biophysical neurons is by means of temporal discrete electrical pulses called *spikes*. This finding has led to the investigation of the "third generation of ANNs" – spiking neural networks (SNNs) [14]. Unlike the current, widely used neuron models that carry out computation via summation of continuous-valued input signals, information transmission in SNNs takes the form of temporal discrete spikes and follow in accordance with the pattern of "integration-threshold-firing & resetting." Each neuron has a membrane potential as a measurement of neuronal excitements; receiving multiple incoming spikes within a given time window raises the membrane potential above some threshold, causing an outgoing spike being sent to any forward-connected neurons.

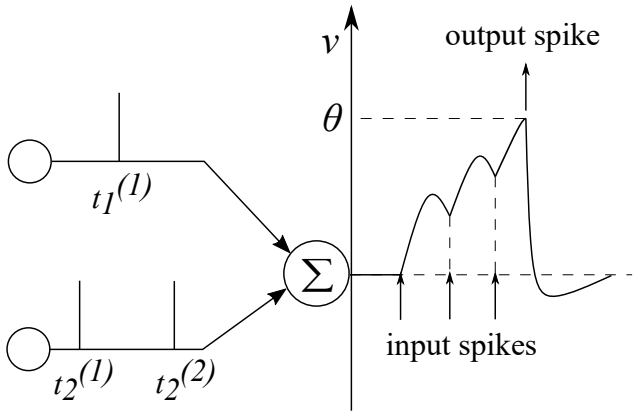


Figure 2.5: Illustration of spike transmission in SNNs. Membrane potential v accumulates as input spikes arrive and decays with time. Whenever it reaches a given threshold θ, an output spike will be fired, and the potential will be reset to a resting value.

Compared with conventional neural systems, SNNs are computationally more powerful [14, 113]. They are also bioinspired learning architectures and can provide faster information processing as observed in biological neural systems [114, 115].

2.5.3 Neuromorphic Chips

Neuromorphic microprocessors where large scale networks of bioinspired neurons and synapses are realised (e.g. TrueNorth [116], the SpiNNaker project [117, 118]) have paved a plausible way to explore low-power brain-like computing. The pure-hardware implementation of spiking neural circuitry has allowed massive parallel bioinspired computing to become a reality [119]. The Tianjic neuromorphic chip [120, 121] is an extremely interesting project which is a configurable hybrid neuromorphic platform that supports SNNs as well as ANNs (MLPs, CNNs and RNNs). It has been applied to balance an unmanned bicycle, which has demonstrated promise for robotic applications.

In recent years, an emerging research interest has focused on integrations of SNNs with event-based neuromorphic cameras called dynamic vision sensor (DVS)[122, 123], where traditional computer vision algorithms are not applicable due to its event-driven property. Such dynamics naturally pair with SNNs. We can find SNNs have been utilised to process the signals for a gesture recognition problem based on the Loihi processor [124] and a goalkeeper project based on the SpiNNaker platform [125].

2.6 SNN Learning

The design of functional SNNs is considered to be difficult, because SNNs behave as complex systems with transient dynamics [126, 127]. Information representation in spiking neuron models is still not entirely resolved. Moreover, synaptic nonlinearities with transmission delay also act as a significant component to the computation power of SNNs. Therefore, the understanding of the learning ability and learning mechanisms of SNNs is still an open challenge in neuroscience [128]. Nevertheless, there is a well-recognised consensus that synaptic plasticity is the key mechanism for learning and memory [129]. From a machine learning perspective, learning in SNNs is more complex than traditional ANNs due to the augmented characteristics. There still does not exist a general-purpose learning algorithm for SNNs [130].

Existing learning methods can be roughly distinguished into three directions: i) gradient-based methods that are derived from the equivalents for traditional neural networks, ii) neuroevolution (NE) algorithms where the gradient information is not required, iii) bioinspired local learning algorithms that are developed for SNNs.

2.6.1 Gradient-Based Methods

While gradient-based learning methods have been very successful in training traditional MLPs [107], their implementations (e.g. SpikeProp [131], ReSuMe [132]) on SNNs are not as straightforward when extracting gradient information from internal spike events because of the discontinuous nature of spiking. Such methods are still immature in SNN training and the convergence properties remain unknown [130]. More recently SNNs have also appeared in the field of deep learning [133, 134, 135, 17]. However, such supervised learning methods require the availability of labelled data and most of the research has been focused on discrete applications (e.g. classification tasks) rather than continuous problems.

2.6.2 Neuroevolution

Neuroevolution [136] is a neural network training methodology in which evolutionary computing (EC) [137] is applied to locate feasible network configurations. Evolutionary algorithms (EAs) are a group of algorithms that are mainly used to solve optimisation and machine learning problems through the stochastic search in a space of possibilities. EAs are able to find out feasible solutions in a huge search space and therefore are perceived as a great tool in neural network training. Evolutionary approaches have also become widely applied in SNN training [138, 139], as the search in the solution space is random without the need for the acquisition of gradient information. The characteristic of a neural network is encoded in artificial genomes and therefore EC is not only able to find out the suitable parameters of connection weights, but also applicable to optimising the network topology itself, which is an essential aspect which affects network functionality [140]. As well as training SNNs before deployment, evolutionary learning has also been a major tool in evolving spiking neural systems (i.e. ESNN) online, where the functionality of the networks are evolved through a variety of continuous learning mechanisms [141, 22]. Note the term *evolving* here by its definition [22], is a broader concept that does not necessarily mean *evolutionary*, although can be achieved by evolutionary approaches.

Evolutionary computing is an abstraction of the natural evolution process. One of the cornerstones of EAs is competition based selection, which is widely known as *survival of the fittest* – individuals that are more adapted to the environment have a higher chance to survive and reproduce. Another basis lies in phenotype variation which incurs diversity over the population. The descendent of those survivors will inherit most of their parents' traits but with random variations that occur during the reproduction process, generating

offspring that loosely resemble their ancestors. Variation operations include *recombination* (also termed *crossover*) where traits of the two parents are combined, and *mutation*, where a small bit of features of the offspring are mutated and differ from any of the parents. The combination of variation and selection thereby leads to a population that is better adapted to the environment, or better able to complete a given task in a methodological sense.

Four variants of EAs have been developed during history: genetic algorithms, genetic programming, evolutionary strategies and evolutionary programming [137]. Nevertheless, the general scheme of EAs is universal (given in Algorithm 1).

Algorithm 1 A General Scheme of Evolutionary Algorithms

```
INITIALISE a population of candidate solutions
EVALUATE each individual & ASSIGN fitness values
while (terminate condition not met) do
    SELECT parents
    RECOMBINE parents
    MUTATE offspring
    EVALUATE new candidates
    SELECT individuals to form the next generation
end while
```

2.6.3 Bioinspired Learning

Hebbian Plasticity

Synaptic plasticity is the key component for learning in biological neural systems [10]. The idea of SNNs is to bridge the gap between neuroscience and computational intelligence, by employing biologically realistic neuron models to carry out computation. The intrinsic plastic feature of biological neural networks has encouraged scientists to investigate the possibility of employing biologically plausible learning mechanisms to computational spiking systems. This class of learning rules are often referred to as "Hebbian learning" because they are inspired by Hebb's postulate [142, 29], in which the weight change between neurons is driven by the causal correlations between presynaptic and postsynaptic spikes:

$$\Delta w_{ij} \propto u_j u_i \tag{2.10}$$

where Δw_{ij} represents the weight change in the connection from neuron j to neuron i; u_j and u_i represent the firing activity of j and i, respectively.

A form of Hebbian plasticity from a temporal spike perspective is the spiking-timing dependent plasticity (STDP) [9], which modulates synaptic weights between neurons based on the temporal difference of spikes. As indicated in Fig. 2.6 the change of weight is driven by the causal correlations between the pre- and postsynaptic spike timings. Studies have shown that STDP can be related to other forms of Hebbian learning rules [143, 144] such as long term potentiation (LTP) and long-term depression (LTD), drawing an intuition on the biophysical process of neural activities.

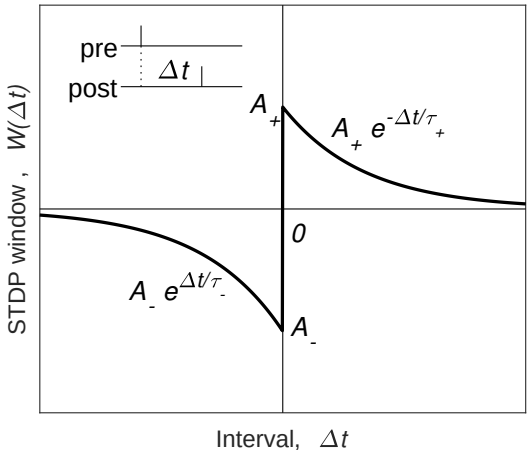


Figure 2.6: STDP window

Reward-Modulated Hebbian Learning

During the studies of these phenomenological learning rules researchers have been trying to understand how Hebbian plasticity is built up over time, i.e., what kind of plasticity can regulate the postsynaptic firing based on incoming presynaptic activations in a desired manner?

Electrophysiological evidence shows that animals learn by rewards carried by Dopamine (DA) [13]. Networks can be insensitive to the ongoing activity and the patterns are preserved in the case where reward signals are not available. This neuroscientific concept called *neuromodulation* [145] has also been studied in the machine learning domain which is usually combined with Hebbian plasticity (i.e. *reword-modulated Hebbian learning* [146]). Generally there are two options approaching the regulation of Hebbian rules: artificial evolution covered previously in Section. 2.6.2 and reinforcement learning (RL) [147] – i.e., evolutionary Hebbian learning and Hebbian reinforcement learning, respectively.

Evolutionary Hebbian learning In this serial of work [148, 149, 150] the authors describe a novel training method for ANNs with the introduction of modulatory neurons. The purpose is to use additional neurons to separate the determination of Hebbian rules that take part in the network configurations from signal processing. In these implementations there are two sets of neurons: the standard neurons calculate the outputs like normal sigmoidal neurons do, while modulatory neurons will contribute to the modification of weights of the surrounding connections. This concept of modulatory neurons has also appeared in earlier work [151], where it is modelled with SNNs in swarm applications.

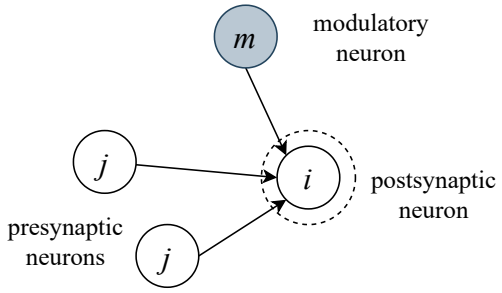


Figure 2.7: Neuromodulation. The connection between the pre- and postsynaptic standard neurons (white circles) are governed by the activation of the modulatory neuron (gray circle).

Connection weights are modified according to the modulatory activations instead of standard activations. A Hebbian plasticity term is introduced in the modulatory part:

$$\Delta w_{ij} = \eta(Au_ju_i + Bu_j + Cu_i + D) \tag{2.11}$$

in which the learning rate η and the scaling coefficients A, B, C and D are determined by an evolutionary process. In addition, the modulatory neurons introduce a multiplication factor m to the change of weight in Eq. (2.11). The resulting Hebbian rule is:

$$\Delta w_{ij} = m\eta(Au_ju_i + Bu_j + Cu_i + D) \tag{2.12}$$

The weight change of the incoming connection to neuron u_i (i.e. Δw_{ij}) is affected by the modulatory activations combined with a plastic term, in which m is a *tanh* activation function of the activity of the modulatory neurons:

$$\begin{aligned} m &= \tanh(m_i/2) \\ m_i &= \sum_{j \in Mod} w_{ji}u_j \end{aligned} \tag{2.13}$$

The Hebbian ABCD model described in Eq. (2.11) has also appeared in other literature that is optimised by either artificial evolution [152, 153, 154] or reinforcement learning [155]. Particularly in this recent work [154], it has been applied to simulate a 2D car-racing task as well as a 3D locomotion task for a quadruped robot, demonstrating that lifetime adaption can be achieved with the evolved Hebbian rules.

Hebbian reinforcement learning The idea of reward modulation also reflects a machine learning paradigm called reinforcement learning, which is a goal-oriented learning process where an agent learns from rewards by the continuous interactions with its environment rather than from a training set of labelled data [156]. In RL, learning is to generate a mapping from the situations to actions by making use of a reward signal [147]. RL is mostly applied in decision making problems where labelled data are not available for supervised learning. RL, with deep learning techniques has come to attract lots of attentions since the birth of AlphaGo [157], a computer program developed by Google DeepMind that learns to play Go by a combination of supervised learning from human expert moves and reinforcement learning from self-play. Later its descendent, AlphaGo Zero [158] which is capable of self-learning without provided human knowledge, has become dominant and defeated all elite professional Go players it played with.

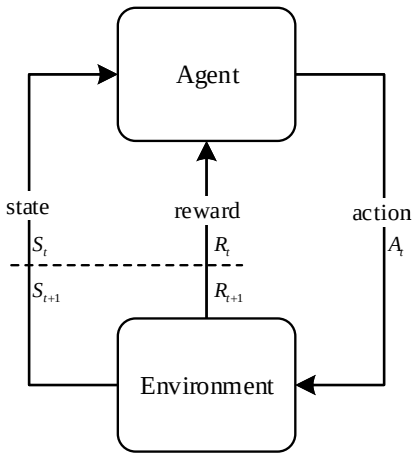


Figure 2.8: The agent-environment interaction in a reinforcement learning paradigm.

Hebbian learning under the RL framework has also been applied to SNNs [159, 160, 161, 162, 163] and other bioinspired neural networks [155, 164].

The aforementioned Hebbian ABCD model is demonstrated successful with RL [155].

2.7 SNN in Robotic Control

The previous section contains the general learning schemes applicable to the training of SNNs. Over the past decades a lot of work has been carried out to solve machine learning problems, most of which has been focused on non-behavioural functionality, e.g., character recognition [165, 166], image classification [167], and approximation [168]. Meanwhile, there is also a rising interest in behaviourally functional SNNs (e.g. SNN in robotic control), which addresses neural activities in closed-loop interaction with the environment [169, 170, 19].

As successful records of spiking neurocontrollers implemented on UAVs have been limited, this section has extended the scope to ground-based vehicles as well. In recent years, SNN studies on robotic control problems [21] have shown promise in this area. Compared with discrete machine learning applications [16, 17], SNNs in robotic control usually require online learning functionality and favour smaller networks in size due to the need to generate control outputs with high frequency and low latency [18, 19, 20]. Recent studies on SNNs have shown their great potential in embedded control systems, for three main reasons:

- Computationally, SNNs are able to yield more powerful computation compared with non-spiking neural systems [14] due to their spatiotemporal dynamics [22], which are characterised by internal interactions over time and space. As such, they can exploit spike timing as well as frequency, and learn to associate data in the time domain as well as in the space domain.

- Biological learning rules with SNNs can provide additional learning and adaptation to the system [23, 24, 25]. These learning methods have become significant in robotic applications where labelled data cannot be obtained in robotic explorations in an environment.

- Binary signalling makes SNNs synergistic with upcoming neuromorphic devices, e.g., where the network is instantiated on-chip [171]. These implementations promise low power due to relatively sparse spiking events [26], as well as fast compute in pure-hardware via analog electronic implementations of spiking circuitry [172].

2.7.1 Hebbian Learning

A major group of SNN neurocontrollers is to employ bioinspired local learning rules (e.g., Hebbian learning, STDP). This approach has been applied to train a behavioural SNN controller for a mobile robot [173] that can avoid obstacles in a simulated environment using sonar sensory signals. The spiking controller has simpler structure compared with conventional ANNs. Later work [174] has complemented the controller with target approaching functionality, which consists of three submodules: wall following, obstacle avoidance and goal approaching. The behaviour of the robot was controlled by one of the submodules at each state determined by a weight setting module.

Obstacle avoidance can also be seen using STDP/anti-STDP on a physical wheeled robot [175]. Provided a reward signal determined by whether the robot moves forward or hits an obstacle, this learning rule is able to perform online weight modification. The spiking neurons are evolved to a semi-synchronous state which is beneficial to manage the coupling between the environmental inputs and the behavioural output.

Hebbian plasticity is a way to synchronise the firing activities of the neurons. The reward signal is used to tell if the changes are on the desired direction and thus provide a benchmark to regulate the synchrony. Such reward-based learning methods can be related to reinforcement learning [147] which can also be found in other work. Reward-modulated STDP learning methods have been introduced to solve basic classification (i.e. XOR) problems [176, 177]. In both papers, a reward signal is used to determine whether the causal correlation of neuron spikes should be reinforced, e.g., the synaptic weight is increased when the reward is positive. A series work has also been carried out in Duke University on Hebbian-based indirect SNN training where the synaptic weights of the networks are modulated using external stimuli. A reward-modulated Hebbian algorithm [178] is used to train a critic flight controller in which a reward function is defined based on the errors between the SNN outputs and the desired outputs. Synaptic strengths of an action controller is then modulated with provided spike inputs from the critic controller, rather than by direct weight manipulation. Later in [179] this indirect learning method is used to train the network to control a virtual insect to seek for a target location. Similarly instead of directly changing the synaptic weights, it seeks to optimise the input spikes by implementing a square-pulse function using a radial basis function (RBF) spike model. Synaptic weight change is driven by input stimuli forcing neurons to fire in desired patterns. Neuromorphic hardware implementation of the SNN controller with STDP learning rules is realised in [180] and [181].

A similar reward-based method is used to control a simulated insect-alike

micro robotic bee [84] designed by Harvard Microrobotics Lab [182]. The proposed controller is capable of achieving stable hovering and trajectory following. An SNN lateral controller is taught to mimic a target LQR controller with known performance. During the training, the state of the robot is fed back to the SNN controller and LQR controller respectively. The output of the LQR controller is set as the reference value and a reward signal is used to modulate connection weights based on the error between the reference output and the decoded network output. In the following work, the authors develop an adaptive SNN controller that exhibits online adaptation functionality [183] during a simulated hovering flight. The neurocontroller comprises two components: an ideal term that is trained offline to approximate a proportional-integral filter (PIF) controller developed against the ideal model; and an adaptive term that is able to learn online to compensate unmodelled system dynamics such as manufacturing variations and actuator dynamics. The aim of the adaptive component is to minimise the steady-state errors between the actual positional velocities and the reference signals in the three translational dimensions.

In [184] the authors use a reward-modulated STDP learning rule to train a target-approaching controller that has a similar structure to the above-mentioned work [174]. The controller has two functionalities - obstacle avoidance and goal approaching. The reward signal is calculated from the error of the output from its reference. Rewards of each connection are backpropagated through the layers.

In Section 2.6 it is mentioned that the intensity and even the direction of synaptic modifications in biophysical neural systems can be modulated by Dopamine. This reward-modulated behaviour can be associated with a class of RL methods called *temporal difference* (TD) learning. A spiking controller is proposed for a simulated wheeled robot to perform wall-following tasks [24]. The robot learns to associate the correct movement with appropriate input conditions via the TD learning rule. In another work, an SNN is used as an altitude controller trying to stabilise a simulated microrobotic bee while performing a take-off manoeuvre [25], based on a reward-modulated STDP via TD learning.

2.7.2 Evolutionary SNNs

This research field is also categorised in the domain of evolutionary robotics (ERs) [44], where evolutionary algorithms (EAs) are applied to robotics embedded in real-time and real environments (either physical or virtual) [185]. Such systems are very different from evolutionary implementations on static ML problems such as classification and regression problems, which are fun-

damentally optimisation from the evolutionary point of view.

Early implementation of evolutionary spiking neurocontrollers could be found in the work of the Autonomous Systems Laboratory at EPFL [18, 45, 186]. This group has focused on vision-based neuroevolution controllers. The authors first evolve a spiking controller for a mobile robot to perform vision-based navigation tasks [18]. Then a similar method is used to steer an indoor blimp [45] to navigate within a room. The proposed evolution here is considerably simple, as only the signs ($\pm$) and presence or absence (1/0) of connections are considered.

In another work, the authors employ simple networks which consist of only four neurons to control a physical wheeled robot [187]. An adaptive online EA is used to evolve both weights values and signs, which are represented by five binary bits. A parallel architecture combining two spiking networks is evolved to control the movement of a mobile robot [188]. Each network processes information from the left/right-side sensor. At each step a selector is used to select the more active network as the output network.

Hardware evolutionary implementation has also been applied to reconfigurable Field Programmable Analogue Arrays (FPAAs) [189]. Both synaptic weights and neuron spiking thresholds are evolved. Mobile robot control can also be seen by using SNN clusters of internally interacting neurons to act collectively as individually operating neurons [20]. Such configurations can be viewed as an approach of population encoding [9] that allows the neurons to run parallelly in hardware implementations and therefore the processing time can be reduced to gain the firing rates.

In UAV control, spiking controllers are evolved to navigate a dynamic quadrotor to stay at a waypoint under challenging wind conditions [19]. EAs are used to generate more satisfied topologies and weights. It is noted that the mutation rate of each network is also self-adaptive. An incremental evolution strategy is employed. Population is first trained in simplified tasks, then seeded as an initial population for more complex tasks. Simulated evidence shows that the evolving spiking networks can stabilise the UAV more accurately and more rapidly than PID controllers and MLP neural networks. A significant finding in this research is that spiking controllers are more "evolvable" than traditional networks, thus are more able to cope with uncertainties.

From the above literature review it is worth highlighting some comments:

- Most of the spiking controllers are tailored for ground-based vehicles. UAV control can only be found performing simulated basic manoeuvres [19]. For ground-based robots, evolution is applied to the formation/removal of connections, while in more complex situations, a larger search

space with more parameter configurations is required to locate beneficial solutions.

- Evolution is either carried out directly on hardware or in simulation only. Physical hardware implementations are restricted to controlling less-dynamic, error-tolerant systems [45]. Hardware SNN control for dynamic UAVs can be problematic because randomly initialised networks are likely to crash UAVs. A common approach in ER is to employ offline evolution to optimise the controller and then deploy the evolved pseudo-optimal controller onto the target platform. However, as explained in Section 2.3, such controllers do not change after the evolutionary training and therefore cannot adapt to unmodelled dynamics and model changes and their performance will be degraded.

2.7.3 Evolving Plastic Neurocontrollers

Continuing the previous discussions, evolution can be perceived as learning on a long-run time scale where the adaptation takes place across several generations [190]. Traditionally evolution in NE is taken as a way of searching for the optimal network configurations. As a result, controllers that are evolved in simulations have static network topologies as well as connection weights when deployed [191]. Such controllers cannot adapt online reducing their performance.

On the other hand, a form of learning in autonomous robotics is the ability to maintain the robot's functionality when the controller solution learned in simulation is transferred to the physical platform, which is often referred to as the ability to *cross the reality gap* [2, 27, 28]. This is in line with a special subset among neuroevolution called *evolution of learning* [30] where evolution is applied to the plastic learning rules of the networks that regulate the connection weights rather than the connections themselves [192, 153, 193, 194, 163, 195, 154] (refer to **evolutionary Hebbian learning** in Section 2.6.3). The resulting robot is able to adapt to model changes that take place after the evolutionary training [196, 197, 191] and therefore exhibit a form of autonomy. Such methodology has also applied to SNNs (i.e. evolving plastic SNNs (EPSNN)) to control simulated robots [198, 199].

2.8 Summary

There is a knowledge gap in SNN implementations for robotic control, as there still does not exist a general-purpose learning algorithm for SNNs. Current

SNN implementations are more focused on classic non-behavioural machine learning problems, while at the same time, there have been emerging research interests that investigate practical robotic applications. High performance learning and adaptation can be achieved in spiking neurocontrollers by integrating bioinspired plasticity with the power of evolutionary computing and reinforcement learning. In addition, neuromorphic microprocessors where large scale networks are realised have paved a plausible way to explore brain-like computing with high processing speed and low power consumption features.

Chapter 3

eSpinn – A Machine Learning Package for Spiking Neurocontroller Development

This chapter is partly based on the following publications:

H. Qiu, M. Garratt, D. Howard, and S. Anavatti. Evolving spiking neurocontrollers for UAVs. In *2020 IEEE Symposium Series on Computational Intelligence (SSCI)*, Dec 2020, pp. 1928–1935. https://doi.org/10.1109/SSCI47803.2020.9308275

H. Qiu, M. Garratt, D. Howard, and S. Anavatti. Evolving spiking neural networks for nonlinear control problems. In *2018 IEEE Symposium Series on Computational Intelligence (SSCI)*, Nov 2018, pp. 1367–1373. https://doi.org/10.1109/SSCI.2018.8628848

3.1 Introduction

Currently existing SNN simulators mainly focus on in-depth simulations of biologically realistic neuron interactions [200, 201, 202, 203]. A few are developed for network implementations on *discrete* applications in the machine learning domain [204], which is also difficult to apply in *continuous* robotic control problems, because the latter usually require online learning functionality and favour smaller networks in size due to the need for real-time computing with high frequency and low latency. The iSpike [205] library is the only open source software package in the robot control domain. However its main focus is on interfacing an SNN simulator (i.e. NeMo [206]) and a humanoid robotic platform (iCub), i.e., how sensory information is encoded

into spike patterns and how spike outputs are decoded into motor commands, rather than the *control* of robots itself. Nevertheless, SNNs have shown great potential in embedded control systems: a) they are able to yield more powerful computation compared with non-spiking neural systems [14] due to their spatio-temporal dynamics; b) they are also believed to be power inexpensive on neuromorphic chips because of their event-driven sparsity [26], which perfectly suits embedded applications such as UAVs.

This chapter presents a C++ software package called `eSpinn` as a tool to developing spiking flight controllers by integrating biological learning mechanisms with neuroevolution algorithms. An overview of the `eSpinn` structure is given in Fig. 3.1. This package allows for the flexible development of evolved spiking controllers, which can be seamlessly transferred from MATLAB and Python simulations of embedded deployment on the target robot. As well as transferring the controller, the learning algorithm itself is transferred, which opens up new possibilities in continuing a simulated learning process on-board the robot.

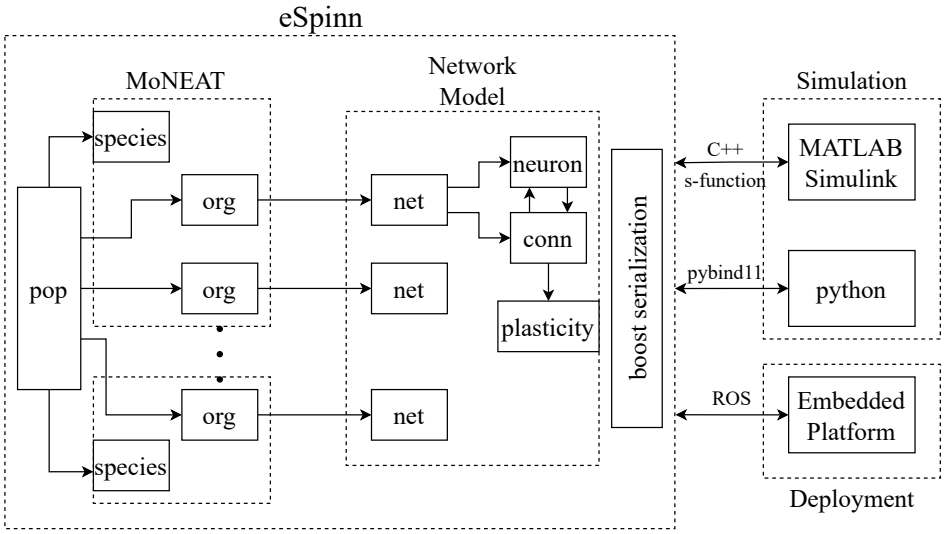


Figure 3.1: Diagram of the `eSpinn` system.

3.2 An Overview

The `eSpinn` library stands for **E**volving **Spi**king **N**eural **N**etworks. It is designed to facilitate the development of spiking network controllers for non-linear control problems via artificial evolution. Currently it supports two popular spiking neuron models (i.e., LIF [126] and 4-parameter Izhikevich

[207]), as well as conventional non-spiking neurons with linear or sigmoid activation functions. It runs a modified version of the NEAT neuroevolution algorithm [140], which has been altered to provide enhanced performance when evolving small scale SNNs.

`eSpinn` is an open source software package released under the Apache License 2.0. It is freely available on GitHub (`https://github.com/hnqiu/eSpinn`). Currently `eSpinn` has only been tested on Ubuntu ($\geq$ 14.04). It is written in C++ and uses `cmake` to manage the build process, with two third-party library dependencies (`Boost Serialization`[1] and `pybind11`[2]).

`eSpinn` is implemented in an object-oriented programming (OOP) style, by making use of the C++ polymorphism feature. As such, it is able to accommodate different network implementations (ANNs, SNNs and hybrid models) with specific dataflow schemes. `eSpinn` features with easy-to-use data archiving interfaces to save and construct data structure to or from files, which is convenient in offline-online hybrid training. `eSpinn` also features a Python wrapper using the `pybind11` library. The C++ data types and methods are exposed in Python, which enables the package to be callable as a Python module.

This software package is organised as follows:

```
/--
  |-- asset/
    |-- archive/
    |-- data/
    |-- docs/
  |-- game/
  |-- scripts/
  |-- src/
    |-- Learning/
    |-- Models/
    |-- Plants/
    |-- PyModule/
    |-- Utilities/
    |-- CMakeLists.txt
    |-- eSpinn_def.h
    |-- eSpinn.h
    |-- files_def.h
  |-- tasks/
  |-- test/
  |-- CMakeLists.txt
  |-- eSpinnLog.md
  |-- README.md
```

[1] `https://www.boost.org/doc/libs/1_58_0/libs/serialization/doc/index.html`
[2] `https://github.com/pybind/pybind11`

The `src/` folder contains the C++ source files of this package, which are categorised into several sub-directories. `src/Models/` contains definitions of neuron models, connection models and the resulting network models, while the `src/Learning/` folder contains the modified NEAT (MoNEAT) as the main learning algorithm. The `src/PyModule` includes Python wrappers to bind the C++ data types to be called in Python programs located in the `tasks` directory. `src/Utilities/` contains interfaces for archiving data structure, training data and variables during the learning process. Generated data will be saved in `asset/data/` and `asset/archive/`, and can be visualised by MATLAB and python scripts in the `scripts/` folder.

`src/Plants/` contains definitions of several plant models, with simulation tasks provided in the `tasks/` folder. The `test/` folder contains testing programs to debug and validate the source files.

3.3 Learning with Spikes

The current widely used ANN neuron models follow a computation cycle of *multiply-accumulate-activate*. The neuron model consists of two components: a weighted sum of inputs and an activation function generating the output accordingly. Both the inputs and outputs of these neurons are real-valued. While ANN models have shown exceptional performance in the artificial intelligence domain, they are highly abstracted from their biological counterparts in terms of information representation, transmission and computation paradigms.

SNNs, on the other hand, carry out computation based on biological modelling of neurons and synaptic interactions. Information transmission in SNNs is by means of discrete electrical pulses called *spikes* generated during a potential integration process. As shown in Fig. 3.2, each neuron has a *membrane potential* as a measurement of neuronal excitements. The dynamics of spiking neurons are described as the response of internal membrane potential to input spikes. Spikes are transmitted via synapses from the *presynaptic* neurons to all forward-connected *postsynaptic* neurons. The membrane potential of the postsynaptic neuron will accumulate upon receiving a spike, otherwise it will decay with time until it reaches a resting potential. Whenever the potential exceeds a given threshold, an output spike will be fired and delivered forwards. The information measured by spikes is in form of timing and frequency, rather than the amplitude or intensity.

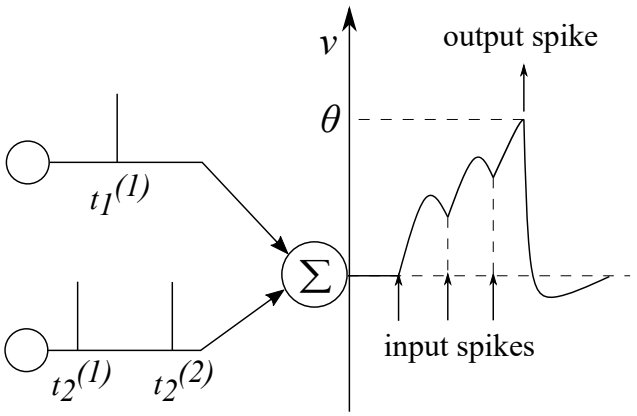


Figure 3.2: Illustration of spike transmission in SNNs. Membrane potential v accumulates as input spikes arrive and decays with time. Whenever it reaches a given threshold θ, an output spike will be fired, and the potential will be reset to a resting value.

3.3.1 Neuron Model

To date, there have been different kinds of spiking neuron models. The Hodgkin-Huxley (HH) model [208] was presented to describe the biologically accurate and detailed mechanism of neuron activities. Therefore, the simulation of this type of model requires huge computational power. Instead, two phenomenological alternatives, the integrate-and-fire (IF) model [9] and the 4-parameter Izhikevich model [207] have been widely used in computational SNNs, due to their simplicity whilst being able to approximate the behaviour of biological spiking neurons. A number of generalisations of I&F models have been introduced, such as the leaky integrate-and-fire (LIF) model and the quadratic IF (QIF) model [126].

One way to provide intuition into the complexity range of these neuron models is by simulating the computational cost. Approximately 1200 floating point operations per second (FLOPS) would be required to simulate the activities of a HH model for 1ms, while only 5 are required for the LIF model and 13 for the Izhikevich model [126]. When implementing a neuron model, trade-offs must be considered between biological reality and computational efficiency.

`eSpinn` currently supports two spiking neuron models (i.e., LIF and Izhikevich), as well as conventional non-spiking neurons with linear or sigmoid activation functions.

The LIF model is a simple spiking neuron model that has been widely used in the literature. Its dynamics can be represented by an elementary electrical

circuit composed of a capacitor C in parallel with a resistor R (Fig. 3.3). The response of the internal membrane potential v is driven by an input current I, which is described as a first-order linear differential equation:

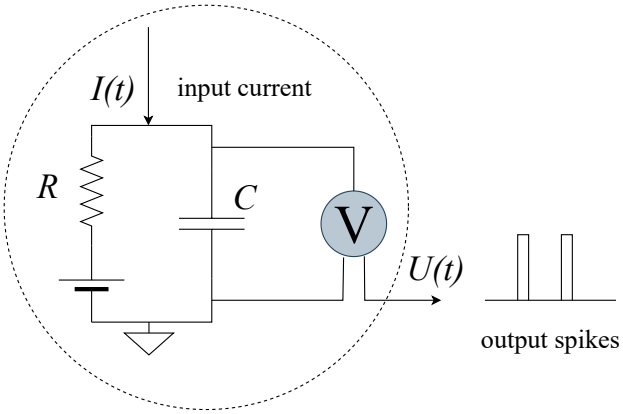


Figure 3.3: Equivalent circuit model for a LIF neuron.

$$C\dot{v} = I - \frac{1}{R}(v - v_{rest}) \tag{3.1}$$

namely,

$$\tau\dot{v} = -(v - v_{rest}) + RI \tag{3.2}$$

where $\tau = RC$ is the passive membrane time constant measuring the voltage leakage rate. The potential v will be reset to a resting value whenever a spike is fired:

$$v = v_{rest} \text{ if } v \geq v_{th} \tag{3.3}$$

where v_{th} is a given voltage threshold.

On the other hand, the two-dimensional Izhikevich model has also gained popularity because of its capability of exhibiting richness and complexity in neuron firing behaviour with only two ordinary differential equations:

$$\begin{aligned} \dot{v} &= 0.04v^2 + 5v + 140 - u + I \\ \dot{u} &= a(bv - u) \end{aligned} \tag{3.4}$$

with after-spike resetting following:

$$\text{if } v \geq v_{th}, \text{ then } \begin{cases} v = c \\ u = u + d \end{cases} \tag{3.5}$$

Here v represents the membrane potential of the neuron; u represents a recovery variable; $\dot{v}$ and $\dot{u}$ denote their derivatives, respectively. I represents

the synaptic current that is injected into the neuron. Whenever v exceeds the threshold of membrane potential v_{th}, a spike will be fired and v and u will be reset following Eq. (3.5). a, b, c and d are dimensionless coefficients that are tunable to form different firing patterns [207]. The membrane potential response of an Izhikevich neuron is given in Fig. 3.4, with an injected current signal.

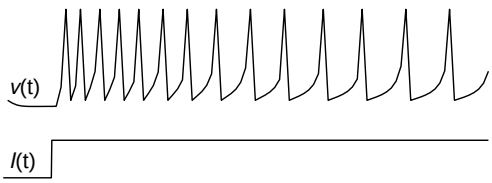


Figure 3.4: Membrane potential response $v(t)$ to an external current signal $I(t)$ of an Izhikevich neuron with the following settings: a = 0.02; b = 0.2; c = -65; d = 2.

A spike train is defined as a temporal sequence of firing times:

$$s(t) = \sum_f \delta(t - t^{(f)}) \tag{3.6}$$

where $\delta(t)$ is the Dirac δ function; $t^{(f)}$ represents the firing time, i.e., the moment of v crossing threshold v_t from below:

$$t^{(f)} : v(t^{(f)}) = v_t \text{ and } \dot{v}(t^{(f)}) > 0 \tag{3.7}$$

3.3.2 Rate Coding vs. Temporal Coding

Deciphering the information in spike train patterns is an important aspect in SNN studies, not only in understanding the nature of biological neural systems, but can also provides intuition in developing mechanisms of computational models.

Computationally, the information representation of spiking neural system can be distinguished between *rate coding* and *temporal coding* schemes [209]. In a rate coding scheme, neural information is encoded into the number of spikes occurring during a given time window, i.e. firing *rate* of spikes. In a temporal coding scheme, the context is encoded into the *timing* between presynaptic and postsynaptic spikes. Other different terminologies can be found to describe these two information representation methods. In [9] they are described as *rate-based* and *spike-based* coding approaches, while in [18] they are described as *connectionist* models and *pulsed* models.

In a rate coding scheme, neuron output is defined as the spike train frequency calculated within a given time window. Loss of precision during this process is likely to happen. `eSpinn` configures a decoding method with high accuracy to derive continuous outputs from discrete spike trains. The implementation involves direct transfer of intermediate membrane potentials as well as decoding of spikes in a rate-based manner.

3.3.3 Network Structure

In `eSpinn`, networks are organised in a three-layer architecture that has hidden-layer recurrent connections, as illustrated in Fig. 3.5. The input layer comprises *encoding* neurons which act as information converters. Hidden-layer neurons are connected via unidirectional weighted synapses among themselves. Such internal recurrence ensures a history of recent inputs is preserved within the network, so that they can share weights over time. Output neurons can be configured as either activation-based or spiking. A bias neuron that has a constant output value is able to connect to any neurons in the hidden and output layers. Connection weights are bounded within [-1, 1]. The MoNEAT (detailed in Sect. 3.4.3) topology and weight evolution algorithm is used to form and update network connections and consequently to seek functional network compositions.

The configuration of each layer is independent from each other. Therefore, networks can be formed with different types of neurons with specific dataflow. For instance, we may construct a spiking network which outputs a binary spike status. We may also use spiking neurons in the hidden layer and sigmoidal neurons as the output, in which information transmission is real-valued, in the form of mean firing rates plus intermediate membrane potential.

3.3.4 Hebbian Plasticity

In neuroscience, studies have shown that synaptic strength in biological neural systems is not fixed but changes over time [210] – connection weights between pre- and postsynaptic neurons change according to their degree of causality. This phenomenon is often referred to as Hebbian plasticity as inspired by the Hebb's postulate [142], which is summarized as *neurons that fire together, wire together*.

Modern Hebbian rules generally describe weight change Δw as a function of the joint activity of pre- and postsynaptic neurons:

$$\Delta w = f(w_{ij}, u_j, u_i) \tag{3.8}$$

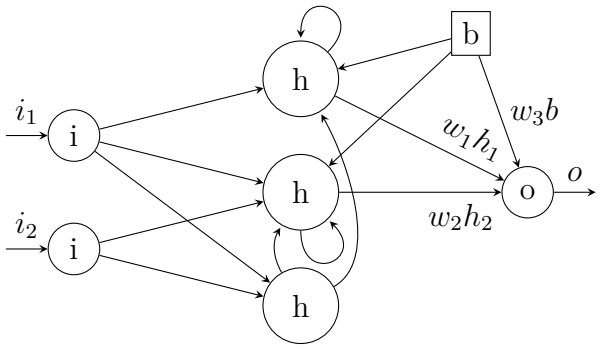


Figure 3.5: Topology of an `eSpinn` network with 2 inputs (i) and 1 output (o) that allows hidden-layer (h) lateral recurrence. The hidden layer consists of spiking neurons whose outputs are derived from direct transfer of intermediate membrane potential as well as decoding of firing rate. A bias neuron (b) is allowed to connect to any neurons in the hidden and output layer. Network output is calculated as a weighted sum of connected neuron activations $\sum w_i o_i$. Weights w_i are bounded within [-1, 1].

where w_{ij} represents the weight of the connection from neuron j to neuron i; u_j and u_i represent the firing activity of j and i, respectively.

In a spike-based scheme, we consider the synaptic plasticity at the level of individual spikes. This has led to a phenomenological temporal Hebbian paradigm: Spiking-Timing Dependent Plasticity (STDP) [9], which modulates synaptic weights between neurons based on the temporal difference of spikes.

While different STDP variants have been proposed [143], the basic principle of STDP is that the change of weight is driven by the causal correlations between the pre- and postsynaptic spikes. Weight change would be more significant when the two spikes fire closer together in time. In detail, if the presynaptic neuron fires slightly *before* the postsynaptic neuron, it means the presynaptic spike contributes to the postsynaptic firing. Therefore the connecting weight between these two neurons is enhanced (the synapse is *potentiated*). If the presynaptic neuron fires just *after* the postsynaptic firing, it means the presynaptic spike has no influence on the postsynaptic neuron, then a decrease of the weight will occur (the synapse is *depressed*). If the two spikes are too distant, the weight remains unchanged. The standard STDP learning window is formulated as:

$$W(\Delta t) = \begin{cases} A_+ e^{-\frac{\Delta t}{\tau_+}} & \Delta t > 0, \\ A_- e^{\frac{\Delta t}{\tau_-}} & \Delta t < 0. \end{cases} \tag{3.9}$$

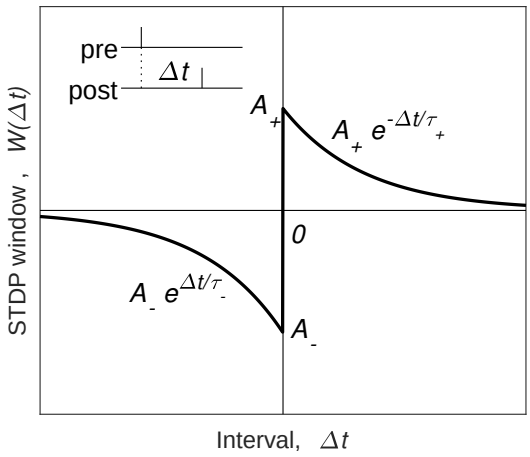


Figure 3.6: STDP window

where A_+ and A_- are scaling constants of strength of potentiation and depression; τ_+ and τ_- represent the time decay constants; Δt is the time difference between pre- and post-synaptic firing timings:

$$\Delta t = t_{post} - t_{pre} \tag{3.10}$$

In `eSpinn` I have introduced a rate-based Hebbian model derived from the nearest neighbour STDP implementation [143], with two additional evolvable parameters k_m and k_c:

$$\dot{w} = u_i(\frac{A_+}{\tau_+^{-1} + u_i} + \frac{k_m(u_j - u_i + k_c) + A_-}{\tau_-^{-1} + u_i}) \tag{3.11}$$

Details of this local learning rule are discussed in Section 6.5.2 of Chapter 6.

3.4 Neuroevolution

While gradient methods have been very successful in training traditional MLPs [107], their implementations on SNNs are not as straightforward because they require the availability of gradient information. Instead, `eSpinn` has developed its own version of a popular neuroevolution approach – NEAT [140], which can accommodate different network implementations and integrate with Hebbian plasticity, as the method to learn the best network controller.

3.4.1 Evolutionary Algorithms

A general scheme of evolutionary algorithms (EAs) is given in Algorithm 1. EAs [137] have become an important set of algorithms in training neural networks, especially in cases where training reference is not immediately available and thus gradient methods are not applicable. Instead, performance evaluation in EAs are represented as a fitness function. Generally in neuroevolution, an encode mechanism is utilised to encode the connection weights and the search for optimal parameters is carried out using stochastic variation operations in a population that consists of a number of individuals. Such methods are likely to converge around one single solution as evolution goes on – a phenomenon known as *genetic drift* [137], which can lead to premature convergence and consequently getting stuck at a local optimum, because the population may quickly converge on whatever network that happens to perform best in the initial population. To address this issue, a population management approach is desired where network diversities can be maintained during the evolution. In this work a modified NEAT algorithm is employed, which is discussed in the next sections.

3.4.2 NEAT

NeuroEvolution of Augmenting Topologies (NEAT) is a powerful evolutionary approach for neural network learning which evolves network topologies along with connection weights. The efficacy of NEAT is guaranteed by: (i) historical markings to solve the variable-length genome problem; (ii) speciation to protect innovation and preserve network diversity, to avoid premature convergence; and (iii) incremental structural growth to avoid troublesome hand design of network topology.

Typical EAs are difficult to genetically crossover neural networks with variant topologies, because they can only operate within fixed-sized genome space. Recombination of divergent genomes tends to produce damaged offspring. However, similar network solutions sharing similar functionalities can be encoded using completely different genomes – a phenomenon known as the *Competing Convention Problem* [140]. To address this problem, NEAT uses historical markings which act as artificial evidence to track the origin of genes. When two genes share the same historical marking, they are categorised as alleles. Therefore, NEAT can match up genomes representing similar network structures and allow mating in a rationale manner.

NEAT also uses an explicit fitness sharing scheme [137] as a population management approach to preserve network multimodality. Historical markings are used as a measurement of the genetic similarity of network topolo-

gies, based on which, genomes are speciated into different *niches* (also termed *species*). Individuals clustered into the same species will share their fitness score together [137]. The fitness of each individual is scaled according to the number of individuals falling in the niche. As competition within the species becomes more intense, solutions will have a lower fitness. Thus, the sparsely populated species will become more attractive, avoiding any one taking over. Therefore, innovations will be protected within niches to have time to optimise.

Finally, an incremental growth mechanism is used in NEAT to discover the least complex effective neural topology, by beginning searching minimal network structure and gradually expanding to more complex networks during evolution.

The NEAT algorithm has been successfully applied to different control problems [140, 211, 69]. However, implementation of NEAT has been restricted to traditional neural systems, with one exception [138] targeting spiking networks.

3.4.3 MoNEAT

NEAT conventionally begins with minimal structure, i.e. inputs are directly connected to the outputs with no hidden-layer neurons. As evolution carries on, NEAT will expand network topologies through mutations. Two topological mutation operators are proposed in NEAT to introduce innovation to the population – an 'AddNode' mutation and an 'AddConnection' mutation. As shown in Fig. 3.7, the 'AddNode' mutation (a) will insert a node to split an existing connection into two; the 'AddConnection' mutation (b) will add a new connection to link two existing nodes that are currently not connected.

These methods have been exceptional in considerably simple cases. However, in practice, it can be slow to evolve optimal structures for complex problems because of the use of gradual topological expansion. In addition, during our experiments, these mutations tends to generate deep networks that have long chains of neurons, which is actually undesired. It will possibly expand the search to unwanted space and consequently the process will be stuck at local optima. A bypass to avoid this is to begin the search with a population of intermediate topologies, either from manual design which requires human experience [212], or via incremental evolution [213] in which the networks solve a downgraded task first and take the intermediate population as a starting point for further evolution to solve the harder task.

Therefore, `eSpinn` complements the original NEAT with an additional mutation method so that the population can quickly expands to more complex structures. Along with the original mutation operators, MoNEAT [214] also

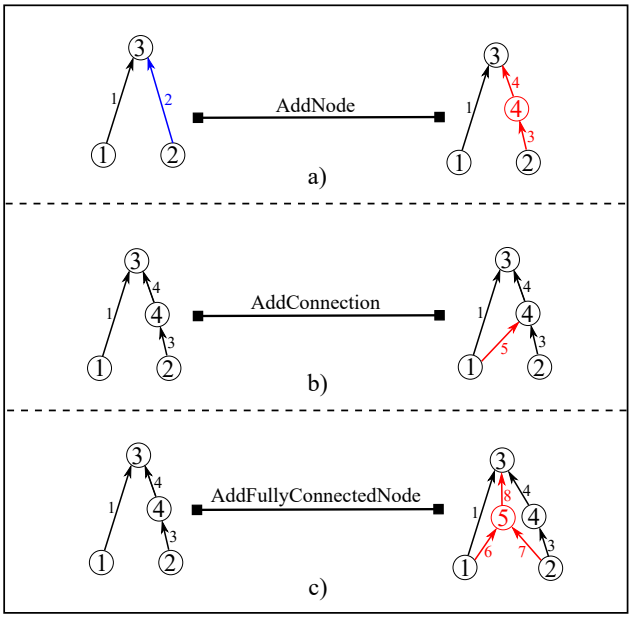


Figure 3.7: Topological mutations in MoNEAT. The "AddNode" mutation inserts a node to split an existing connection into two. The "AddConnection" mutation adds a new connection between two existing neurons that were previously unconnected. The "AddFullyConnectedNode" mutation adds a neuron that is fully connected from each input and to each output. In these operations, newly created neurons and connections will be assigned the next available node IDs and innovation numbers which are traced globally.

preserves the possibility to discover minimal effective structures. The mutation I propose in MoNEAT is an 'AddFullyConnectedNode' method (Fig. 3.7-c) – a neuron is added to connect itself from each input neuron and to each output neuron. These new connections will be assigned a weight of 0, which will circumvent the initial effect of the mutation – the newly created network will have the same functionality as its parent, with an expanded searching area. By using speciation the innovation can be preserved and the child will then have enough time to evolve its new structure.

Finally, `eSpinn` integrates the newly proposed mutation and keeps track of all the innovations (i.e. structural variations) globally. Whenever an innovation occurs, it can be traced to see whether it has already existed. This mechanism will ensure networks with the same topological growth will be assigned the same innovation numbers, which is essential during the process of network structural expansion.

3.5 Examples Using eSpinn

3.5.1 MoNEAT Evaluation

I first evaluate the performance of MoNEAT with a simple feedback control task for a non-linear plant model. The task is to drive the plant to follow a reference signal with its feedback measurements as inputs, shown in Fig. 3.8. The entire problem is written in C++ to speed up the evaluations.

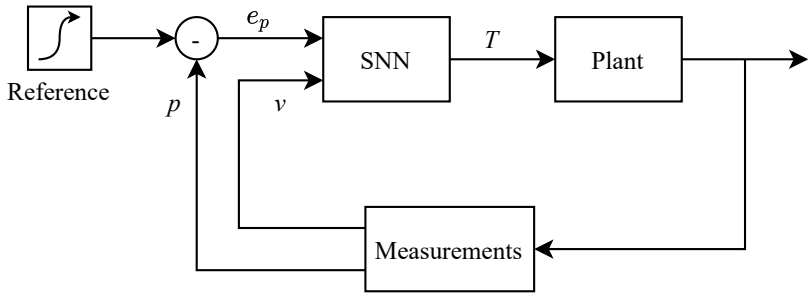


Figure 3.8: Diagram of the feedback control simulation. The SNN controller takes two states as inputs: the position error e_p and current velocity v, and learns to generate a control command T that will be fed to the plant model.

The plant model is described in (3.12), which is a general linearised approximation of the heave dynamics of a hexacopter [212]. Acceleration of the plant a is presented as a linear combination of the controller command T and velocity v. v on the other hand is the integral of the acceleration, and finally

position p is the integral of v:

$$\begin{aligned} a &= k_T\, T + k_v\, v + b \\ v &= \int a\, dt \\ p &= \int v\, dt \end{aligned} \tag{3.12}$$

where k_T, k_v and b are constant coefficients. The controller takes the position error e_p and current velocity v as inputs, and learns to generate a control command T.

In this task a population is initialised with 150 networks that consist of hybrid neuron models where: the hidden-layer neurons are Izhikevich spiking neurons; and the output neuron is a sigmoidal activation unit that takes a weighted sum of outputs from the hidden neurons and the bias. During evaluation, networks in the population will be activated one by one to drive the plant model, and given a fitness value based on the system output error. Ten runs of the task have been carried out using MoNEAT and NEAT respectively. Each run lasts for 50 generations. Evolutionary parameters of the two approaches are identical, except in MoNEAT the "AddFullyConnectedNode" mutation is enabled with a probability of 0.005. The probabilities of the "AddNode" and "AddConnection" mutations are 0.01 and 0.02 respectively in both approaches. Table 3.1 shows the averaged best fitness values and the standard errors among the 10 runs. The MoNEAT algorithm is statistically superior to the original NEAT[3].

Table 3.1: Best Fitness Values of MoNEAT vs. NEAT

Fitness	**MoNEAT**	**NEAT**
Mean	0.93824	0.92721
Std. Error	0.00634	0.01033

3.5.2 Flappy Bird

Games have been an effective and interesting way to study computational intelligence algorithms. In `eSpinn` I have presented a Flappy Bird game implementation as a testbed for the proposed algorithm.

[3]based on the Mann–Whitney U test

The Flappy Bird game is a classic side-scrolling game[4] that was prevalent in 2014. The goal of the game is to fly a bird through gaps between columns of pipes and avoid collisions for as long as possible. This game has become a popular benchmark problem in the development of ML algorithms, where a variety of reinforcement and evolutionary learning techniques have been applied [215].

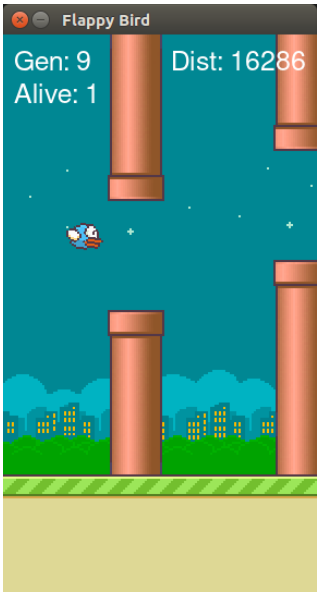


Figure 3.9: A successful AI Flappy Bird engined by the `eSpinn` package

In this work the game is implemented in Python. The physics of the game is rather simple to understand: the bird will jump upwards whenever the user taps the screen and descend otherwise. As shown in (3.13), v^y is used to represent the bird's vertical speed. The Y axis is heading down so a positive v^y means the bird is descending. Velocity will increase by a steady acceleration a^y until reaching the maximum v^m and will be set to a negative flap speed v^f when the bird flaps.

$$v_t^y = \begin{cases} v^f & \text{, if (flapped)} \\ v_{t-1}^y + a^y \, \Delta t \text{ until } v^m & \text{, otherwise} \end{cases} \tag{3.13}$$

Each bird takes two known states as inputs: the horizontal and vertical distance between the bird and the gap ahead. An additional bias neuron is connected to the output neuron. The bird controller is constructed with saturated linear units as inputs, and LIF spiking neurons in the hidden (if

[4] `https://en.wikipedia.org/wiki/Side-scrolling_video_game`

any) and output layer. The bird will flap if a spike is fired in the output neuron.

At the beginning, a population of 50 birds will be initialised and evaluated simultaneously. Fitness is defined as the steps the bird stays alive. When all birds die out, MoNEAT will be called and a new population of 50 child birds are formed from the best parents. The game will then restart and the new population will be evaluated. Evolution will be terminated when the champion bird successfully travels a distance of 10,000 steps.

During early stages of training the birds will die out easily, either crashed to the ground or the pipes. After some generations there will be some (or one) outperform the rest and they will start to multiply. Finally one or several of their offspring will successfully accomplish the task. The evolution largely takes around a dozen generations to find out a successful bird. The smallest functional network turns out to be quite simple – inputs are directly connected to the output with 0 hidden neurons.

3.5.3 Pole Balancing

Finally, the proposed SNN controllers are evaluated using a classic non-linear control benchmark – the pole balancing problem (also known as the inverted pendulum problem). The performance is compared with conventional sigmoidal networks. The pole balancing problem is not only inherently unstable, but also capable of varying degrees of complexity by limiting the state variables provided to the controller, which makes it ideal for designing and testing nonlinear control methods.

Previous attempts to solve this problem using SNNs with fixed-topology can be found in [216, 217]. In this section, a different approach is taken using both the NEAT and MoNEAT algorithm to evolve network topologies and connection weights. The purposes of this experiment are twofold:

- To benchmark spiking controllers against the original sigmoidal counterpart. Therefore, the same evolutionary strategy (i.e. NEAT) is first used in both type of networks: only the NEAT mutation components (i.e. AddConnection & AddNode mutation) are activated to introduce innovations; recurrence is only allowed within hidden layers.

- To compare MoNEAT with the original NEAT algorithm, i.e., how is MoNEAT performing comparing with NEAT? In this task spiking controllers are evolved using both methods and their performances are compared against each other.

The original NEAT C++ source code is available publicly[5]. All the experiments are programmed in C++ and performance analysis is carried out using MATLAB.

Benchmark Problem

Fig. 3.10 shows the cart-pole system to be controlled, which consists of a cart that can move left or right within a bounded one-dimensional track, and a pole that is hinged to the cart. The problem is to balance the pole upright for as long as possible by applying a force F_t to the cart parallel to the track. The system has four state variables:

x – cart position

θ – pole angle

$\dot{x}$ – cart velocity

$\dot{\theta}$ – pole angular velocity

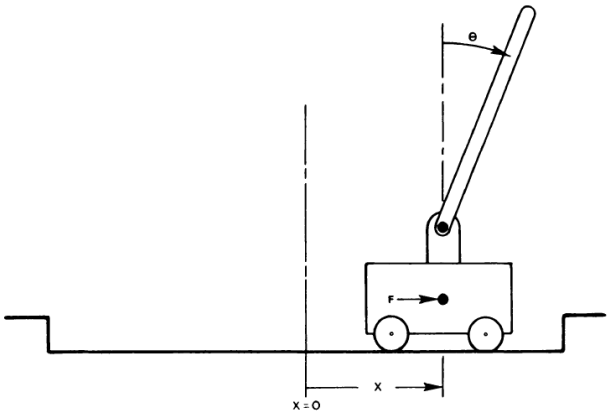


Figure 3.10: Cart-pole system, taken from [218].

For simplicity, we neglect the friction acting on the cart from the track and that of the pole on the cart. The system is then formulated by two nonlinear differential equations [218]:

[5] `http://nn.cs.utexas.edu/?neat-c`

$$\ddot{\theta}_t = \frac{g\sin\theta + \cos\theta(\frac{-F_t - m_p l\dot{\theta}^2\sin\theta}{m_c + m_p})}{l(\frac{4}{3} - \frac{m_p\cos^2\theta}{m_c + m_p})} \tag{3.14}$$

$$\ddot{x}_t = \frac{F_t + m_p l(\dot{\theta}^2\sin\theta - \ddot{\theta}\cos\theta)}{m_c + m_p} \tag{3.15}$$

where $g = 9.8$ m/s^2 denotes the gravitational acceleration; $m_c = 1.0$ kg denotes the mass of cart and $m_p = 0.1$ kg denotes the mass of pole; $l = 0.5$ m is half length of the pole.

The discrete form of state variables are updated following:

$$\begin{cases} x_{t+1} &= x_t + \tau\dot{x}_t \\ \dot{x}_{t+1} &= \dot{x}_t + \tau\ddot{x}_t \\ \theta_{t+1} &= \theta_t + \tau\dot{\theta}_t \\ \dot{\theta}_{t+1} &= \dot{\theta}_t + \tau\ddot{\theta}_t \end{cases} \tag{3.16}$$

with τ representing the time step.

A force generated by the spiking controller at each time step will be used to update the state variables following (3.14), (3.15) and (3.16). A failure signal is generated when the cart reaches the track boundary, which is ± 2.4 meters from the track centre, or if the pole tilts beyond the failure angle, which is ± 12 degrees (or around 0.21 radian) from the vertical.

Experimental Setup

I first start with the basic balancing task with complete state variables. This Markovian problem can act as a base performance measurement before going to the more challenging non-Markovian version without velocity information. In both tasks the cart-pole system model is unknown to the spiking controller.

The controller contains a population of 150 networks, which will be evolved using the NEAT algorithm. Per epoch, the champion of each species is duplicated when the number of networks in that species is larger than 5. The best 20% of networks in each species are allowed to reproduce, after which, all parents are discarded and the remaining 150 offspring will form the next generation.

Each network will be evaluated and assigned a fitness value. Fitness is defined as the number of time steps that the balanced criteria is not violated. Otherwise a failure signal is generated and evaluation is moved on to the next network.

Pole Balancing with Velocity

The first task is to balance the pole with velocity information. Evolution begins with minimal structure. At initialisation, each network consists of five input nodes and one output node. A bias neuron and four encoding neurons each receiving one state variable. Each input neuron including the bias is connected to the output node. Input neurons will convert the measured data into currents to be injected to spiking neurons. This is a common practice which is called the 'current coding' method. In this task, hidden (if any) and output neurons are configured as LIF neurons. A binary force ($F_t = \pm 10$ Newtons) is determined based on the output neuron spike status, which will then be applied to the cart at each time step. The time step τ in (3.16) is set to 0.01 seconds.

During evolution, hidden nodes are allowed to be added with a probability of 0.03. Connections are added with a probability of 0.05. The 'AddFully-ConnectedNode' mutation is disabled to provide benchmark comparison with sigmoidal networks using the original NEAT package. Although mutations are enabled, the minimal structure is already able to solve this simple task. A successful solution is identified when it is able to balance the pole for 100,000 time steps, which is equivalent to around 15 minutes of simulated time.

NEAT is applied to evolve both SNNs and sigmoidal networks. Each test is run for 60 episodes. Table 3.2 shows a summary of fewest generations needed to complete the task. A failure run means the controller fails to find a solution within 50 generations. We can see both approaches are able to find out successful solutions in all 60 runs, but it is clear that the spiking controller takes fewer generations to solve the task. Example runs of both tests are shown in Fig. 3.11 and Fig. 3.12.

Table 3.2: Fewest Generations Required to Find A Successful Solution for Markovian Pole Balancing Problem

	Best	Worst	Median	Mean	Failure Rate
NEAT-sigmoidal	4	32	8	9.6	0/60
NEAT-SNN	1	7	3	3.28	0/60

Pole Balancing without Velocity

Our main task is to balance the pole without velocity information. Instead of using bang-bang control, a continuous output force within [-10, 10] Newtons

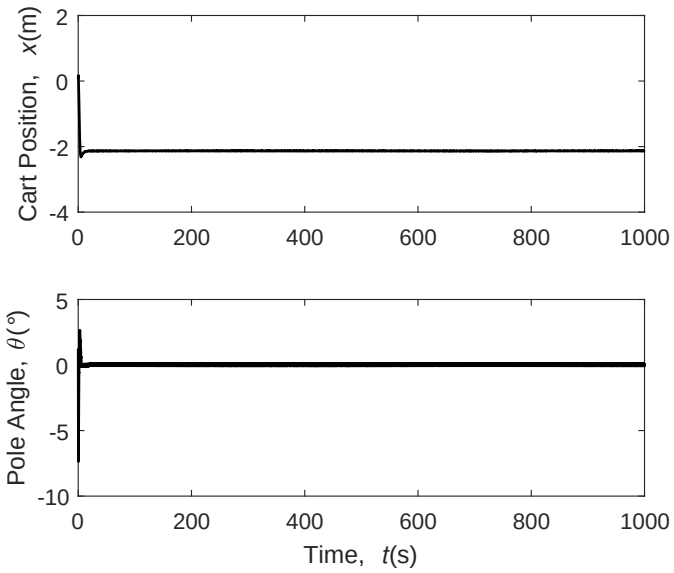


Figure 3.11: NEAT-sigmoidal: cart position x and pole angle θ of the pole balancing task with velocity information.

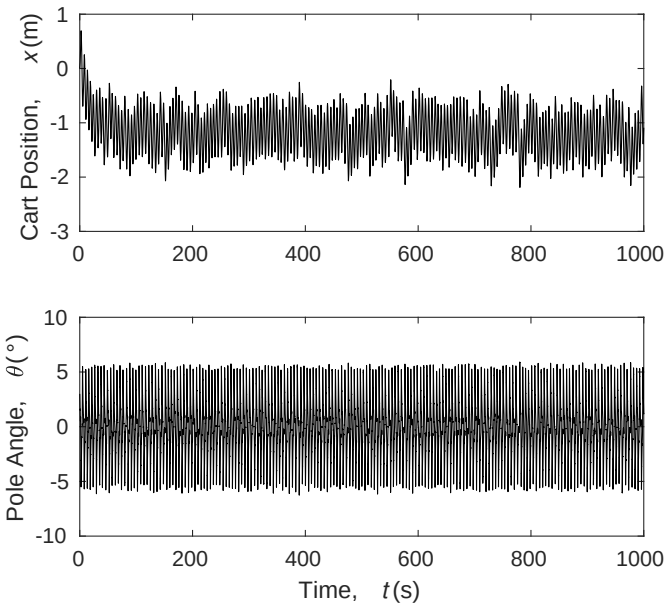


Figure 3.12: NEAT-SNN: cart position x and pole angle θ of the pole balancing task with velocity information.

is used for this more challenging problem. A sigmoidal neuron is used as the network output, in which a normalised force F_n is calculated based on a weighted sum of firing rates from connected neurons following a modified sigmoid function. F_n is then scaled and shifted to generate the force F_t:

$$F_n = \frac{1}{1 + \exp(-\sigma \sum w_i r_i)} \tag{3.17}$$

$$F_t = 10(2F_n - 1) \tag{3.18}$$

where σ is a positive decay variable, which will be automatically tuned during evolution; r_i denotes the firing rate of the i^{th} connected neuron and w_i denotes the corresponding connection weight.

Note NEAT use a compatibility distance function δ to determine the similarity of network solutions. When the distance between any two individuals is smaller than a threshold δ_t, they are categorised into the same species. The compatibility distance δ is defined as:

$$\delta = \frac{c_1 E}{N} + \frac{c_2 D}{N} + c_3 \overline{W} \tag{3.19}$$

where E and D denote the number of *excess* and *disjoint* genes; $\overline{W}$ denotes the average connection weight difference; N is the total number of genes; c_1, c_2 and c_3 are user-defined coefficients for altering the significance of these factors. In this task a component of σ difference (denoted as D_σ) is added to the original compatibility distance:

$$\delta = \frac{c_1 E}{N} + \frac{c_2 D}{N} + c_3 \overline{W} + c_4 D_\sigma \tag{3.20}$$

At initialisation, the cart position is randomised between [-0.3, 0.3] metres, the pole angle is set between [-3.0, 3.0] degrees, while the cart velocity and the pole's angular rate are set to 0. Time step τ is set to 0.01 seconds. Successful solutions are dictated if the pole is balanced for 5000 time steps.

Similarly, NEAT is applied to the spiking controller and the results are compared against the original NEAT. Connections and nodes will be added based on a probability of 0.05 and 0.03. Recurrence is only allowed within the hidden spiking neurons[6].

A summary is shown in Table 3.3. The proposed spiking controller is essentially better than sigmoidal networks in solving this problem. It requires fewer generations to find a functional solution. It also has a lower failure rate over 20 runs. It is clear that SNNs are more able to solve this non-Markovian problem. Further, the Mann-Whitney U-test is used to assess the statistic difference between the two sets of samples. The result shows that the spiking controller has significantly better performance at $p < 0.01$.

[6]Recurrent connections in the output layer are allowed in the original NEAT package.

Table 3.3: Fewest Generations Required to Find A Successful Solution for Non-Markovian Pole Balancing Problem

	Best	Worst	Median[a]	Mean[a]	Failure Rate
NEAT-sigmoidal	2	45	20.5	22.7	2/20
NEAT-SNN	2	30	3	4.3	0/20

[a]Values are calculated assuming failure runs take 51 generations.

To visualise the evolution progress, the best networks' fitness values are averaged over the 20 runs. Fig. 3.13 shows the mean and standard deviation of fitness values of both tests at successive generations. In the beginning, only some of the networks can find a path to optimization, thus introducing a large fitness deviation. As evolution goes on, individuals with higher fitness values will gradually take over the entire population. Finally a successful network will be identified.

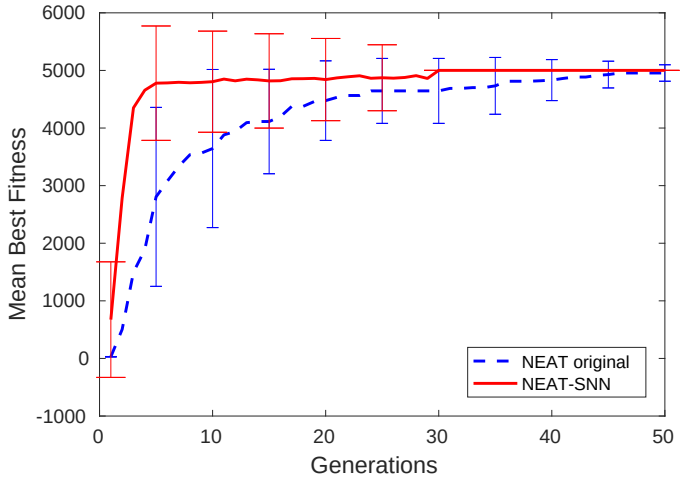


Figure 3.13: Best networks' mean fitness values in progress over 20 runs, with error bars denoting the standard deviation.

A further look is taken at how well both types of networks can solve the non-Markovian task if we let the evolution continues. It turns out without output node recurrence, the sigmoidal networks are only able to balance the pole for around 9,000 steps. On the other hand, the SNNs can easily hold the pole upright for a longer time period – feasible spiking controllers can still be found even if the successful step is set to 100,000.

Fig. 3.14 and Fig. 3.15 show the status of cart position, pole angle and the output force applied to the cart during a successful run. In both approaches, all these three variables are oscillating around some certain points. However, it is clear that in the SNN approach the pole is very likely to maintain balanced even if the task continues.

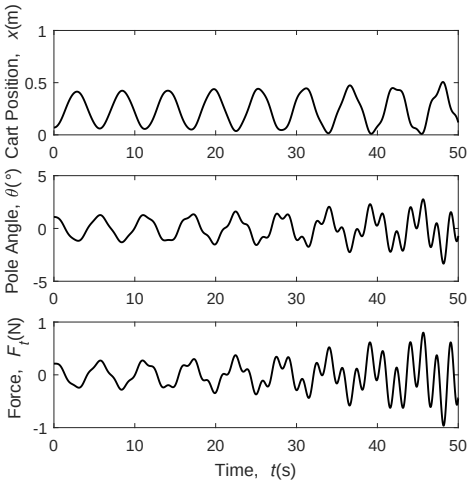


Figure 3.14: NEAT-sigmoidal: cart position x, pole angle θ and force F_t applied to the cart of the non-Markovian pole balancing task (without velocity information).

Now let us take a look at how a feasible solution looks like. Fig. 3.16 illustrates an evolved effective network topology that is able to balance the pole for as long as 100,000 steps. Direct connection from the cart position to the output is deactivated. A hidden node is instead inserted in between – output force is less desired to be affected by the cart position. On the other hand, the connection from the pole angle is much potentiated. Interestingly, this topology can solve the problem even without recurrent connections to calculate the derivatives. The membrane potential integration process can be conceived as a way to share weights over time. Such dynamics are shown to be able to yield more powerful computation than conventional sigmoidal networks.

MoNEAT vs. NEAT: Pole Balancing without Velocity

Finally, a comparison is made between MoNEAT and NEAT. Spiking controllers are evolved using both methods to balance the pole for a run of 100,000 steps. Statistics over 20 runs are shown in Table. 3.4. The MoNEAT approach is essentially faster to discover functional solutions than NEAT. Among 20

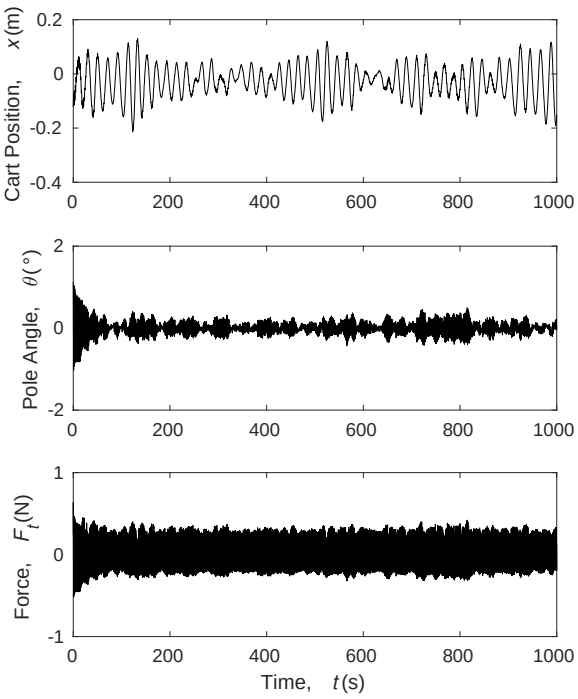


Figure 3.15: NEAT-SNN: cart position x, pole angle θ and force F_t applied to the cart of the non-Markovian pole balancing task (without velocity information).

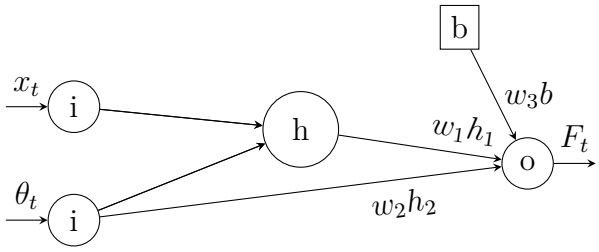


Figure 3.16: Effective spiking network topology that is able to balance the pole for up to 100,000 steps. Network inputs consist of cart position x_t and pole angle θ_t. Output force F_t is calculated based on a weighted sum of firing rates $\sum w_i r_i$, in which r_i denotes the firing rate of the i^{th} connected neuron and w_i denotes the corresponding connection weight.

runs of MoNEAT tests only one instance failed to find out a successful solution in 50 generations, while the NEAT approach has failed 16 runs. As anticipated, effective MoNEAT solutions are more likely to be more complex in structure than NEAT solutions, because MoNEAT does not necessarily guarantee the *minimal* network structure.

Table 3.4: Fewest Generations Required to Find A Successful Solution for Non-Markovian Pole Balancing Problem Using NEAT and MoNEAT

	Best	Worst	Median[a]	Mean[a]	Failure Rate
NEAT-SNN	28	43	51	47.9	16/20
MoNEAT-SNN	12	40	20	22.75	1/20

[a]Values are calculated assuming failure runs take 51 generations.

A total of 100 runs per each method have been conducted across various successful steps to provide more detailed statistics. Fig. 3.17 shows the generations in both methods that are required to identify a successful SNN to be able to balance the pole. Each scenario is repeated 20 runs.

3.6 Discussion

The design of functional SNNs is considered to be difficult, because SNNs behave as complex systems with transient dynamics [126]. In this work, we present a general network structure that derives strength from topology evolution, which has demonstrated to be a powerful tool in solving non-linear problems. The potential integration dynamics of spiking neurons are essentially different from conventional ANNs, which have provided additional computational power apart from the proposed recurrent network topology.

The philosophy behind NEAT is that by expanding network topology starting from a minimal structure, evolution will be faster in the search for functional network compositions. Following this idea, MoNEAT is developed to quickly escape away from less favourable search areas, at the cost of possible missing discovery of the *minimal* effective structure. However, in a small-sized network implementation, the negative effect is negligible.

In addition, I believe adaptive probabilities of mutation methods can offer extra flexibility in the search for structural variations. We can base the probabilities on the network's fitness value, and decrease the probabilities if

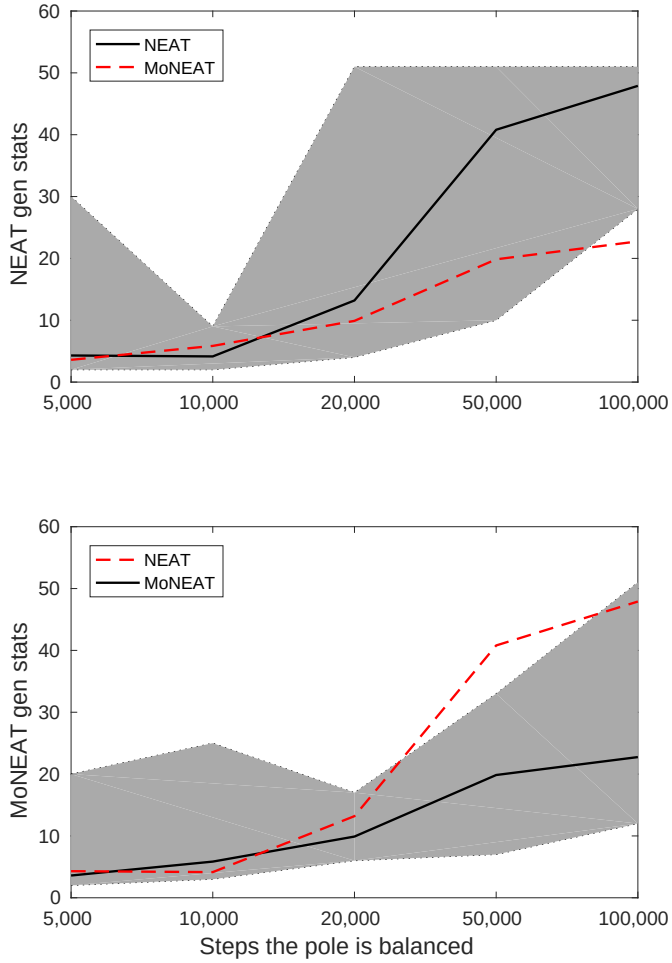


Figure 3.17: Generations required to find out a successful SNN to balance the pole for various steps using both NEAT and MoNEAT. The grey shades represent the areas between the minimal and maximum generations across 20 repeats. The thick lines represent the averaged generations calculated from the 20 runs.

mutations are less likely desired, i.e. when evolution is close to finding feasible solutions.

3.7 Conclusion

This chapter presents an SNN machine learning package targeted for embedded control applications, with support for MATLAB and Python simulations as well as seamless operations on embedded hardware. Through a set of experiments, I have demonstrated that SNNs are powerful computational systems that can solve nonlinear control problems significantly better than traditional sigmoidal networks. Sufficient evidence has shown that the `eSpinn` SNN implementation with MoNEAT can be a general problem-solving framework. I hope the library can contribute to the computational intelligence community and benefit researchers and developers in these areas for other applications.

Chapter 4

System Overview

4.1 Introduction

This chapter provides an overview of the platform and systems used to conduct actual flight experiments using SNN. A hexacopter UAV is employed as the platform to test the SNN control algorithms. Sensor data used as input to the controllers is acquired from an indoor Vicon motion capture system (MCS) and an onboard inertial-measurement unit (IMU). Prototyping of the controller is implemented in the Robot Operating System (ROS) environment, in which message communication can be established wirelessly between the UAV and the ground control station (GCS). Each of these system components is described in this chapter (Section. 4.2–4.5). In addition, this chapter will also describe the control system of the hexacopter platform, including the controller architecture (Section. 4.6) and inner loop dynamics (Section. 4.7).

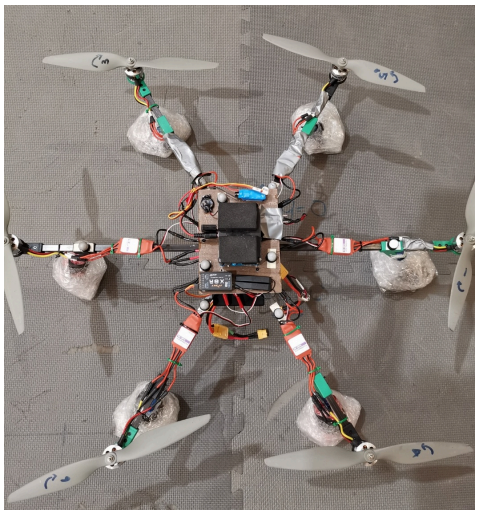
4.2 Hexacopter Platform

Multirotor UAVs can operate in a wider range of environments than fixed-wing aircraft because of their VTOL (vertical take-off and landing) characteristics, enabling them to hover and take off in confined spaces. They are able to perform more aggressive manoeuvres but are naturally less stable, which requires a well-tuned flight controller to achieve great flight performance [8].

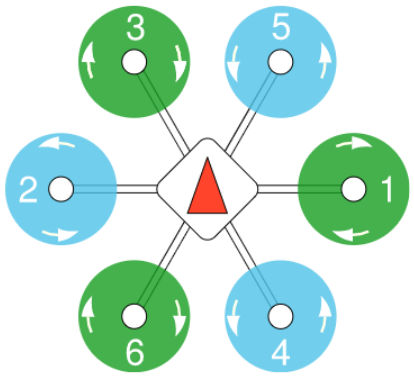
4.2.1 Hardware

Common multirotor frames are tricopter, quadcopter, hexacopter and octocopter. Irrespective of the type, a multirotor platform usually includes a

body frame, an onboard flight controller, electronic speed controllers (ESCs), motors, propellers and a battery. The experimental platform used in this work is a hexacopter UAV shown in Fig. 4.1a, which employs the X geometry airframe, as in Fig. 4.1b.



(a) Hexacopter assembled



(b) Hexacopter X geometry airframe, from `https://docs.px4.io/master/en/airframes/airframe_reference.html`

Figure 4.1: Hexacopter UAV platform used in this thesis

The body frame is a RotorBits Hexacopter, a commercial modular assembly multirotor kit[1]. A Pixhawk 2 Cube autopilot system[2] is fitted on top of the body. An additional onboard companion computer (Odroid XU4[3]) is used as a higher level flight controller that is needed to provide sufficient computation capability. The Odroid is interfaced with the Pixhawk via a USB to UART RS232 adapter, which is wired to the Pixhawk `TELEM2` port. It is also equipped with a NetGear WNA3100M mini WiFi adapter to establish wireless connections to the ground control stations (GCSs) and a Desktop computer running the Vicon Tracker MCS software (refer to Section 4.3). An FrSky X8R telemetry receiver is connected to the Pixhawk to receive control commands from the radio controller, which in this setup is an FrSky 2.4GHz Taranis X9D Plus RC Transmitter. The onboard controller configuration and telemetry setup are shown in Fig. 4.2.

The hexacopter is equipped with six 11×4.5 propellers, which are powered by Turnigy Aerodrive DST-1200 brushless DC (BLDC) motors controlled by

[1] `https://hobbyking.com/en_us/rotorbits-hexcopter-kit-with-modular-assembly-system-kit.html`

[2] `https://docs.px4.io/master/en/flight_controller/pixhawk-2.html`

[3] `https://wiki.odroid.com/odroid-xu4/odroid-xu4`

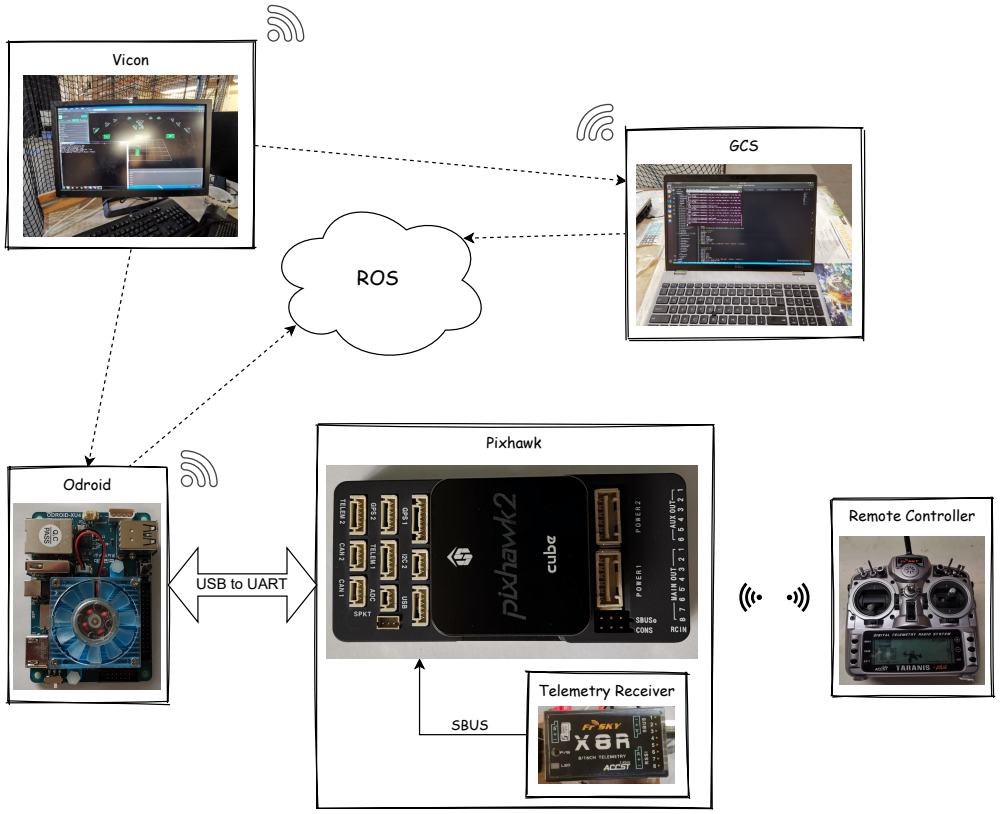
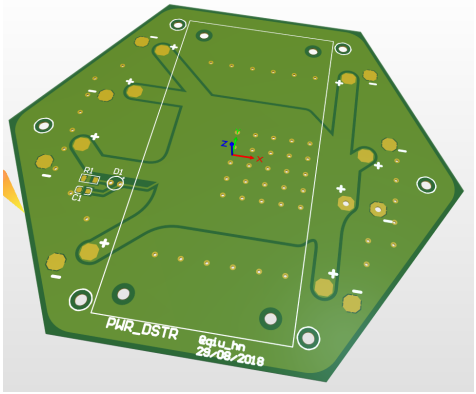


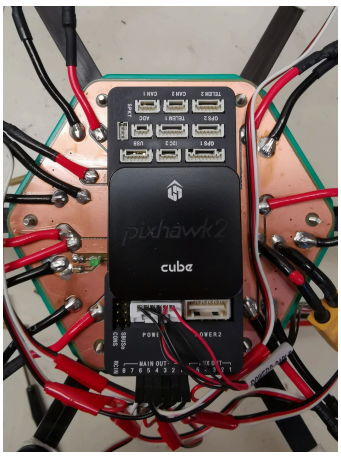
Figure 4.2: Hexacopter onboard controllers & telemetry radio setup. The Pixhawk 2 Cube is the lower-level flight controller which essentially is a MIMO control signal mixer decoding the control commands to PWM outputs for each of the rotors. It is able to receive movement control channels from the radio controller via the X8R telemetry receiver. Furthermore, it is interfaced with an Odroid XU4 higher-level controller via a USB to serial port adapter. The Odroid is responsible for attitude control, position control as well as other high-level motion planning tasks (e.g., trajectory planning, obstacle avoidance). It is equipped with a WiFi module to receive real-time flight data from the Vicon system. It is also able to communicate with the GCS via the ROS network (refer to Section 4.4).

Turnigy Plush 30A Electronic Speed Controllers (ESCs). A Turnigy 5000mAh 3S Lipo battery is used to power up the aforementioned on-board electronics.

A single-layer power distribution board (Fig. 4.3) is manufactured to distribute the battery power to on-board electronics. An LED is added to indicate the power connection. Copper trace width to ESCs is set as 5 mm with a thickness of 1 oz/ft^2, which is able to provide sufficient capacity for large current loads of up to $8A$. I have also 3D-printed a battery case to hold the battery in a fixed position (Fig. 4.4). The container is attached to the bottom of the hex center mount, so that the drone's center of gravity can stay as low as possible.

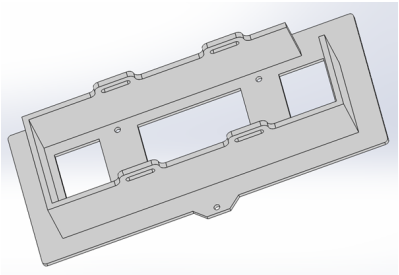


(a) 3D view of the power distribution board

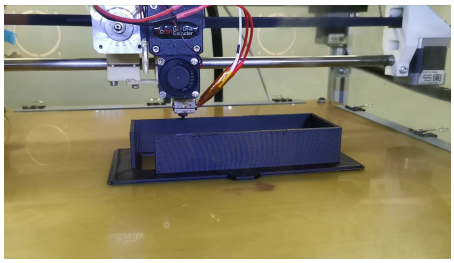


(b) Power board assembled with the Pixhawk flight controller

Figure 4.3: Hexacopter power distribution board



(a) 3D view of the battery case



(b) 3D printing of battery case

Figure 4.4: Battery case

4.2.2 Autopilot System

The Pixhawk Cube flight controller (shown in Fig. 4.3b) is selected as the lower-level flight controller. It is responsible for control signal mixing, which essentially is a MIMO (multiple-input multiple-output) device that mixes the movement control commands (i.e., roll, pitch, yaw and throttle) to calculate the pulse width modulation (PWM) outputs for each of the rotors. It also acts as a signal router that passes on signal channels from the radio controller to the Odroid.

Specifications

Pixhawk is an open-hardware framework[4] which has hardware designs which are publicly available. Cube is based on the FMUv3 hardware design and runs the PX4 Autopilot firmware. It is featured with a 32-bit STM32F427 SoC and a STM32F103 fail-safe co-processor. It also has three sets of IMU sensors, 14 PWM servo outputs and abundant peripherals.

PX4 Autopilot[5] is an open source flight control software package released under the BSD-3-Clause License. Firmware installation and vehicle setup and calibration are carried out using the QGroundControl Desktop[6].

Signal Mixing

Movement control commands will be translated to actuator commands in Pixhawk that determine the PWM outputs controlling the servo motors. This MIMO mixer has 4 input channels and 6 actuator outputs corresponding to each rotor. The attitude control commands (i.e., roll, pitch and yaw) range from -1.0 to 1.0, while the throttle command is in the range of [0, 1.0]. On the other hand, the output group (actuator commands) ranges from -1.0 to 1.0.

IMU

The IMU in the Pixhawk Cube flight controller consists of a 3-axis gyroscope, a 3-axis accelerometer and a magnetic compass, which is able to provide extra redundancy to measure linear and angular rates of the body. In this work, the linear motion will be captured using the Vicon system (detailed in the following Section 4.3). The IMU is used to collect the orientation information (specifically roll and pitch only). Orientation received from the IMU is in

[4] `https://github.com/pixhawk/Hardware`
[5] `https://github.com/PX4/PX4-Autopilot`
[6] `https://github.com/mavlink/qgroundcontrol`

the form of a quaternion, which is described as $q = [q_0 \quad q_1 \quad q_2 \quad q_3]^T$. The quaternion is subsequently converted into Euler angles, i.e., roll (ϕ), pitch (θ) and yaw (ψ), using the following transformation:

$$\begin{bmatrix} \phi \\ \theta \\ \psi \end{bmatrix} = \begin{bmatrix} atan2(\,2(q_0q_1 + q_2q_3),\; {q_0}^2 - {q_1}^2 - {q_2}^2 + {q_3}^2) \\ asin(2(q_0q_2 - q_3q_1)) \\ atan2(\,2(q_0q_3 + q_1q_2),\; {q_0}^2 + {q_1}^2 - {q_2}^2 - {q_3}^2) \end{bmatrix} \tag{4.1}$$

4.2.3 Companion Computer

The onboard companion Odroid-XU4 is used for higher-level control, which is based on a cascaded control architecture that includes attitude control at the innermost layer, position control and trajectory planning in the outer. The outputs of the Odroid are control commands of movement channels (roll, pitch, yaw and throttle) that will be passed through the USB-to-UART adapter to the Pixhawk.

Specifications

Odroid-XU4 (Fig. 4.5) is a small yet powerful single-board computer that supports Ubuntu and Android systems. It utilises two CPUs: an ARM Cortex-A15 Quad core CPU (up to 2.0GHz) and a Cortex-A7 Quad core CPU (up to 1.4GHz), which is able to provide additional processing capability. Provided with a 32GB eMMC memory module for system storage, the computer can achieve high speed application launching and data transfer.

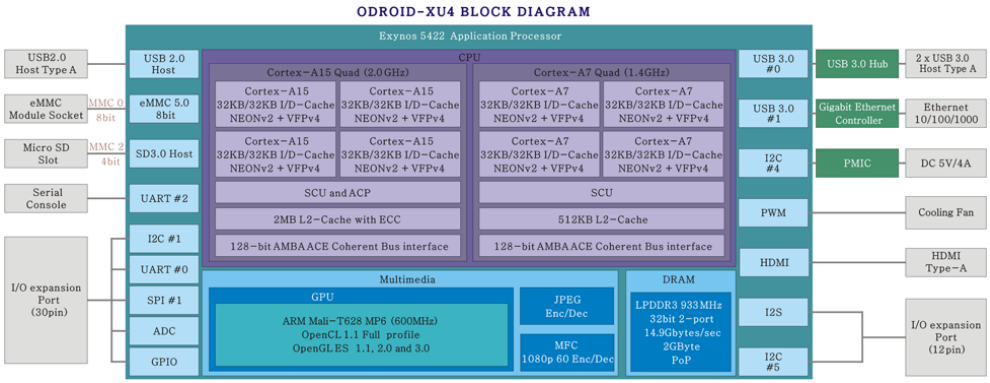


Figure 4.5: Odroid-XU4 specifications, from `https://wiki.odroid.com/odroid-xu4/odroid-xu4`

Data Communications

The companion computer is equipped with a WiFi module so that it can receive flight data from the Vicon system (detailed in the following Section 4.3) and communicate with the ground control station. On the other hand, it is interfaced with the Pixhawk using a USB to serial port converter that is connected to the Pixhawk `TELEM2` port.

Data received from the Vicon system (through WiFi) as well as the on-board IMU (via USB-to-UART) are fused before being fed into the control algorithm. Outputs of the controller are movement control commands that will be fed back to the Pixhawk. As illustrated in Fig. 4.6, data transmission is accomplished using the ROS network (refer to Section 4.4). The communication is in the Micro Air Vehicle Link (MAVLink) format[7], which is a communication protocol for UAVs. It is usually used for communication between UAVs and/or GCSs, and between on-board components of the vehicle as well.

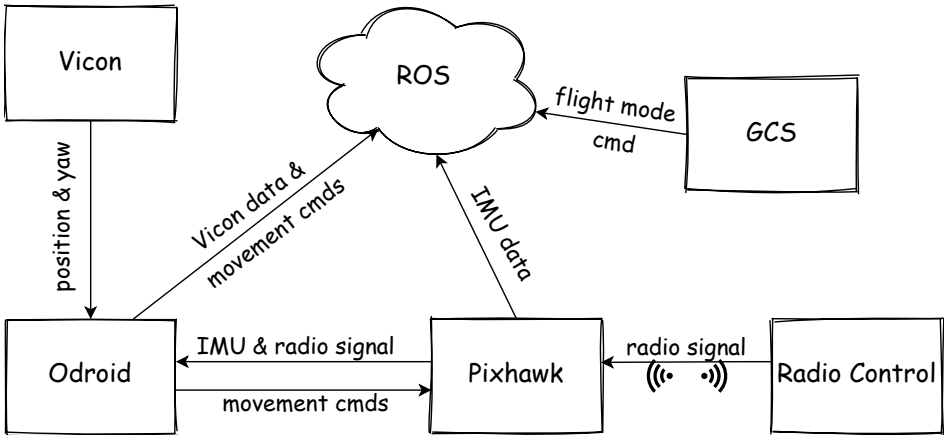


Figure 4.6: Data communication among the subsystems

4.2.4 Servo Dynamics

The hexacopter has been fitted with six ESCs to steer each of the motors on the platform. Actuator signals that are translated from the movement commands will be linearly converted to PWM outputs in the Pixhawk, which are processed by the speed controller and amplified by the MOSFETs in the ESCs, as shown in Fig. 4.7.

[7] `https://mavlink.io/en/`

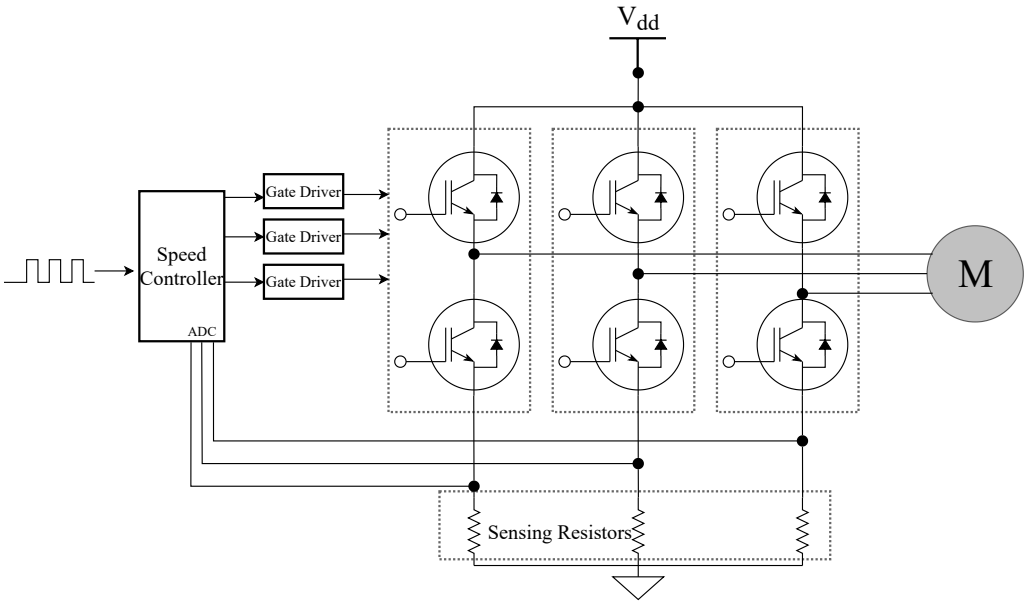


Figure 4.7: Hexacopter PWM-based motor control diagram using ESCs

The PWM-based motor control is in a pull-push structure, which will incur the dead time distortion effect [219] that causes torque ripples and reaction delay. Conventionally the servo dynamics can be simplified as a first order transfer function [91, 220, 52]:

$$G(s) = \frac{1}{\tau s + 1} \tag{4.2}$$

The time constant τ is estimated as 0.1 s in my test with a step response.

4.3 Vicon Motion Capture System

Flight tests are carried out in our indoor flight facility – a Vicon motion capture system[8] shown in Fig. 4.8. The flight testing area is $8.5m \times 5.5m \times 5m$, and is equipped with 21 motion capture cameras. Vicon is on the leading solutions for robot localisation tasks [221]. The system is able to provide accurate real-time positioning with a low-latency data stream, which can be used for vehicle state estimation and conveniently fused into the UAV control algorithm. In addition, the captured data can also serve as ground-truth for controller performance analysis.

[8] `https://www.vicon.com/`

4.3.1 Vicon Tracking

The Vicon system uses optical-passive capture techniques to track object movements, i.e., it is equipped with infrared cameras to track retroreflective markers that are attached to objects to be tracked.

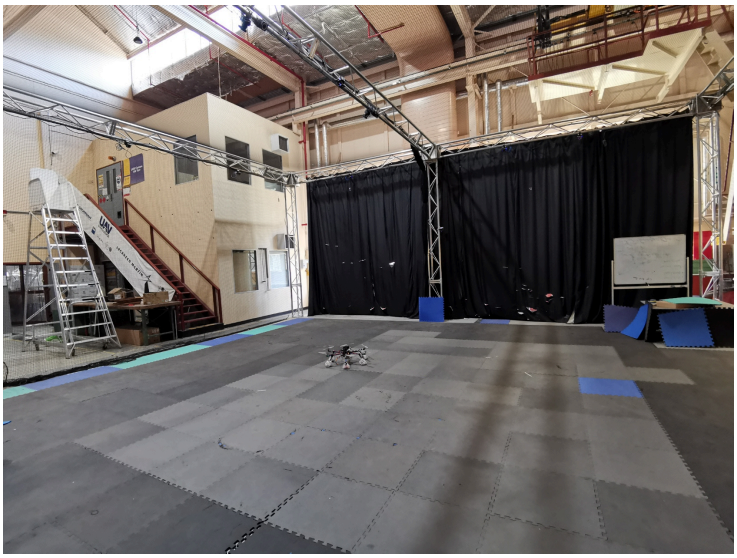


Figure 4.8: Vicon indoor flight facility

The Vicon Tracker application[9] shown in Fig. 4.9 is the host software in which objects are visualised and motion data are accessed. Firstly, the hexacopter being tracked is attached with several markers in an asymmetrical arrangement (Fig. 4.10). Then the object is created in Vicon Tracker to be displayed, with a unique name as a reference provided to the following client application when the state of the object is requested.

4.3.2 Vicon Data Streaming

Finally, with the Vicon DataStream Software Development Kit (SDK)[10], a C++ Vicon DataStream client program is created to collect the position and orientation of the drone and broadcast the data packets through Wifi using the user datagram protocol (UDP).

UDP is a lightweight connectionless protocol in which message-oriented packets are broadcasted to all devices on an IP network. Handshaking communications are not required to establish connections between the devices.

[9] `https://www.vicon.com/software/tracker/`
[10] `https://www.vicon.com/software/datastream-sdk/`

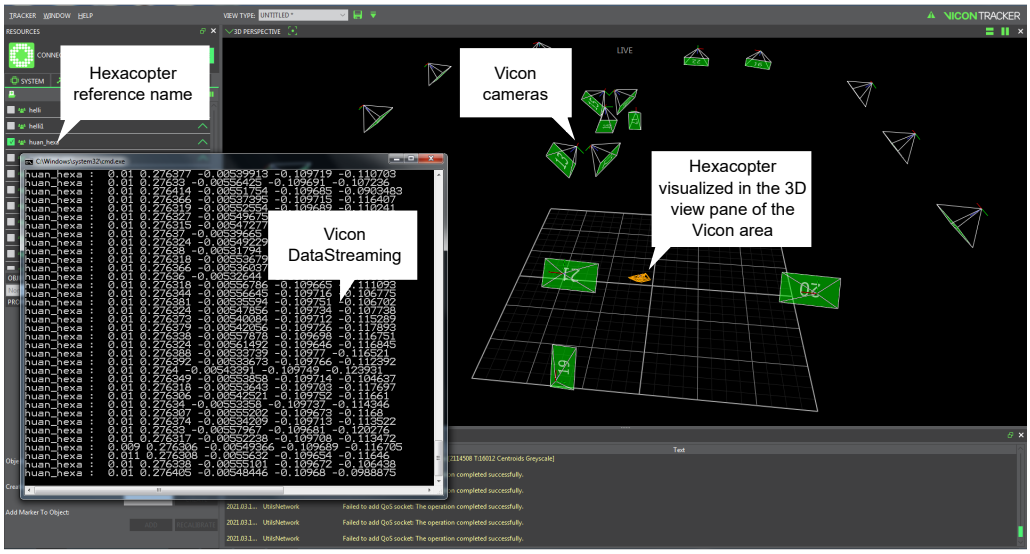


Figure 4.9: Vicon Desktop. Robot states are captured and visualised on the Vicon Tracker application. Flight data are then broadcasted in the C++ Vicon DataStream program.

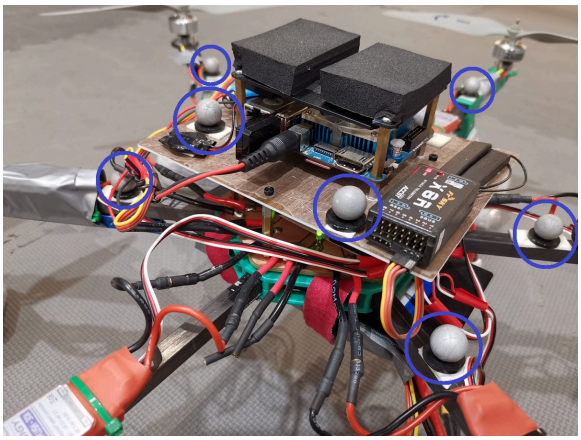


Figure 4.10: Markers are attached to the hexacopter in an asymmetrical arrangement to be tracked by the Vicon Tracker application.

There is no error correction and transmission delay, which is often used in time-sensitive and data streaming applications.

Apart from a 3-byte header offset and a 1-byte error-checking tail, each data packet contains the position, the velocity, the acceleration of the drone in 3 dimensions, as well as the yaw angle and yaw rate. Each data field in the packet occupies 4 bytes. The total size of a packet is 48 bytes. Flight data are sent from the Vicon DataStream program at a rate of 100 Hz.

Positional velocities, accelerations and the yaw rate are estimated from the captured positions and yaw angle. To minimise the estimation error, a moving average filter is applied to each of the data fields to reduce the sensing noise. The averaging window size is 10.

4.4 Robot Operating System

4.4.1 An Overview

Robot Operating System (ROS)[11] is a widely used open source software framework for robotics software development. It is not an operating system (OS) in the traditional sense, but a structured framework above the driver level of the host OS, which is able to provide language-independent and network-transparent communication across a variety of robotic platforms [222].

ROS originally operated on Ubuntu platforms. Currently newer versions are still more or less "experimental" on other operating systems such as Debian and Windows 10 which are supported by the community. ROS is essentially a collection of packages and libraries written in C++, Python and Lisp, each of which provides specialised functionality.

4.4.2 ROS Package Organisation

In ROS, applications are organised in "packages" which may consist of all or parts of the following items:

```
/--
  |-- package.xml
  |-- CMakeLists.txt
  |-- include/
  |-- launch/
  |-- msg/
  |-- scripts/
  |-- src/
```

[11] http://wiki.ros.org/ROS/Introduction

```
|-- srv/
```

Listing 4.1: ROS Package Organization

Specifically, `package.xml` and `CMakeLists.txt` are required files that must be included in a ROS package. The `package.xml` file specifies the package properties as well as the dependencies. The `CMakeLists.txt` file is `cmake`-compliant that describes the package's source files and dependencies as well as instructions in generating targets.

4.4.3 Terminology

ROS is designed in a graph structure based on hardware abstraction. It is a centralised system in which the ROS core service (i.e. `roscore`) must be established at the startup of the system. The `roscore`[12] command will launch a ROS Master, a ROS Parameter Server and a `rosout` logging node. ROS programs must be executed after the `roscore` service is established. The fundamental concepts of ROS implementations are *nodes*, *topics*, *messages* and *services* [222].

Node

ROS processes are represented as nodes in a graph structure, connected by edges called topics. ROS nodes are modules that process a specific operation. Therefore, a ROS package may contain multiple nodes. For instance, the `uav_comm` package described in Section 4.5.3 contains two data processing nodes: one is to process data from Vicon and the other is to process data from the onboard IMU.

Topic

Nodes are able to communicate with each other via topics, whose contents are described as messages. ROS topics are streamed in a unidirectional many-to-many mechanism. Each node will be able to advertise messages to a topic it publishes. On the other hand, a topic may also be subscribed by other nodes and thus create a direct connection from the publishing node to the subscribing node(s).

Message

ROS messages are formatted data that can be either of standard types or user-defined. ROS uses a language-neutral interface definition language (IDL) to

[12] `http://wiki.ros.org/roscore`

describe the fields of messages[222], which are defined in text files with the `.msg` extension under the `msg/` directory (shown in Listing 4.1).

Service

Unlike the ROS topic's "broadcast" communication paradigm, ROS services are intended for request-response interactions. A service is always defined by a pair of message types in `.srv` files under the `srv/` directory: one for the request and the other for the response. One and only one node can advertise a service. A client will then send the request message to the service and await the reply.

Catkin

ROS uses `catkin`[13] to manage the build process, which is a collection of `cmake` macros and Python snippets that provide command line verbs to build the system.

Other tools

Apart from the core infrastructure, the ROS ecosystem has also offered a variety of tools that can be of great convenience during the development, e.g., `Rviz` and `rqt` to visualise and plot data, as well as `roslaunch`[14] to set up ROS parameters and launch multiple ROS nodes in a single file, which is in the XML format with the `.launch` extension under the `launch/` directory.

4.5 UAV Control Using ROS

In this section the ROS UAV control packages will be detailed.

4.5.1 Network Setup

In these experiments, ROS is running across multiple devices (Fig. 4.11). The ROS network uses the SSH protocol (refer to my post[15] for details) to establish connections among these machines. Therefore, we also need to specify the IP address of the ROS master server and the declared local machine, as well as the port to run the master process on:

[13] `http://wiki.ros.org/catkin`
[14] `http://wiki.ros.org/roslaunch`
[15] `https://hnqiu.github.io/2019/04/30/using-ssh.html`

```
export ROS_MASTER_URI=http://odroid:11311
export ROS_IP=odroid
```

In this setup the drone is configured as the master server. The ROS Master which is run by the `roscore` command, is the entry point for naming and registration services. The URI characters should be set the same across all devices.

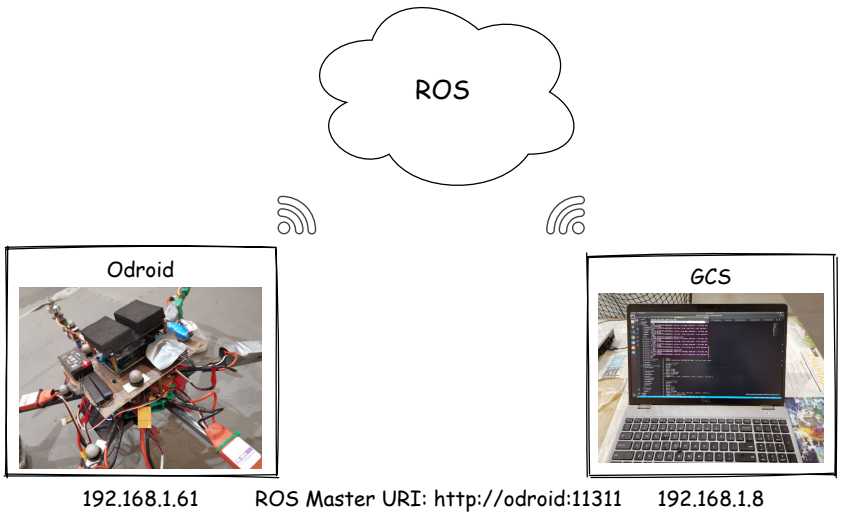


Figure 4.11: ROS communication among multiple machines

4.5.2 Dependencies

The ROS network uses a MAVROS node for data communication among all devices. Therefore, the UAV control packages used in this thesis rely on the following packages:

- `mavlink`[16]
  MAVLink is a communication protocol for UAVs. It is usually used for communication between UAVs and/or GCSs, and between onboard components of the vehicle as well.

- `mav_comm`[17]
  This package manages message and service definitions for MAV communication.

[16] `https://github.com/mavlink/mavlink`
[17] `https://github.com/PX4/mav_comm`

- `mavros`[18]
  MAVROS is the ROS gateway for MAVLink. In this package a `mavros` node will be established, with MAVROS topics for data communication. Some of the necessary topics are:

```
/mavros/actuator_control
/mavros/battery
/mavros/rc/in
/mavros/state
```

Installation instructions are provided in the MAVROS GitHub repository[19].

4.5.3 UAV Control Package Arrangement

UAV control in this thesis is separated into three packages:

- `uav_comm`
  This package is to process sensor data from the Vicon system and the onboard IMU. The package contains two ROS nodes: one subscribes positional and yaw states from Vicon and publishes the processed data to the `"/drone/viconraw"` topic; the other subscribes the angular states of roll and pitch from the IMU and publishes the processed data to the `"/drone/imuraw"` topic.

- `uav_ctrl`
  This package is the main control node where the control algorithms are implemented. It receives the sensor data from both of the aforementioned topics (`"/drone/viconraw"` and `"/drone/imuraw"`). Controllers are implemented in a nested layered paradigm, where the higher-level controllers (velocity & position control) pass their results to the attitude angle controllers; then the angle controllers pass the outputs to the innermost angular rate controllers. The outputs of the rate controllers are movement control commands that will be sent to the Pixhawk by broadcasting the message to the ROS topic `"/mavros/actuator_control"`. The attitude control commands are in the range of [-1.0 1.0], whilist the throttle command is in [0, 1.0]. These movement commands are mapped to actuator channels, from which the PWM outputs for the rotors are subsequently obtained.

- `groundstation`
  This package is run on the user-side GCS. It collects and logs the flight data into text files which can be later visualised using MATLAB.

[18] `https://github.com/mavlink/mavros`
[19] `https://github.com/mavlink/mavros/blob/master/mavros/README.md`

4.5.4 Importing eSpinn

The `uav_ctrl` package uses the aforementioned `eSpinn` library (in Chapter 3) and `boost serialization` as dependencies. Since the `eSpinn` package is also managed by `cmake`, it can be seamlessly integrated into the ROS build system.

- First `eSpinn` is built as a static library

```
# build eSpinn as STATIC lib
add_library(eSpinn STATIC ${SRC_LIST})
target_link_libraries(eSpinn ${Boost_LIBRARIES})
```

- Then the `uav_ctrl` package is built by linking to the `eSpinn` library as well as the `boost serialization` library.

```
# find boost dependencies
find_package(Boost 1.54 REQUIRED COMPONENTS serialization)

...

# include eSpinn & boost dir
include_directories(
  include
  ${eSpinn_INCLUDE_DIR}
  ${Boost_INCLUDE_DIRS}
  ${catkin_INCLUDE_DIRS}
)
# link eSpinn lib dir
link_directories( ${eSpinn_LIB_DIR} )

...

# link executables to the appropriate libraries
target_link_libraries(uav_ctrl eSpinn boost_serialization
   ${catkin_LIBRARIES})
```

4.6 `uav_ctrl` Package

The `uav_ctrl` package is arranged to follow a conventional cascaded control paradigm (as per in Fig. 4.12), where the higher-level controllers (velocity & position control) pass their results (i.e. desired attitudes) to the inner attitude controllers; then the inner controllers pass the outputs (i.e. movement control commands) to a signal mixer that calculates appropriate rotor speed commands. Control signal mixing will be discussed later in Section. 5.4.1.

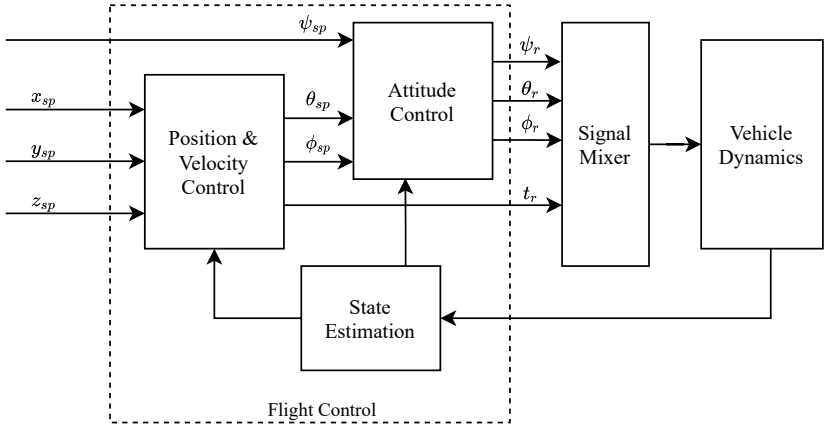


Figure 4.12: A cascaded control architecture adopted in this work. x_{sp}, y_{sp} and z_{sp} denote the position setpoints of the UAV in the Cartesian coordinate, while ϕ_{sp}, θ_{sp} and ψ_{sp} denote the setpoints of roll, pitch and yaw angles respectively. ϕ_r, θ_r, ψ_r and t_r are the corresponding movement control commands.

To make the program structure clear, the `uav_ctrl` package is implemented in an object-oriented programming (OOP) style and takes advantage of the C++ inheritance feature to organise the package structure.

4.6.1 Sensor Data Collection

The base class (named `UavCtrl`) initialises a ROS node handle and establishes the MAVROS connection. It subscribes drone states from the Vicon and the onboard electronics, including the position and orientation of the drone and electronic status (e.g., arming status, battery information).

Next the derived `RadioControl` class subscribes control signals from the radio controller. Switches on the radio controller are used for flight mode configuration, e.g., manual control mode, landing mode, termination due to emergency, etc. Control signals from the movement channels (pilot sticks) are real values. In manual flight mode, these raw data are converted to some specified ranges. The attitude (roll, pitch and yaw) channels are used as attitude setpoints, e.g., 10 degrees in yaw. The throttle channel is normalised to fall into the range of [0 1], which is directly fed to the actuator command topic `"/mavros/actuator_control"`.

4.6.2 Controller Implementation

Flight control is implemented on top of the data collection class (i.e `Radio Control`). `AttitudeControl` takes the attitude setpoints as inputs (either

from the position controllers or from the radio controller directly), and gives out the attitude control commands (ϕ_r, θ_r and ψ_r). Along with the throttle command t_r, they are sent to the signal mixing unit (in the Pixhawk autopilot).

Finally the `PositionControl` class is inherited from the `AttitudeControl` type, where the SNN controller is realised. The `eSpinn` facilities are initialised in this class with the provided APIs. Outputs of x and y control are attitude setpoints which will be passed to the pitch and roll control respectively. Meanwhile height control outputs the throttle command directly.

4.7 Attitude Dynamics

4.7.1 Attitude Control

Attitude control is implemented on top of the `RadioControl` type, which uses proportional-integral-derivative (PID) controllers [42]. In the implementation used in this thesis, if the measured plant output is denoted y and the error between the desired setpoint r and the measured value is denoted $e_y = r - y$, then the PID controller output (u) is the sum of proportional, integral and derivative terms, as formulated in Eq. 4.3.

$$u(t) = K_p e_y(t) + K_i \int e_y(t)dt + K_d \dot{y}(t) \tag{4.3}$$

Unlike the traditional PID control, the D term in this work employs the derivative of the *plant output* instead of the derivative of the *error*. This is to avoid a huge D term output when a sudden setpoint change incurs.

In a similar fashion, use can also be made of proportional (P), proportional+integral (PI) and proportional+derivative (PD) controllers. In these cases, the integral or derivative terms of Eq. 4.3 are omitted as appropriate.

Controller structure

Yaw control is implemented in a parallel PID form (Fig. 4.13), while the roll and pitch control is a cascaded angle-rate controller which is implemented in a P-PID structure (Fig. 4.14). The angle controller is a P-only controller, the output of which is the corresponding rate setpoint. The angular rate controller is at the innermost level with two independent PID controllers for each of the axes, i.e., roll and pitch respectively. Compared with angle control only, in which the P term will have a sudden change when the setpoint is specified, this angle-rate control scheme can achieve more stable flight performance.

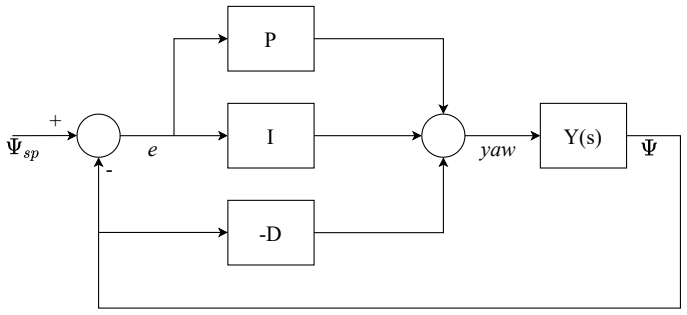


Figure 4.13: Yaw (Ψ) control is implemented in a parallel PID form.

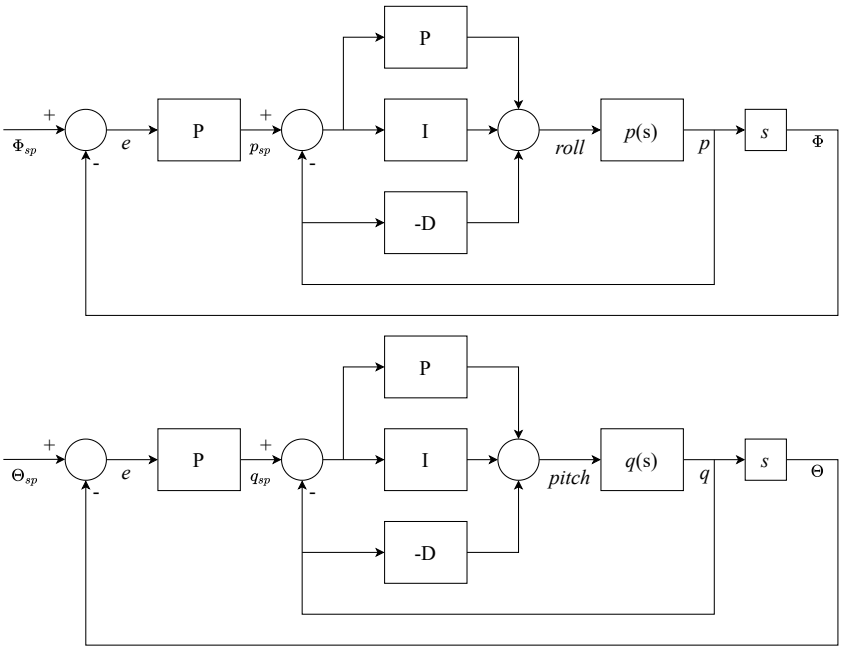


Figure 4.14: Roll (Φ) and pitch (Θ) control is implemented in a P-PID structure. p and q are the angular rates of roll and pitch, respectively.

PID tuning

Tuning of PID controllers is to infer the optimal P, I and D gains in a trial-and-error manner, until the system response is as quick as possible but without oscillations or excessive overshoots.

To tune the attitude control, the drone is set to fly in manual mode. Flight data (including the PID terms) during the course of tests are recorded in a text file and a MATLAB script is created to visualise the responses of the hexacopter in each axis.

I first start with roll control and pitch control, and finally yaw control. Initially the roll and pitch PID parameters use the same values, and fine-tuning for each of the axes is left until the initial tuning is good enough. Empirically the tuning of the controller begins with the P gain, then I gain and D gain if necessary[20]. The tuning steps are as follows:

- First the angular P gain is set as 1. Then the rate P gain is slowly increased until it has a fast response without introducing oscillations. The gain is increased by around 20% per iteration, and decreased by 10% for final tuning.
- Secondly increase the angular P gain until the angular response is fast but without oscillations.
- Thirdly increase the D gain to compensate overshoots. The P gains in the above steps will need to be re-adjusted accordingly.
- Finally the I gain is set to be able to recoup drifts over time. It should not be set too large as it will introduce slow oscillations if so.

A step response is measured to determine the performance of the PID controllers [223]. To test the controller gains, first hover the hexacopter and then rapidly push the pilot stick to one side to provoke a step input to the attitude controller. A well-tuned controller should be able to steer the drone to follow the input without oscillation or overshoot.

4.7.2 Inner loop dynamics

Flight tests have been carried out to gather data to analyse the system dynamics of the inner loops (i.e., roll and pitch). A data logging program is created in the `groundstation` package (described in Section. 4.5.3) to collect

[20] `https://docs.px4.io/master/en/config_mc/pid_tuning_guide_multicopter.html`

the responses of the channels to given attitude setpoints. Data is processed using the MATLAB `System Identification Toolbox`[21].

The relationship of the actual attitude A_a and the setpoint A_{sp} can be identified as a delayed first order transfer function:

$$\frac{A_a(s)}{A_{sp}(s)} = K\frac{e^{-ts}}{Ts+1} \tag{4.4}$$

The roll channel Φ and the pitch channel Θ are given as:

$$\begin{aligned} \frac{\Phi_a(s)}{\Phi_{sp}(s)} &= 0.921\frac{e^{-0.24s}}{0.160s+1} \\ \frac{\Theta_a(s)}{\Theta_{sp}(s)} &= 0.831\frac{e^{-0.24s}}{0.115s+1} \end{aligned} \tag{4.5}$$

4.8 Summary

In this chapter a detailed description is presented on the UAV platform and systems that are used to conduct flight experiments. Control algorithms developed in this thesis can be replicated rapidly on other platforms as the control package runs on the widely-used ROS environment that is supported by a large variety of hardware [224, 225, 226].

[21] `https://au.mathworks.com/products/sysid.html`

Chapter 5

Simulated Control of a Hexacopter Using SNN

This chapter is partly based on the following publication:

H. Qiu, M. Garratt, D. Howard, and S. Anavatti. Evolving spiking neurocontrollers for UAVs. In *2020 IEEE Symposium Series on Computational Intelligence (SSCI)*, Dec 2020, pp. 1928–1935. https://doi.org/10.1109/SSCI47803.2020.9308275

5.1 Introduction

Although learning from scratch approaches purely using physical hardware have been demonstrated [45, 46], extensive hardware-in-the-loop training is required which is time consuming, expensive and can result in equipment damage. Simulation is still an important portion of work in learning-based control development for autonomous robotics, as it is able to provide an observable and reproducible development environment with no risk of damage to hardware. This is especially the case for evolutionary learning, where initial populations are random and the learning process itself is stochastic.

In this chapter the mathematical modelling of a simulated hexacopter UAV is introduced. The hexacopter model incorporates full rigid body dynamics in six degrees of freedom (DoF), as well as nonlinear aerodynamic rotor and fuselage models. Further, a solution is presented to simulate the full control of a hexacopter UAV in six DoF via artificial evolution of spiking neurocontrollers in a modular manner. By decomposing the neurocontroller into modules, it is demonstrated that the development of UAV flight control can be accomplished by an incremental evolutionary approach with MoN-

EAT, a modified version of the NEAT neuroevolution algorithm [140] that provides enhanced performance when evolving small-scale SNNs. The resulting neurocontrollers have simple structures yet are able to provide satisfactory control which has been shown to outperform PIDs.

In the following sections, an overview of the hexacopter model is presented first in Section 5.2. Mathematical modelling of the body dynamics will be formulated in Section 5.3, followed by explanations of signal mixing and servo dynamics in Section 5.4 as well as aerodynamics in Section 5.5. In Section 5.6 the controller development process will be detailed. Results are given in Section 5.7. Discussions and conclusions are provided in Section 5.8 and 5.9.

5.2 System Overview

The system to be controlled is a Simulink hexacopter model derived from the previous work of our group [71], which is constructed from first principles. Such mathematical models have been commonly used in the literature [50, 8, 51, 87, 96]. The Simulink model used here incorporates full 6-DoF rigid body dynamics, as well as nonlinear aerodynamic rotor and fuselage models. Communication lags and servo dynamics are also included, while sensor noise or the ground effect is omitted. The theory used in this modelling can be found in [227, 228, 51].

The hexacopter model used in this chapter is based on a hierarchical architecture. The top-level diagram of the system is given in Fig. 5.1, which is divided into several subsystems. Many aspects of the hexacopter dynamics are modelled with C/C++ S-functions[1], which describe the functionalities of Simulink blocks in C/C++ with MATLAB built-in APIs.

On the left side of Fig. 5.1 are control modules that have been decomposed into 6 DoF, which include the following blocks:

- 'Outer Loop Controller' for translational x and y control,
- 'Attitude Controller' for roll and pitch control,
- 'Yaw Controller,' and
- 'Height Controller'.

Each of the modules takes a reference signal from a random signal generator and outputs a command signal bounded within realistic realms. Details of the controller implementations will be explained in Section 5.6.

[1] `https://au.mathworks.com/help/simulink/s-function-concepts-c.html`

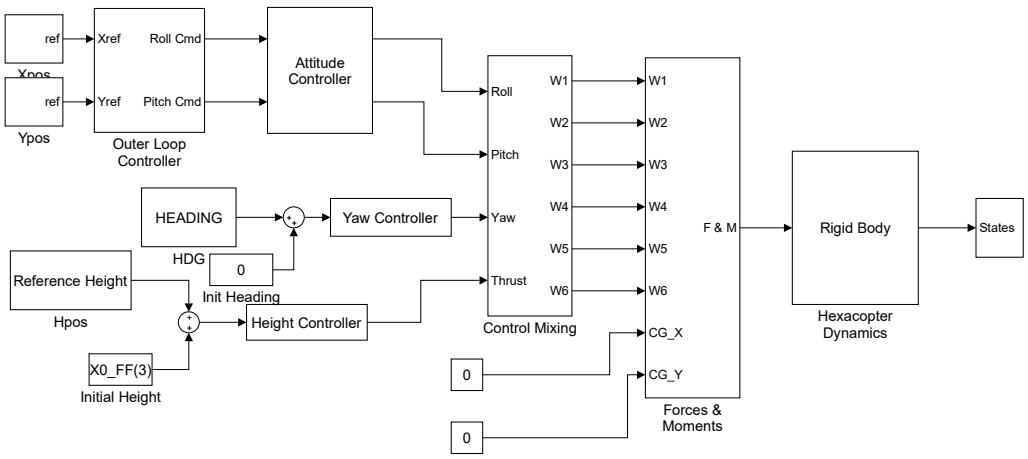


Figure 5.1: Top-level diagram of the hexacopter control model.

The 'Control Mixing' block then combines these controller commands to calculate appropriate rotor speed commands using a linear mixing matrix $\boldsymbol{Q}$ (refer to Eq. (5.18)) according to the position arrangement of rotors. Rotor thrusts and moments of forces are calculated from the rotor speeds based on the relative airflow through the blades [227] in the 'Forces & Moments' block (detailed in Section 5.5). Finally, forces and moments are fed to the 'Hexacopter Dynamics' block to obtain the linear and angular accelerations of the plant based on Newton's second law of motion (detailed in Section 5.3). Local states of the hexacopter will then be converted to the earth-based coordinate system (North-East-Down, described in Section 5.3.1) that can be visualised and archived in the 'States' block.

5.3 Hexacopter Dynamics

5.3.1 Aircraft Conventions

Conventionally in aerospace, the positioning frame of reference is North-East-Down (NED). The x-axis points north horizontally; the y-axis points east horizontally and the z-axis points downwards towards the centre of Earth. This set of coordinate frame is used for aircraft positioning and navigation. Another set of reference frame is fixed on the body of the vehicle that is also necessary for state estimation with regard to the body (e.g. acceleration estimate). As illustrated in Fig. 5.2, the body axes of the hexacopter are fixed to the centre of gravity and rotate with the platform. The right-handed axes systems is oriented such that when the hexacopter is in a level attitude, the

x-axis (longitudinal) is pointing in a forward direction; the z-axis is pointing downwards and the y-axis (latitudinal) is pointing laterally to the right.

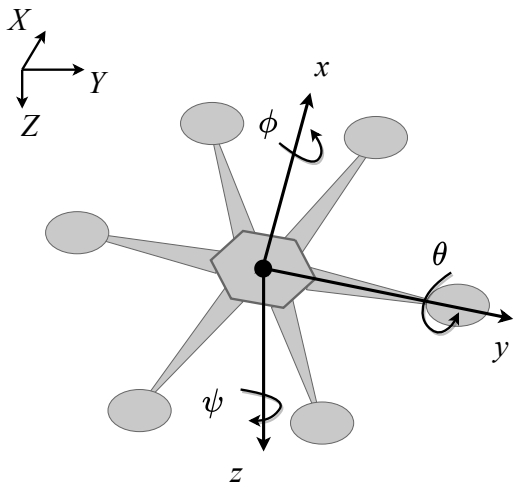


Figure 5.2: Aircraft coordinate system: body-fixed frame (xyz) and earth-based noninertial frame (XYZ).

The velocity along the axes x, y and z are denoted as u, v and w respectively. Rotations of orientation around these axes, known as Euler angles, (i.e. roll, pitch and yaw) are denoted as ϕ, θ and ψ respectively. Rotations are carried out in the order yaw, then pitch, then roll. Meanwhile the rotation rates about the axes x, y and z are given as p, q and r respectively. Orientation of the vehicle with respect to the earth-based frame is parameterised in terms of quaternions instead of Euler angles to avoid singularities when solving trigonometric functions at angles close to $\pm 90°$ (gimbal lock). The quaternion is described as:

$$q = [q_0 \quad q_1 \quad q_2 \quad q_3]^T \tag{5.1}$$

in which the quaternion elements satisfy:

$$q_0^2 + q_1^2 + q_2^2 + q_3^2 = 1 \tag{5.2}$$

The quaternions can be obtained with the following transformation Eq.(5.3)

from Euler angles:

$$\begin{aligned} q_0 &= \cos\frac{\phi}{2}\cos\frac{\theta}{2}\cos\frac{\psi}{2} + \sin\frac{\phi}{2}\sin\frac{\theta}{2}\sin\frac{\psi}{2} \\ q_1 &= \sin\frac{\phi}{2}\cos\frac{\theta}{2}\cos\frac{\psi}{2} - \cos\frac{\phi}{2}\sin\frac{\theta}{2}\sin\frac{\psi}{2} \\ q_2 &= \cos\frac{\phi}{2}\sin\frac{\theta}{2}\cos\frac{\psi}{2} + \sin\frac{\phi}{2}\cos\frac{\theta}{2}\sin\frac{\psi}{2} \\ q_3 &= \cos\frac{\phi}{2}\cos\frac{\theta}{2}\sin\frac{\psi}{2} - \sin\frac{\phi}{2}\sin\frac{\theta}{2}\cos\frac{\psi}{2} \end{aligned} \tag{5.3}$$

The transformation between the body-fixed coordinate and the earth-based coordinate can be described by a rotation matrix $\boldsymbol{B}$, which is given as:

$$\boldsymbol{B} = \begin{bmatrix} q_0^2 + q_1^2 - q_2^2 - q_3^2 & 2(q_1q_2 + q_0q_3) & 2(q_1q_2 + q_0q_3) \\ 2(q_1q_2 - q_0q_3) & q_0^2 + q_1^2 - q_2^2 - q_3^2 & 2(q_2q_3 + q_0q_1) \\ 2(q_1q_3 + q_1q_2) & 2(q_2q_3 - q_0q_1) & q_0^2 - q_1^2 - q_2^2 + q_3^2 \end{bmatrix} \tag{5.4}$$

Note that the rotation matrix is orthogonal, therefore the transformation between the two coordinates would be equally convenient as:

$$\boldsymbol{B}^{-1} = \boldsymbol{B}^T \tag{5.5}$$

5.3.2 Rigid Body Dynamics

Multicopter airframes are generally modelled as a rigid body mass acted upon by the combined effect of gravity and forces generated by the propellers, as well as moments of the forces. The body dynamics are implemented in the 'Hexacopter Dynamics' block in Fig. 5.1. Newton's second law of motion is used to calculate the linear and angular accelerations and hence the state of the drone will be updated.

Linear Motion

Based on Newton's second law, the net force $\boldsymbol{F}$ acting upon the hexacopter is the product of object mass and its linear acceleration vector in the Euler space:

$$\boldsymbol{F} = m\dot{\boldsymbol{V}} \tag{5.6}$$

If $\boldsymbol{V}$ is given in the body-fixed frame, then

$$\boldsymbol{F} = m\dot{\boldsymbol{V}} + \boldsymbol{\omega} \times m\boldsymbol{V} \tag{5.7}$$

where $\boldsymbol{\omega} = [p \quad q \quad r]^T$ is the vector of angular rate.

Let F_x, F_y and F_z be the components of force acting upon the vehicle along the x, y and z axes in the earth-based frame, respectively. The linear motion can be represented as:

$$\begin{bmatrix} F_x \\ F_y \\ F_z \end{bmatrix} = m \begin{bmatrix} \dot{u} \\ \dot{v} \\ \dot{w} \end{bmatrix} + m \begin{bmatrix} p \\ q \\ r \end{bmatrix} \times \begin{bmatrix} u \\ v \\ w \end{bmatrix} \tag{5.8}$$

Angular Motion

The net moment of force $\boldsymbol{M}$ acting upon the body determines the rate of change of the angular momentum $\boldsymbol{L}$:

$$\boldsymbol{M} = \dot{\boldsymbol{L}} \tag{5.9}$$

Similar to Eq. (5.7), if $\boldsymbol{L}$ is given in the body-fixed frame:

$$\boldsymbol{M} = \dot{\boldsymbol{L}} + \boldsymbol{\omega} \times \boldsymbol{L} \tag{5.10}$$

$$\boldsymbol{L} = \boldsymbol{I}\boldsymbol{\omega} \tag{5.11}$$

where $\boldsymbol{I}$ is the inertia matrix. The hexacopter is symmetric with regard to the xz and yz plane. Therefore, the inertia matrix $\boldsymbol{I}$ is diagonal:

$$\boldsymbol{I} = \begin{bmatrix} I_x & 0 & 0 \\ 0 & I_y & 0 \\ 0 & 0 & I_z \end{bmatrix} \tag{5.12}$$

Let L, M, N be the angular momentum about the x, y and z axis respectively in the earth-based frame. The angular motion can be represented as:

$$\begin{bmatrix} L \\ M \\ N \end{bmatrix} = \begin{bmatrix} I_x & 0 & 0 \\ 0 & I_y & 0 \\ 0 & 0 & I_z \end{bmatrix} \begin{bmatrix} \dot{p} \\ \dot{q} \\ \dot{r} \end{bmatrix} + \begin{bmatrix} p \\ q \\ r \end{bmatrix} \times \begin{bmatrix} I_x & 0 & 0 \\ 0 & I_y & 0 \\ 0 & 0 & I_z \end{bmatrix} \begin{bmatrix} p \\ q \\ r \end{bmatrix} \tag{5.13}$$

5.3.3 Summary

From Eq. (5.8) and Eq. (5.13), the status of the UAV can be obtained by solving the following differential equations:

$$\begin{aligned} F_x &= m(\dot{u} + qw - rv) \\ F_y &= m(\dot{v} + ru - pw) \\ F_z &= m(\dot{w} + pv - qu) \\ L &= I_x\dot{p} + qr(I_z - I_y) \\ M &= I_y\dot{q} + rp(I_x - I_z) \\ N &= I_z\dot{r} + pq(I_y - I_x) \end{aligned} \tag{5.14}$$

The orientation of the hexacopter is represented as a quaternion, of which the derivatives is updated with a skew-symmetric matrix following Eq. 5.15.

$$\begin{bmatrix} \dot{q_0} \\ \dot{q_1} \\ \dot{q_2} \\ \dot{q_3} \end{bmatrix} = -\frac{1}{2} \begin{bmatrix} 0 & p & q & r \\ -p & 0 & -r & q \\ -q & r & 0 & -p \\ -r & -q & p & 0 \end{bmatrix} \begin{bmatrix} q_0 \\ q_1 \\ q_2 \\ q_3 \end{bmatrix} \tag{5.15}$$

The rotation matrix $\boldsymbol{B}$ in Eq. (5.4) is applied to convert the local velocities of the UAV (denoted as $[u \quad v \quad w]$) to the earth-based coordinate. The global velocities $[\dot{X}\ \dot{Y}\ \dot{Z}]$ are obtained with the following Eq. (5.16). $[\dot{X}\ \dot{Y}\ \dot{Z}]$ are then integrated to obtain the global position $[X\ Y\ Z]$.

$$\begin{bmatrix} \dot{X} \\ \dot{Y} \\ \dot{Z} \end{bmatrix} = \boldsymbol{B} \begin{bmatrix} u \\ v \\ w \end{bmatrix} \tag{5.16}$$

The aforementioned formulations (i.e. Eq. 5.3, Eq. 5.4, Eq. 5.14, Eq. 5.15 and Eq. 5.16) are all implemented in a C++ S-function, which needs to be built using the `mex`[2] command into an callable object that is wrapped up in Simulink as an S-function block.

To have an intuitive understanding of the attitudes of the drone, the quaternion is subsequently converted into Euler angles, i.e., roll (ϕ), pitch (θ) and yaw (ψ), using the following transformation. Similarly, Eq. 5.17 is implemented in another C++ S-function.

$$\begin{bmatrix} \phi \\ \theta \\ \psi \end{bmatrix} = \begin{bmatrix} atan2(\,2(q_0q_1 + q_2q_3),\ {q_0}^2 - {q_1}^2 - {q_2}^2 + {q_3}^2) \\ asin(2(q_0q_2 - q_3q_1)) \\ atan2(\,2(q_0q_3 + q_1q_2),\ {q_0}^2 + {q_1}^2 - {q_2}^2 - {q_3}^2) \end{bmatrix} \tag{5.17}$$

[2] `https://au.mathworks.com/help/matlab/ref/mex.html`

5.3.4 Model Parameters

A list of the hexacopter inertia properties is provided in Table 5.1.

Table 5.1: Inertia Properties of the Hexacopter Platform

Parameter	Meaning	Value	Unit
m	mass	3.0	kg
I_x	moment of inertia about x-axis	0.04	kgm^2
I_y	moment of inertia about y-axis	0.04	kgm^2
I_z	moment of inertia about z-axis	0.06	kgm^2
g	gravitational acceleration	9.81	ms^{-2}

5.4 Signal Mixing and Servo Dynamics

5.4.1 Control Signal Mixing

In Fig. 5.1, controller commands (i.e. roll (ϕ), pitch (θ) yaw (ψ) and thrust (t)) are combined to calculate rotor speed commands in the 'Control Mixing' block. This is done with a linear mixing matrix $\boldsymbol{Q}$:

$$\boldsymbol{W} = \boldsymbol{Q}\boldsymbol{U} \tag{5.18}$$

where $\boldsymbol{W} = [W_1 \quad W_2 \quad W_3 \quad W_4 \quad W_5 \quad W_6]^T$ is the vector of rotor speed commands and $\boldsymbol{U} = [\phi_r \quad \theta_r \quad \psi_r \quad t_r]^T$ is the vector of control commands. Essentially the mixing matrix $\boldsymbol{Q}$ denotes how much percentage of the control commands should be given to the each of rotors. For example, a thrust command increments the rotor speed command of each rotor equally, whereas a roll to the right command will increase the rotor speed for rotors on the left side of the craft whilst decreasing the rotor speeds on the right. Angular velocities of the rotors $\boldsymbol{\Omega}$ are then converted linearly from the rotor speed commands:

$$\boldsymbol{\Omega} = k_T \boldsymbol{W} \tag{5.19}$$

where k_T is a constant scalar (refer to Table. 5.2).

To make the model more realistic, a communication delay of 0.1 s is introduced to the rotor speed commands when obtaining the rotor thrusts and torques (described in Section 5.5.1).

5.4.2 Servo Dynamics

Meanwhile, the servo dynamic property is modelled as a first order transfer function, with a time decay constant of 0.1 s:

$$G_s(s) = \frac{1}{0.1s + 1} \tag{5.20}$$

This modelling is a simple yet effective representation of the servo dynamics, and is not uncommon in previous research [227].

5.5 Aerodynamics

5.5.1 Rotor Thrust and Torque

Rotor Thrust

The aerodynamics of rotors are described as per the momentum theory [229] and Glauert's induced flow model [230]. Details of the modelling are derived in [227].

In brief, the rotor is modelled as an infinitely thin disc that induces a pressure jump across the plate by the blades. Suppose the hexacopter velocity V_∞ can be decomposed into two components: V_n and V_t that are perpendicular and tangential to the rotor disc respectively. The relationship between the thrust T and induced velocity V_i is given as:

$$T = \frac{\rho a (\Omega R)^2 A_b}{2} \left[\frac{1}{3}\theta_0 (1 + \frac{3}{2}\mu^2) - \frac{1}{2}\lambda' \right] \tag{5.21}$$

where

$$\lambda' = \frac{V_i + V_n}{\Omega R} \quad \text{and} \quad \mu = \frac{V_t}{\Omega R} \tag{5.22}$$

and

$$V_i^2 = \sqrt{(\frac{\hat{V}}{2})^2 + (\frac{T}{2\rho A})^2} - \frac{\hat{V}^2}{2} \tag{5.23}$$

where

$$\hat{V} = \sqrt{V_T^2 + (V_n + V_i)^2} \tag{5.24}$$

To solve Eq. (5.21) and Eq. (5.23) numerically, a binary search algorithm [227] is made to iteratively infer the induced velocity, until the value of V_i can make the difference of the thrust (ΔT) as close to zero as possible.

Rotor Torque

Rotor torques are calculated by dividing the rotor power P by the angular velocity:

$$N = \frac{P}{\Omega} \tag{5.25}$$

where P is composed of the sum of the induced power $P_i = \kappa T V_i$ and the power to overcome the profile drag of the rotor blades P_0 as per equation 5.26. The κ parameter is a correction factor to account for non-uniform inflow and other effects. Profile power is calculated from the rotor blade drag coefficient using blade element theory as per the textbook by Leishman [231].

$$P = P_i + P_0 \tag{5.26}$$

5.5.2 Fuselage Drag

A drag force is applied to the fuselage in a direction opposite to the velocity vector of the aircraft. The magnitude of the drag term in each axis is:

$$\begin{aligned} D_x &= 0.5\rho S_x C_d u^2 \\ D_y &= 0.5\rho S_y C_d v^2 \\ D_z &= 0.5\rho S_z C_d w^2 \end{aligned} \tag{5.27}$$

where C_d is the drag coefficient that is roughly estimated as 1.0 for a flat plate perpendicular to the air flow; ρ is the air density; S_x, S_y and S_z are the equivalent flat plate areas of the fuselage in the respective directions.

5.5.3 Summary

Thrusts and torques of the rotors are summed with the fuselage drag to provide the total forces (F_x, F_y and F_z) and moments (L, M, N) acting on the aircraft, which are then used to update the states of the hexacopter according to Eq. 5.14. Outputs of the 'Forces & Moments' block are:

- yawing torque, obtained by summing up the torque of each rotor.
- rolling and pitching torques, as the sum of products of rotor thrusts and their corresponding torque arms.
- collective thrust, which equals to the sum of thrust of each rotor combined with a drag term introduced on the fuselage caused by aircraft climb/descent, of which the direction is opposite to the vector sum of aircraft velocity.

5.5.4 Aerodynamics Properties

A list of the properties of aerodynamics is provided in Table 5.2.

Table 5.2: Aerodynamics Properties

Parameter	Meaning	Value	Unit
k_T	ratio of rotor command to rotation rate	66.5	rad/s
R	radius of rotor blades	0.164	m
S_x	area of fuselage about x-axis	0.01	m^2
S_y	area of fuselage about y-axis	0.01	m^2
S_z	area of fuselage about z-axis	0.015	m^2
ρ	air density	1.225	kgm^{-3}

5.6 Hexacopter Control

The problem to be resolved in this chapter is the full control of the hexacopter in 6 DoF. This is done by decomposing the flight control into sub-modules and evolving them incrementally.

As shown in Fig. 5.3, yaw control and height control are implemented separately; positional x-axis control is connected to pitch; y-axis control is connected to roll. Each controller takes two known states as inputs: the error between the reference and the actual value in that channel and the current positional/angular velocity. Other than these, the rest are unknown to the controllers. The output of each controller is updated at a rate of 50 Hz (every 0.02 s), which is a command signal bounded within realistic realms. Controller commands are then fed to the signal mixer to obtain the rotor speed commands according to Eq. (5.18).

One of the main issues to be noted in state estimations is that the flight data is usually noisy, due not only to the structural vibration effects on the on-board sensors during flights, but also to the uncertainties of electronic measurements. Therefore, a Gaussian distributed white noise (signal-to-noise ratio is 20 dB) is added to each set of the controller inputs.

A random step signal is implemented in an S-function block to generate a reference for each controller. Both the step values and durations are seeded randomly within specified ranges.

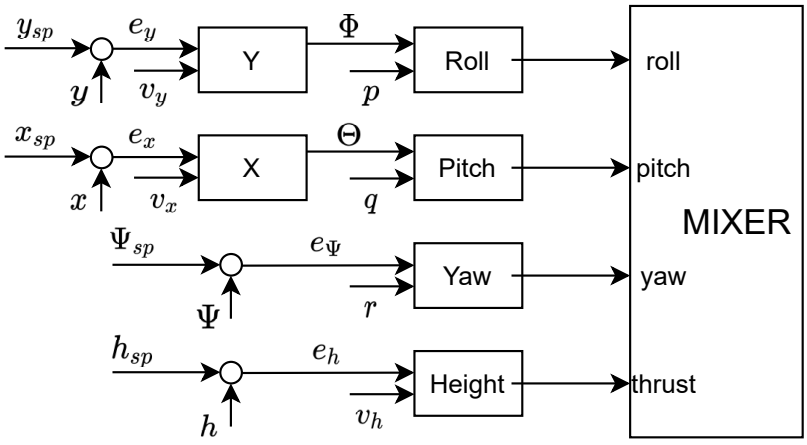


Figure 5.3: Hexacopter control diagram. Each controller takes two known states as inputs: the error between the set point and the actual value (i.e., position errors e_h, e_x, e_y and attitude errors e_Ψ, e_Θ, e_Φ); the current velocity (i.e., positional velocities v_h, v_x, v_y and angular velocities r, q, p).

5.6.1 Controller Development

PID controllers are first implemented on each DoF as a benchmark. Gains of the PIDs are designed to display fast response and minimal steady state output errors in a step response. The next phase is to incrementally develop a neurocontroller for each DoF, by substituting the PIDs with SNNs and evolving them using the MoNEAT algorithm. Such methodology has been proven to work in the literature [48]. The direction of evolution is towards networks that are able to drive the hexacopter to follow a random reference signal with minimal error during the course of flight. Algorithmic flow of the evolution for each DoF is similar:

- Networks are first initialised with minimal structure (2-1-0-1: input-bias-hidden-output), with randomly assigned connection weights. A population of 150 networks are created and categorised into species.

- Afterwards, each network will be evaluated iteratively and assigned a fitness value based on its performance during a flight of 80 s. The fitness calculation used is detailed in Section 5.6.2.

- Finally, these networks will be ranked in descending orders using a fitness sharing scheme [137]. The best parent networks in each species are allowed to reproduce and their offspring will form the next population. Probabilities of the three types of mutations ("AddConnection",

"AddNode" and "AddFullyConnectedNode") are 0.02, 0.01 and 0.01 respectively.

Evolution will terminate either when the population has been stagnant for 12 consecutive generations, or if it has reached a threshold of 50 generations. These numbers are empirically determined to make sure the evolution will most likely plateau in fitness performance without running indefinitely.

5.6.2 Fitness Evaluation

Note the control system to be solved is a Constraint Problem [232], because the states of the UAV must be bounded within some certain range in the real world. However, constraint handling is not straightforward in NEAT – invalid solutions that violate the system's boundary can be generated, even if their parents satisfy these constraints. Therefore, in this experiment the feasibility-first principle [232] is employed to handle the constraints.

The potential solution space is divided into two disjoint regions, the feasible region and the infeasible, by whether the hexacopter is staying in the bounded area during the entire simulation. For infeasible candidates, a penalised fitness function is introduced so that their fitness values are guaranteed to be smaller than those which are feasible.

Fitness of feasible candidates is defined based on the mean normalised absolute error during the simulation:

$$f = 1 - |\bar{e^n}| \tag{5.28}$$

where $|e^n|$ denotes the normalised absolute error between the actual and reference position. The desired solution will have a fitness value close to 1. For infeasible candidates, penalised fitness functions are used so that their fitness values will be smaller than those feasible.

Position & Yaw control

For position control (i.e. x, y and height control) and yaw control, fitness of infeasible solutions is calculated from the time that the hexacopter stays in the bounded area:

$$f = k(t_i/t_t) \tag{5.29}$$

where t_i is the steps that the hexacopter successively stays in the feasible region, and t_t is the total amount of steps the entire simulation lasts. A penalty is applied using a scalar k of 0.2.

Roll & Pitch Control

Solutions are harder to find for the inner loop controllers. During the early stages, none of the networks are feasible as they die out similarly quickly. The number of feasible steps the system lasts using different networks does not differ significantly and thus it is difficult to tell which individuals are the better ones simply from Eq. (5.29) that is used in position control. Therefore, fitness in infeasible roll/pitch control is calculated by adding a penalty p on top of Eq. (5.28):

$$f = 1 - |\bar{e^n}| - p \tag{5.30}$$

$|\bar{e^n}|$ here is the mean normalised absolute error during flight in the feasible region. Penalty is defined based on the distance in time towards simulation completion.

5.7 Results

5.7.1 Evolution in Progress

Flight control is decomposed into 6 modules and evolved incrementally. The entire process can be roughly divided into four steps:

Yaw The first step is evolution of yaw control, during which, feedback controls of the other controller modules are disabled and their outputs are set to zero to eliminate the effects on the training controller. Euler angles are kept in check during the whole simulation. The controller candidate will be categorised as infeasible if any of the drone states goes beyond the specified range.

Roll & Pitch The second step is evolution of roll and pitch control. Likewise, controller outputs of other channels are set as zero in each process. Orientations of the hexacopter are bounded within specified ranges, otherwise the simulation is terminated and the controller candidate is classified as infeasible. Additionally, outer loop controls (x and y) are disconnected from pitch and roll. A random reference generator for desired pitch and roll will instead be connected to the attitude controllers during evolution.

Height Evolution of height control is similar to the development of yaw control.

X and Y Position Finally, the best roll and pitch controllers are activated to train the X- and Y-axis controls. The outer loop controllers are connected back to the inner loops and allowed to evolve.

The whole process has been run ten times in order to conduct a statistical analysis. Table 5.3 shows the averaged fitness values of the evolved spiking controllers in 6 DoF. Two-tailed Mann–Whitney U tests have been conducted to assess the statistical significance between the two approaches. The SNN controllers have better performance compared with the PIDs in all DoF. The differences between the two approaches shown in this thesis are more apparent than those from our earlier publication [214], which has demonstrated that SNNs are more able to cope with system nonlinearities and uncertainties.

Table 5.3: Averaged Fitness in Ten Runs of SNN Controllers vs. PID Controllers

DoF	**Mean Fitness**	
	SNN	PID
Yaw	0.85993	0.83987
Roll	0.88331	0.85875
Pitch	0.86909	0.84957
X	0.90563	0.89912
Y	0.90237	0.88961
Z	0.82525	0.80618

System responses of these channels using the final evolved controllers are given in Fig. 5.4, 5.5 and 5.6. The hexacopter is able to track the reference with smaller overshoots and stable state errors.

5.7.2 Evaluation

Finally, the evolved controllers are all activated and an evaluation signal is used to validate their functionality. Performance of the resulting system is plotted in Fig. 5.7. We can see the subsystems are actually coupled together. When the hexacopter needs to move horizontally, the attitudes (roll or pitch) will alter to tilt the rotor thrust forces and provide horizontal force components to move the vehicle. As a result, vertical lift will decrease and the hexacopter will descend a bit first and recover later with adjusted propeller speed.

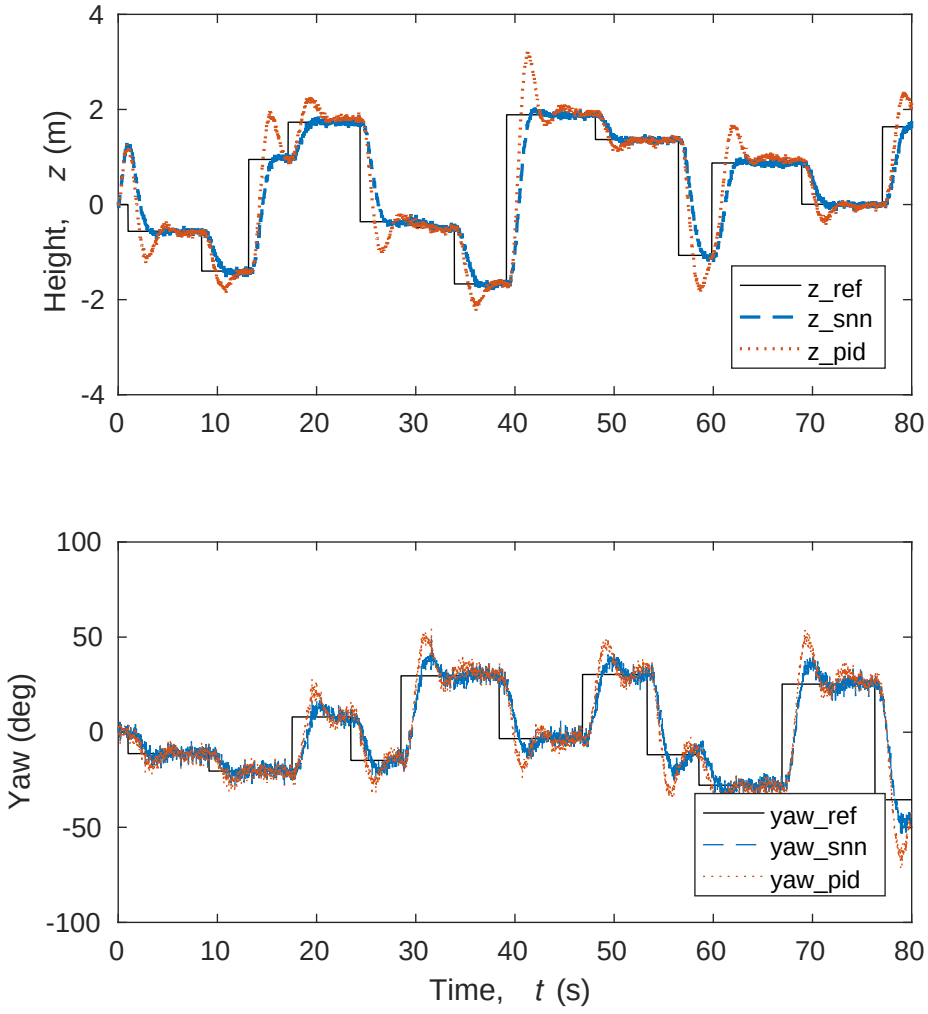


Figure 5.4: Response of the hexacopter to a random height (above) and yaw (below) reference using the evolved controllers. Compared with PIDs, SNNs have shown closer tracking to the reference signal with smaller overshoots.

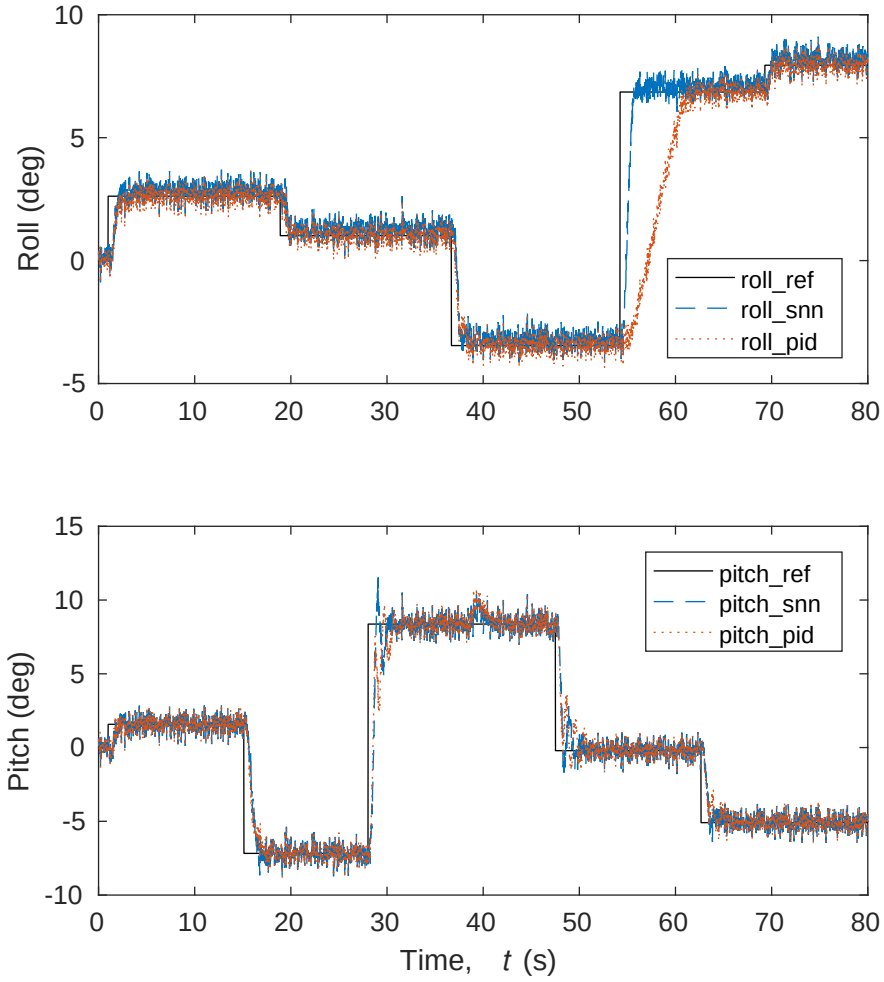


Figure 5.5: Response of the hexacopter to a random roll (above) and pitch (below) reference using the evolved controllers. SNNs are able to follow the reference more closely than the PIDs.

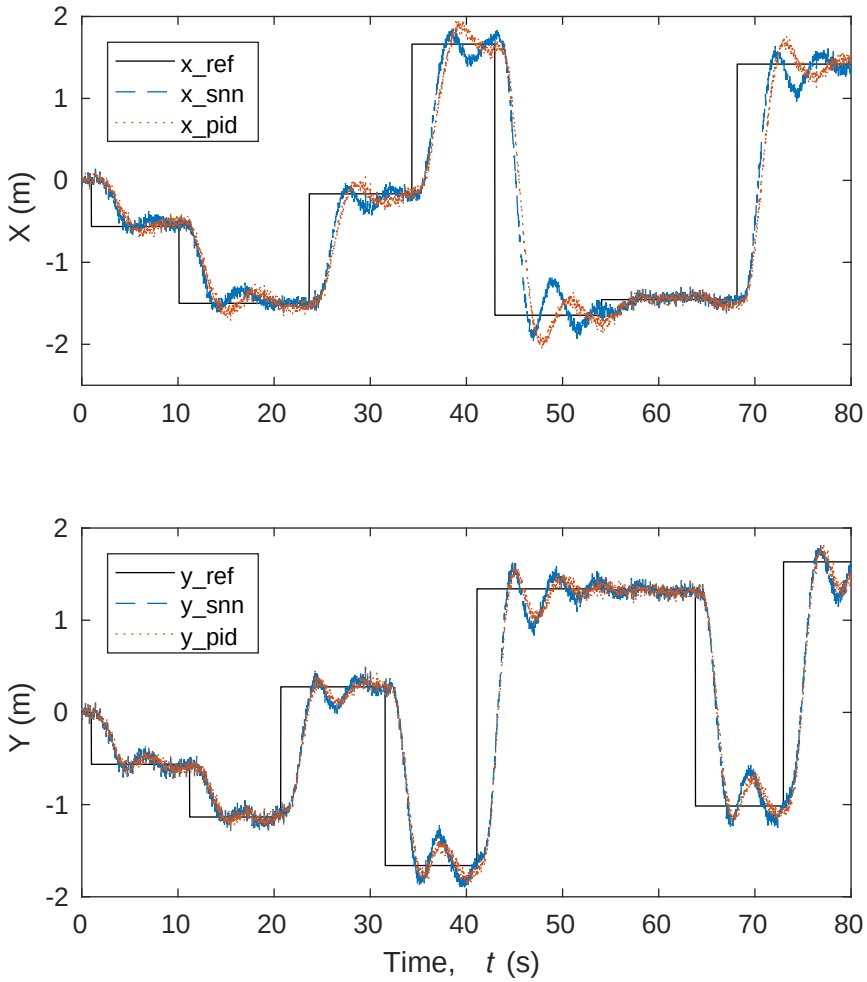


Figure 5.6: Response of the hexacopter to a random X-position (above) and Y-position (below) reference using the evolved controllers. SNN control follows the reference signal more rapidly than the PIDs in both axes. However, their responses are quite similar, as is shown in their fitness values.

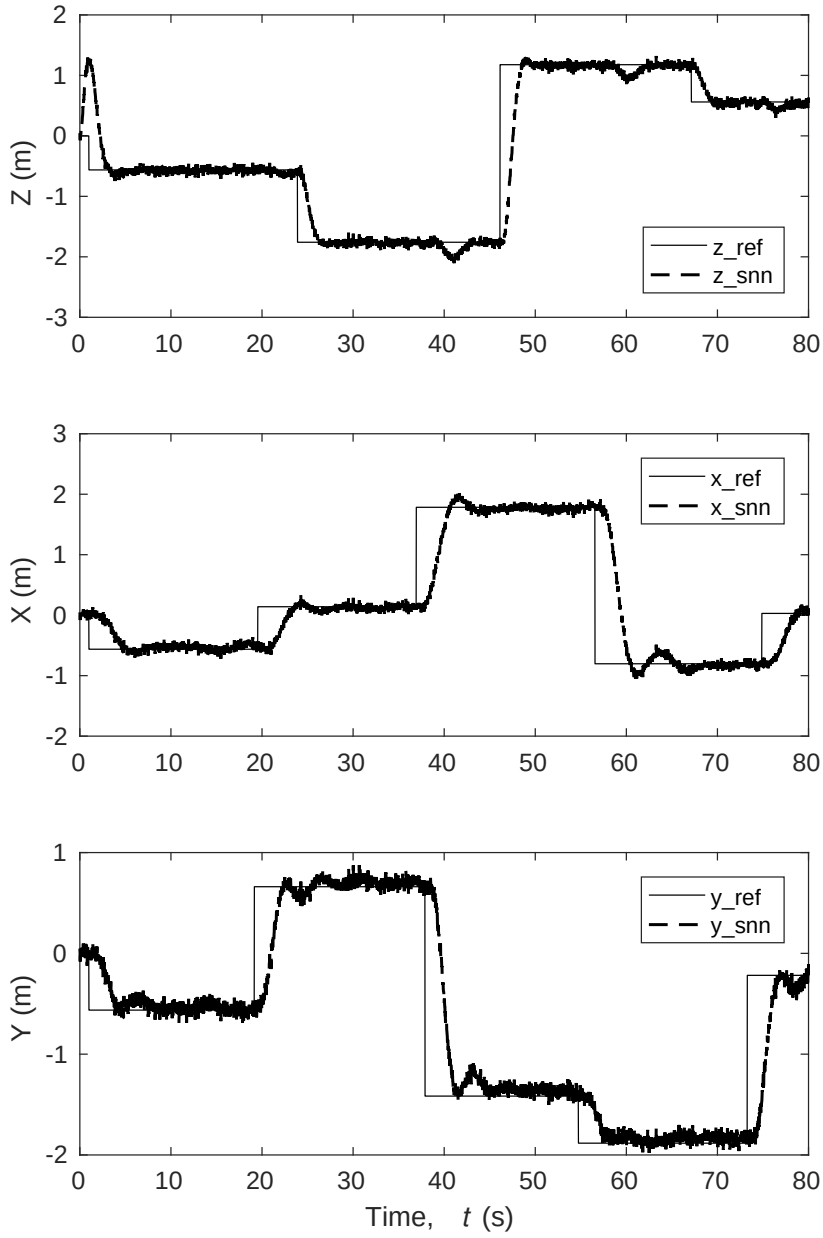


Figure 5.7: Full control of the hexacopter using SNNs

5.8 Discussions

The experiments shown above make a comparison between SNN (non-plastic) and PID control. From Table 5.3 we can observe the attitude (roll, pitch and yaw) and height control have seen clear improvements, while the changes in X and Y control is less significant.

The reason behind this is that although a single PID controller only has three parameters, the cascaded structure in the X and Y axes has introduced extra degrees of flexibility in the system, so that it is still able to achieve fairly good performance. In a real life example, the PX4 Autopilot [73] has utilised a four layer structure (position-velocity-attitude-rate) which has 12 gain parameters in total to secure robust flight performance.

Similarly in SNN control, as the evolution of network topology continues, the degree of variables (connections and neurons) will be able to expand so that they can provide optimised control of the system, as SNNs (and more generally ANNs) are proven universal function approximators [177, 113]. The proposed approach benefits us in that, on one hand, artificial evolution casts an automatic mechanism to make the tuning of network configurations painless, which has also become popular in the gain scheduling of PID control [77]; whilst on the other hand, synaptic plasticity in SNNs empowers the system to be adaptive online, so that the controller developed in simulations will be able to be implemented on physical platforms directly.

5.9 Conclusion

In this chapter the mathematical modelling of a hexacopter UAV is presented. A simulated hexacopter model is constructed in the MATLAB Simulink environment. The primary contribution of this chapter is the successful development of spiking neurocontrol for the hexacopter, which is accomplished by decomposing it into modular networks and evolving them incrementally with the MoNEAT algorithm.

Chapter 6

Heave Control Using Plastic Spiking Neurocontrollers

This chapter is partly based on the following publication:

H. Qiu, M. Garratt, D. Howard, and S. Anavatti. Towards crossing the reality gap with evolved plastic neurocontrollers. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference*, GECCO '20. New York, NY, USA: Association for Computing Machinery, 2020, pp. 130–138. https://doi.org/10.1145/3377930.3389843

6.1 Introduction

A critical issue in evolutionary learning is the transfer of controllers learned in simulation to reality [28]. Previous approaches often require simulation models with a high level of accuracy, otherwise significant errors may arise when the well-designed controller is being deployed onto the targeted platform. This chapter tries to overcome the transfer problem from a different perspective, by designing a spiking neurocontroller which utilises synaptic plasticity to cross the reality gap via online adaptation. With the goal of simulation-to-reality transfer, this chapter proves the concept in a time-efficient manner by transferring from a simpler to a more realistic model, a transfer that encapsulates some issues inherent in crossing the reality gap, i.e. incomplete capture of true flight dynamics and oversimplification of true conditions. Through a set of experiments, it is shown that the evolved plastic spiking controller can maintain its functionality by self-adapting to model changes that take place after evolutionary training, and consequently exhibit better performance than its non-plastic counterpart.

Organisation of the rest of this chapter is as follows. Section 6.2 introduces the reality gap problem, followed by a brief description on the proposed approach by evolving plastic learning rules in spiking neurocontrollers in section 6.3. Section 6.4 presents the plant model to be controlled as well as the system identification process, while Section 6.5 describes the controller development process in detail. Results and analysis are given in Section 6.6. Finally, discussions and conclusions are presented in Section 6.7 and 6.8.

6.2 The Reality Gap Problem

There still exist a large group of aerial robotic studies nowadays relying on simulations as an intermediate step for developing control algorithms [41, 5]. On one hand, a simulation environment is convenient to provide control strategy evaluation and performance analysis. On the other hand, many learning-based approaches spend most of their time conducting large numbers of fitness evaluations. The processing time for these evaluations can be significantly reduced in simulations with no risk of damage to the hardware. This is especially the case for small UAVs, as these platforms are highly dynamic, with strong couplings between different subsystems [5]. Controller design for these agile platforms is naturally difficult, as a poorly-performing controller can lead to catastrophic consequences, e.g., the UAV crashing.

Mathematical models of UAVs in simulations are generally ideal representations of the system dynamics, and thus are not capable of capturing every aspect of the characteristics of the platforms. Such complex nonlinear systems are arduous, and virtually impossible to formulate accurately, because some of the dynamics cannot easily be modelled analytically, e.g., actuator kinematic nonlinearities, servo dynamics, sensing lags, etc [52]. Ignoring these effects can significantly deteriorate the performance of the designed controller when being deployed onto the targeted platform. To address this issue, a common practice is to develop control algorithms based on an identified model that is obtained via a system identification process, in which a data-fitting process is applied to model the exact dynamics from the measured plant's input and output data. Such implementations have been successful amongst previous research [54, 55, 52, 41, 56].

While a lot of works have pursued a perfect model that well characterises UAV platforms, a key issue is that loss of performance is still likely to happen when transferring the well-designed (in simulation) controller onto the real platform that has somewhat different dynamics – a phenomenon known as the *reality gap* [53]. A form of learning in autonomous robotics, therefore, is the ability to maintain the robot's functionality when the controller solution

learned in simulation is transferred to the physical platform, which is often referred to as the ability to *cross the reality gap* [2, 27, 28]. The robot should be robust enough to operate as anticipated even when model variations are encountered.

6.3 Problem Description

In this chapter a novel approach is demonstrated to compensate the gap across different platform representations, which works specifically with a spiking neurocontroller that exhibits online adaptation ability through Hebbian plasticity. An evolutionary learning strategy is proposed for SNNs, which includes artificial evolution of topology and weight configurations as per NEAT, as well as integration of biological plastic learning mechanisms.

Plasticity evolution, as utilised in conventional ANNs [192, 148, 193] and SNNs [194], takes place in the rules that govern synaptic self-organisation instead of in the synapses themselves. The evolved controller is able to exhibit online adaptation due to plasticity, which allows successful transfer from a simple identified model to an accurate model that is built from first principles, indicating that transfer to reality would be similarly successful.

The focus of this chapter is on the development of height control of a hexacopter UAV. The controller takes some known states of the plant model (i.e., error in z-axis between the desired and current position as well as the vertical velocity) and learns to generate a functional action selection policy. The output is a thrust command that will be fed into the plant so that its status can be updated.

The proposed approach to resolve the problem is threefold. First, explicit mathematical modelling of the aircraft is not required. Instead, system identification is carried out to construct a heave model to loosely approximate the dynamics of the hexacopter, during which only basic assumptions are made that the vertical acceleration of the UAV is linearly correlated with the input thrust command and the vertical velocity at a given timestep. In reality, such models are simple to develop and fast to run. Second, neuroevolution takes place as usual to search through the solution space for the construction of functional networks to control the identified heave model. Network topology and initial weight configurations are determined. Finally, the best candidate controller with the highest fitness value is selected for further evolution. Hebbian plasticity is activated and the plastic rule coefficients are optimised by leveraging the power of evolutionary algorithms. The resulting Hebbian plasticity is able to self-regulate connection weights online according to local neural activations when the controller is deployed on the more realistic model.

6.4 Identification of Heave Model

6.4.1 System Modelling

The hexacopter model is derived from first principles and constructed in MATLAB Simulink, which contains 6-DoF rigid body dynamics and non-linear aerodynamics. The detailed mathematical modelling of the hexacopter is elaborated in Chapter 5. The top-level diagram of the system used in this chapter is given in Fig. 6.1.

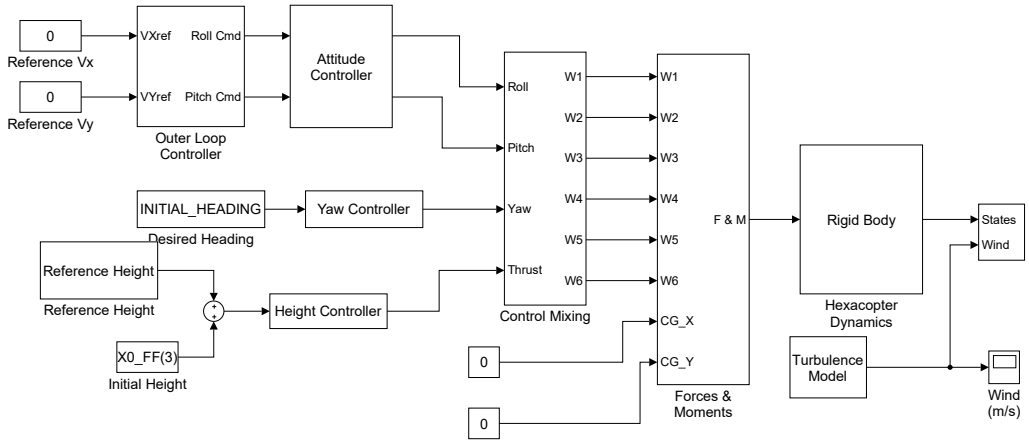


Figure 6.1: Top-level diagram of the hexacopter control model

The 'Control Mixing' block combines controller commands from the 'Attitude Controller,' 'Yaw Controller' and 'Height Controller' to calculate appropriate rotor speed commands using a linear mixing matrix. In the 'Forces & Moments' block rotor speeds are used to calculate the thrust and torque of each rotor based on the relative airflow through the blades. Then the yawing torque will be obtained by simply summing up the torque of each rotor. Rolling and pitching torques can also be calculated by multiplying the thrust of each rotor by its corresponding moment arm. Meanwhile, a drag force is also applied to the fuselage in a direction opposite to the velocity vector of the aircraft. The rotor thrusts and torques are summed with the fuselage drag to provide the total forces (F_x, F_y and F_z) and moments (L, M, N) acting on the aircraft.

Afterwards, the thrust and moments are fed to the 'Hexacopter Dynamics' block. Assuming the UAV is a rigid body, Newton's second law of motion is used to calculate the linear and angular accelerations and hence the state of the drone will be updated. To convert the local velocities of the UAV to the earth-based coordinate a rotation matrix is applied to obtain the global

velocities, which is parameterised in terms of quaternion to avoid singularities when solving trigonometric functions at angles close to $\pm 90°$.

Finally, closed-loop simulations have been tested to validate the functionality of the Simulink model. A random signal generator is implemented to generate the height reference. A PID controller is used in the height control loop that displays fast response and low steady output error as a benchmark.

6.4.2 Identification of Heave

At the first stage a loose approximation is built to resemble the heave dynamics of the hexacopter. Essentially, this is to model the relationship between the vertical velocity v_z, collective thrust T and the vertical acceleration a_z. Fig. 6.2 shows the nonlinear response of vertical acceleration with varying thrust command when the vertical speed is at speeds between -3 m/s to 3 m/s. Note here that the acceleration is actually the net effect of z-axis force acting on the body, which is generated from the rotor thrust and vertical drag caused by rotor downwash and fuselage. The net acceleration a_n would be a_z plus the gravitational acceleration g.

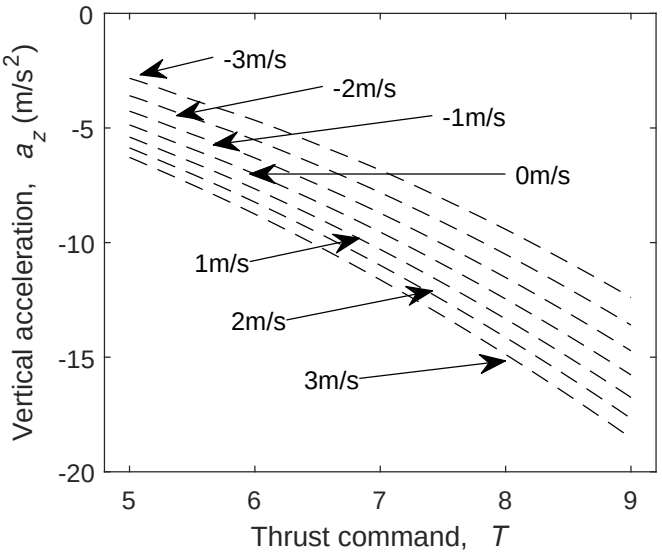


Figure 6.2: Nonlinear relationship between vertical velocity v_z (-3 m/s to 3 m/s), thrust command T and vertical acceleration a_z.

In the identified model, vertical acceleration a_z is approximated as a linear combination of the thrust command T and vertical speed v_z. v_z, on the other hand, is obtained by integrating the net acceleration of z-axis a_n:

$$\begin{aligned} a_z &= k_T T + k_v v_z + b \\ a_n &= a_z + g \\ v_z &= \int a_n \end{aligned} \tag{6.1}$$

where k_T and k_v are configurable coefficients; b is a bias that is also tunable to make sure that the linear function will be expanded at the point where the net acceleration equals zero, i.e., $a_z = -g$.

Let us take two of the acceleration curves from Fig. 6.2 (i.e., for $v_z = 0\,\mathrm{m/s}$ and $v_z = 1\,\mathrm{m/s}$) to model the linear function. The resulting identified linear model is given in Fig. 6.3. k_T is identified as the slope of a_z against T when $v_z = 0$ at the point where $a_n = 0$. k_v is then calculated from the vertical distance between the two nonlinear curves. Finally, b is set to shift the linear curve vertically, so that the identified model will be tangent with the hexacopter curve at the point where $a_n = 0$.

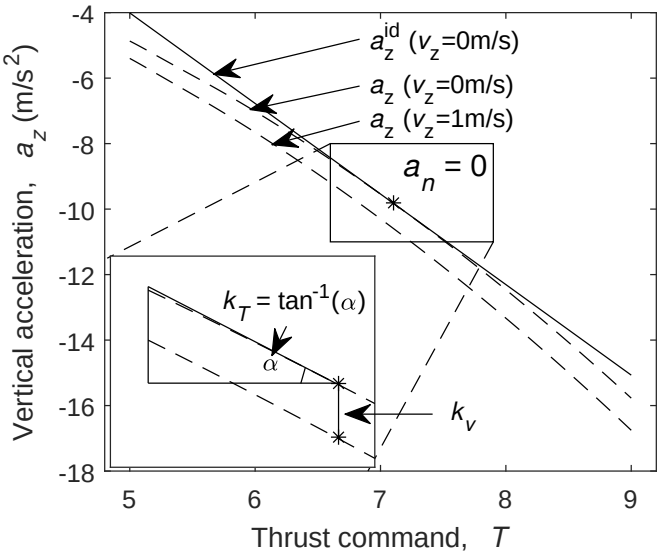


Figure 6.3: Acceleration curves of the identified model (a_z^{id}) and the hexacopter model (a_z) with varying thrust command. The identified curve is tangent with that of the hexacopter model at the point where the net acceleration a_n is 0. α is the slope angle of the identified linear curve, from which k_T is obtained. k_v is calculated from the vertical distance between the two nonlinear curves.

Finally, the same random thrust command is fed to the two different models for validation of functional similarity. System response of the two

models are given in Fig. 6.4. Clearly the response of the identified model differs from the hexacopter model, which is desired, but still the identified model approximates the original system to some extent.

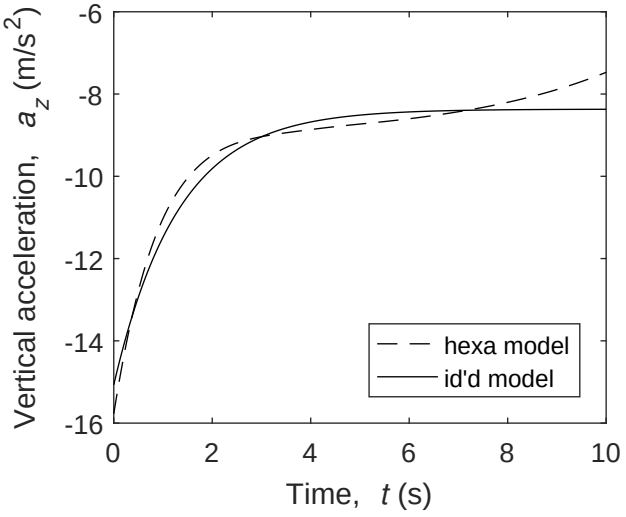


Figure 6.4: Validation of identified heave model. System response of the two models when fed with the same thrust command signal.

6.5 Controller Development and Deployment

The controller is developed offline against the identified model. To speed up the evolution process, the whole simulation during the development is implemented in C++, with the `eSpinn` package described in Chapter 3.

Neuroevolution is first carried out to locate beneficial topology and weight configurations of the non-plastic networks. Plasticity of the champion network is then enabled and allowed to optimise with further evolution. Upon completion of the previous steps, the final network controller is obtained and ready for deployment. To construct the controller in the Simulink hexacopter model, it is implemented as a C++ S-function block. The controller is transferred including the network structure as well as the plasticity learned in simulation.

6.5.1 Evolution of Non-plastic Controllers

Network Configuration

In this experiment the two-dimensional Izhikevich model described in Section 3.3 is employed. The neuron model is formulated with two ordinary differential equations:

$$\begin{aligned} \dot{v} &= 0.04v^2 + 5v + 140 - u + I \\ \dot{u} &= a(bv - u) \end{aligned} \tag{6.2}$$

with after-spike resetting following:

$$\text{if } v \geq v_t, \text{ then } \begin{cases} v = c \\ u = u + d \end{cases} \tag{6.3}$$

Details of the variables are explained in Section 3.3.1.

A three-layer architecture that has hidden-layer recurrent connections is employed, as illustrated in Fig. 6.5. The input layer consists of two neurons which are position error in z-axis e_z and vertical velocity v_z, other than which, the system's dynamics are unknown to the controller. Hidden layer neurons (h) are spiking, whose outputs o_i are based on decoding of firing rates and direct transfer of intermediate membrane potentials. Output of the controller is thrust command that will be fed to the plant model, which is configured as a linear unit to obtain real-value outputs from a weighted sum of activations from hidden-layer neurons. A bias neuron that has a constant output value is able to connect to any neurons in the hidden and output layers. Connection weights are bounded within [-1, 1]. The NEAT topology and weight evolution scheme is used to form and update network connections and consequently to seek functional network compositions.

Encoding of sensing data is done by the encoding neurons in the input layer. Input data are first normalised within the range of [0,1], so that the standardised signal can be linearly converted into a current value (i.e., I in Eq. (6.2)). This so-called "current coding" method is a common practice to provide a notional scale to the input metrics.

Training Process

With the identified model developed according to Eq. 6.1, the first step is to search for optimal network compositions by evolving SNNs using the NEAT algorithm. By 'optimal,' the SNN controller is defined to be able to drive the plant model to follow a reference signal with minimal error in height during the course of flight. Each simulation flight lasts 80 s and is updated every 0.02 s.

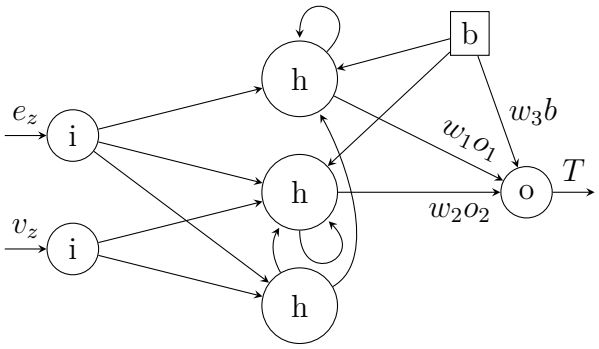


Figure 6.5: Spiking neurocontroller topological expansion using NEAT. Network inputs (i) consist of position error in z-axis e_z and vertical velocity v_z. Hidden layer neurons (h) are spiking, whose outputs o_i involve direct transfer of intermediate membrane potential and decoding of firing rate. A bias neuron (b) is allowed to connect to any neurons in the hidden and output layer. Output thrust command T is calculated based on a weighted sum of incoming neuron activations $\sum w_i o_i$, which will be fed to the hexacopter plant model. Weights w_i are bounded within [-1, 1].

At the beginning, a population of non-plastic networks are initialised and categorised into different species. These networks are feed-forward, fully-connected with random connection weights. The initial topology is 2-4-1 (input-hidden-output layer neurons), with an additional bias neuron that is connected to all hidden and output layer neurons. After initialisation, each network will be iterated one by one to be evaluated against the plant model. A fitness value will be assigned to each of them based on their performance. Afterwards, these networks will be ranked within their species according to their fitness values in descending order. A newer generation will be formed from the best parent networks using NEAT: only the top 20% of parents in each species are allowed to reproduce, after which, the previous generation is discarded and the newly created children will form the next generation. During evolution, hidden layer neurons will increase with a probability of 0.005, connections will be added with a probability of 0.01.

The program terminates when the population's best fitness has been stagnant for 12 generations or if the evolution has reached 50 generations[1]. During the simulation, outputs of the champion will be saved to files for later visualisation. The best fitness will also be saved. Upon completion of simulation, data structure of the whole population will be archived to a text file, which can be retrieved to be constructed in later steps.

[1]empirically determined

Constraint Handling

Similar to Section 5.6.2, fitness evaluation during the evolution is based on a feasibility-first principle [232]. The potential solution space is divided into two disjoint regions, the feasible region and the infeasible, by whether the hexacopter is staying in the bounded area during the entire simulation. For infeasible candidates, a penalised fitness function is introduced so that their fitness values are guaranteed to be smaller than those which are feasible.

The fitness function of feasible solutions is defined based on the mean normalised absolute error during the simulation:

$$f = 1 - |\bar{e^n}| \tag{6.4}$$

where $|e^n|$ denotes the normalised absolute error between actual and reference position. Since the error is normalised, the desired solution will have a fitness value close to 1.

For infeasible solutions, the fitness is defined based on the time that the hexacopter stays in the bounded region:

$$f = k(t_i/t_t) \tag{6.5}$$

where t_i is the steps that the hexacopter successively stays in the bounded region, and t_t is the total amount of steps the entire simulation has. Penalty is applied using a scalar k of 0.2.

6.5.2 Enabling Plasticity

Hebbian Plasticity

Here a rate-based Hebbian model derived from the nearest neighbour STDP implementation [143] is introduced, with two additional evolvable parameters k_m and k_c:

$$\dot{w} = u_i(\frac{A_+}{\tau_+^{-1} + u_i} + \frac{k_m(u_j - u_i + k_c) + A_-}{\tau_-^{-1} + u_i}) \tag{6.6}$$

where k_m is a magnitude term that determines the amplitude of weight changes, and k_c is a correlation term that determines the correlation between pre- and postsynaptic firing activity. These factors are set to be evolvable so that the best values can be autonomously located.

Fig. 6.6 shows the resulting Hebbian learning curve. The connection weight has a stable converging equilibrium at u_θ, which is due to the correlation term k_c. This equilibrium corresponds to a balance of the pre- and postsynaptic firing, with which the network itself is able to regulate the postsynaptic firing corresponding to incoming presynaptic spikes.

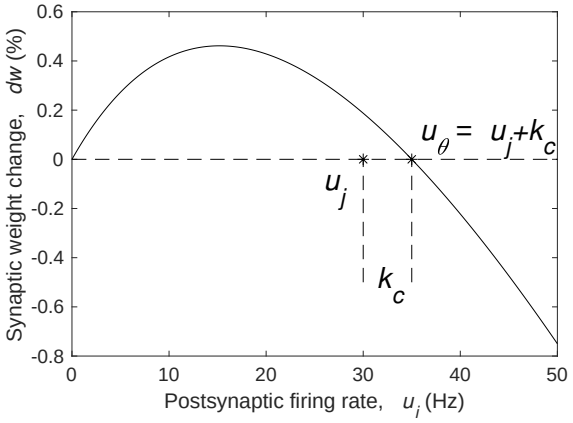


Figure 6.6: Hebbian learning curve with $A_+ = 0.1$, $A_- = -0.1$, $\tau_+ = 0.02\,\mathrm{s}$, $\tau_- = 0.02\,\mathrm{s}$

Evolution of Plasticity

Once the step is done to discover the optimal network topology, the champion network from the previous step is loaded from file, with the Hebbian rule activated. It is spawned into a NEAT population, where each network connection has randomly initialised Hebbian parameters (i.e., k_m and k_c in Eq. 6.6). An EA is used to determine the best plasticity rules by evolving the two parameters.

Networks are evaluated as previously stated. The best parents will be selected to reproduce. During this step, all evolution is disabled except for that of the plasticity rules, e.g. the EA is only used to determine the optimal configuration of the plasticity rules.

Each connection can develop its own plasticity rule. The above-mentioned processes will be offline and only involve the identified model. The dynamics of the accurate model are unknown to the controller. On completion of training, the champion network with the best plasticity rules will be deployed to drive the hexacopter model, which is a more true-to-life representation of the real plant.

6.6 Results and Analysis

Ten runs of the controller development process have been conducted to perform statistical analysis. Data are recorded to files and analysed offline with MATLAB.

6.6.1 Adaptation in Progress

Table 6.1 shows the fitness changes of the best controller during the course[2]. From left to right are non-plastic networks controlling the identified model, plastic networks controlling the identified model and plastic networks controlling the hexacopter model, respectively. The fitness values are averaged among the 10 runs.

Table 6.1: Best Networks' Mean Fitness Values in Progress

	Non-plastic on id'd model	Plastic on id'd model	Plastic on hexa model
Fitness	0.8378	0.8699	0.8595

As stated in 6.5.1, evolution would be terminated if the performance does not improve for 12 consecutive generations before the threshold of 50. For non-plastic controllers, only one of the 10 runs has reached the threshold, and its fitness has only increased by 0.0034 in the last 15 generations. This indicates the evolutionary runs of non-plastic controllers have plateaued and further evolution is unlikely to find better solutions. On the other hand, when plasticity is enabled, an increase in fitness can be clearly observed when controlling the same identified model. The plastic controllers demonstrate better performance even when transferred to control the hexacopter model that has different dynamics.

6.6.2 Plastic vs. Non-plastic

A second comparison is conducted between non-plastic and plastic controllers on the hexacopter model. Results are given in Table 6.2. For 9 out of the 10 runs, we can see a performance improvement when plasticity is enabled. The only one not being better, still has a close fitness value. Statistic difference is assessed using the two-tailed Mann-Whitney U-test between the two sets of data. The U-value is 21, showing the plastic controllers are significantly better than the non-plastics at $p < 0.05$.

Fig. 6.7 shows a typical run using the non-plastic and plastic controller. We can see the plastic control system has a faster response as well as smaller

[2]The figures here are different from our earlier publication [212] because there is a bit difference in the fitness calculation. However, the relative conclusions made among these approaches remain unchanged. Note the values in the following sections are updated as well.

Table 6.2: Fitness of Non-Plastic vs. Plastic Controllers on the Hexacopter Model

Fitness	Non-plastic	Plastic
Run 1	0.8376	0.8701
Run 2	0.8149	0.8543
Run 3	0.8522	0.8793
Run 4	0.8559	0.8930
Run 5	0.8107	0.8324
Run 6	0.8092	0.8332
Run 7	0.8349	0.8676
Run 8	0.8375	0.8512
Run 9	0.8438	0.8732
Run 10	0.8422	0.8413
Mean	0.8389	0.8596

steady state error. It is clear that plasticity is a key component to bridge the gap between the two models.

6.6.3 Validation of Plasticity

To verify the contribution of the proposed Hebbian plasticity, the evolved best plastic rule is extracted and applied to other networks that have sub-optimal performance. With plasticity enabled, a sub-optimal network is selected to repetitively drive the hexacopter model to follow the same reference signal. Fig. 6.8 shows the progress of 4 consecutive runs when a) plasticity is disabled; b-d) plasticity is enabled.

We can see that in Fig. a), there is a considerable steady state system output error. When plasticity is turned on, connection weights begin to adjust themselves gradually. The system follows the reference signal with a decreasing steady state error until around 0.005 m. Meanwhile a fitness increase is witnessed from a) 0.84259, b) 0.85512, c) 0.86457 to d) 0.86784.

The same results can be obtained when the rule is assigned to other near-optimal networks, while for those with poor initial performance, plasticity learns worse patterns. This analysis has justified the proposed evolutionary approach to search for the optimal plastic function, demonstrating that plasticity narrows the reality gap for evolved spiking neurocontrollers.

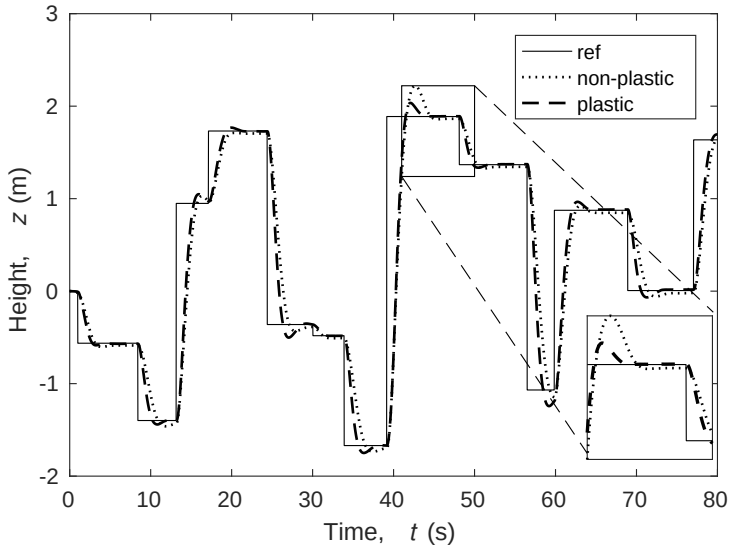


Figure 6.7: Height control using the non-plastic and plastic SNNs

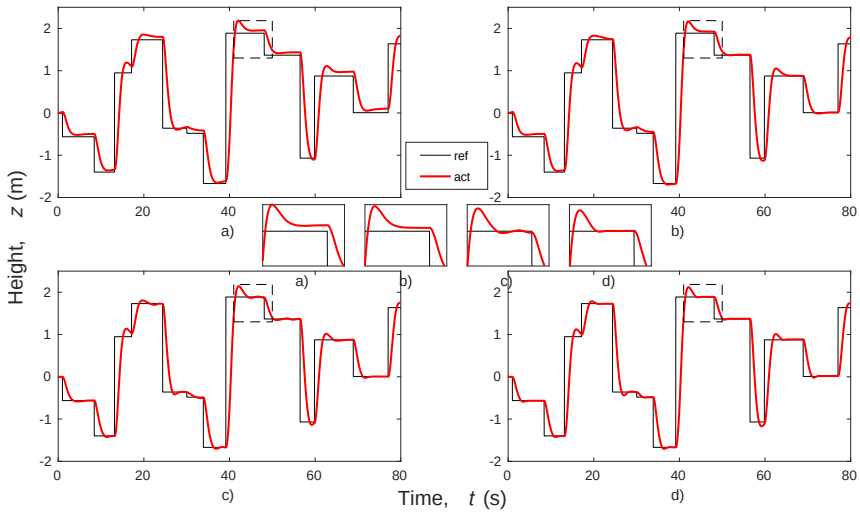


Figure 6.8: Performance improvement during 4 consecutive runs when a) plasticity is disabled; b-d) plasticity is enabled

6.6.4 Comparing with PID control

PID control is a classic linear control algorithm that has been dominant in engineering. The aforementioned PID height controller is taken for comparison. Note here the PID controller is designed directly based on the hexacopter model, whereas the SNN controller only relies on the identified model and utilises Hebbian plasticity to adapt itself to the new plant model. System outputs of the two approaches are given in Fig. 6.9. Evidently the spiking controller has smaller overshoot and steady state output error. The PID controller has a mean absolute error of 0.108 m during the course of flight, while the plastic SNN controller has a value of 0.090 m.

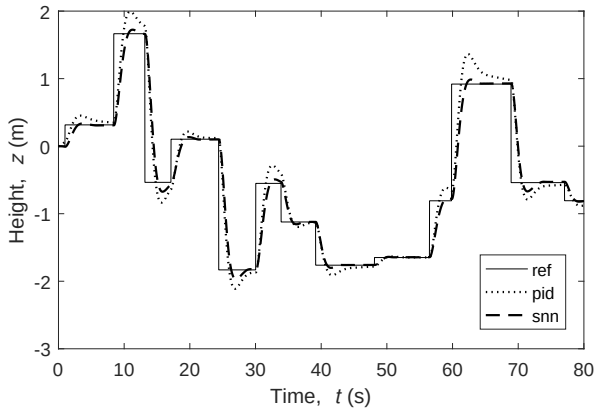


Figure 6.9: Height control using PID and plastic SNNs

6.6.5 Introducing Servo Dynamics and Sensor Noise

Finally, flight tests have been carried out to evaluate the performance of plastic controllers on a more realistic model. A first order servo transfer function with a time constant of 0.1 s and a communication delay of 0.1 s are introduced in the model (refer to Section 5.4). Additionally, a Gaussian distributed white noise is added to the sensor output to simulate actual flight data. The signal-to-noise ratio (SNR) is 20 dB.

An additional reward signal γ is introduced to the Hebbian rules shown in Eq. (6.6):

$$\Delta w = \gamma \Delta w_H \tag{6.7}$$

which is set based on the mean standard error in height.

A sub-optimal controller (trained against the linear approximation) is used to simulate the model to follow a random reference for up to 160 seconds, with

and without the plastic rules. The responses are shown in Fig. 6.10, as well as the mean absolute errors accumulated within a moving window of 5 s. Apparently the non-plastic version has a larger mean error across the course of flight (0.4325 m), while the plastic one has a mean value of 0.3320 m. The system is able to adapt online as the steady state error is decreasing until minimum.

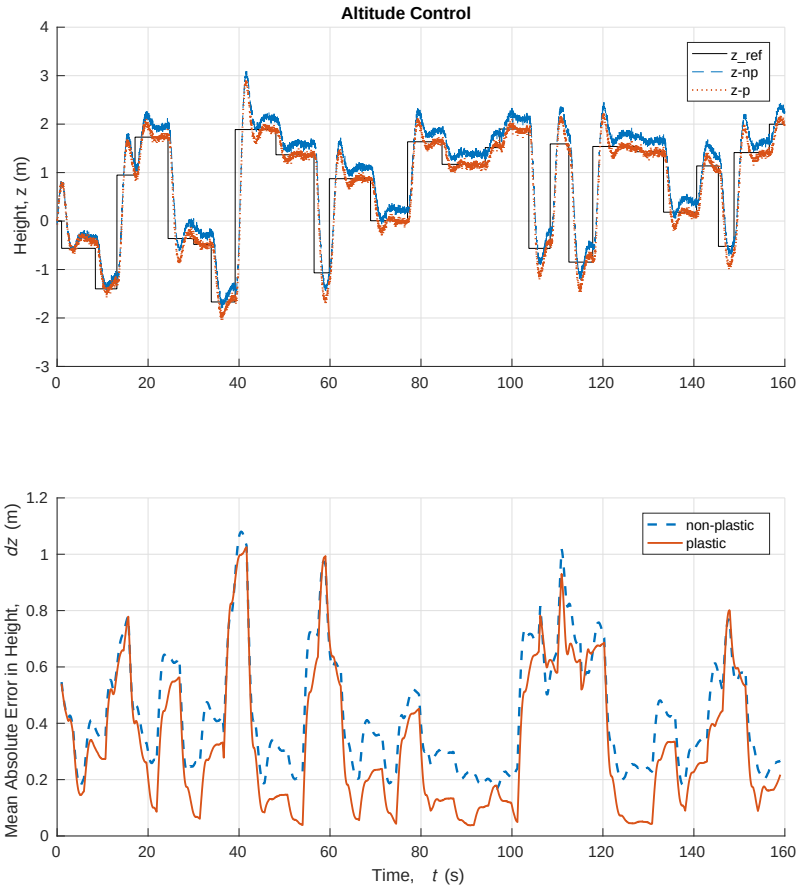


Figure 6.10: Simulation against the model with servo dynamics and sensor noise, where the mean absolute errors are obtained using a moving average method with a window of 5 seconds. The plastic run is able to adapt to the model variance in just a few dozen seconds and consequently has smaller overshoots and steady state errors.

To visualise the adaptation process, a weight watcher is also introduced to monitor connection weight changes during the simulation. The result is

given in Fig. 6.11. We can see weight changes have become stationary in just a few dozen seconds. Only a portion of the connections are modified during the process, while others do not change significantly.

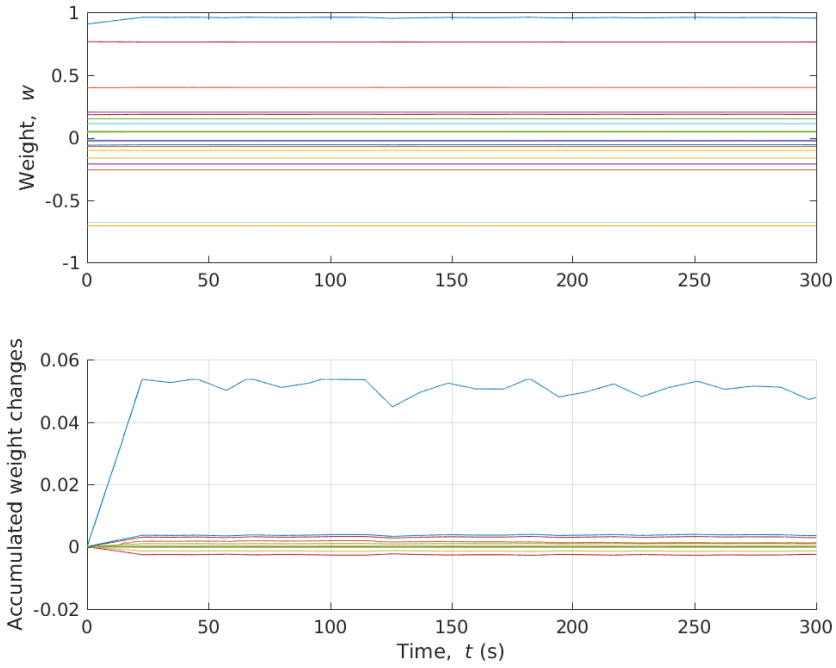


Figure 6.11: Adaptation of connection weights during a longer simulation

6.7 Discussion

When transferring the pseudo-optimal controllers to physical-world applications, one may argue we can still rely on evolution to tweak the connection configurations. However, one main problem is that learning in evolution is delayed because the fitness signal during the process is not immediately available. What is proposed in this chapter is to evolve in advance the adaptive characteristics of the neurocontroller, such that the controller can be self-organising and adaptive to model variations during the entire lifetime in real-time.

Hebbian plasticity is used to learn to form local correlations among the neurons, which controls the direction of neural synchronisation/desynchronisation, e.g. if the presynaptic is firing more frequently, how the postsynaptic neuron should be regulated. With the online reward signal used in Eq. (6.7),

the network system is able to go back and forth to reach a balanced internal state [175].

6.8 Conclusion

This chapter presents a solution to applied evolutionary aerial robotics, where evolution is used not only in network initial construction, but also to formulate plasticity rules which govern synaptic self-modulation for online adaptation based on local neural activities. This work has proposed a plausible strategy to overcome the reality gap problem. It is demonstrated that plasticity can make the controller adaptive to model variations in a way that evolutionary approaches cannot accommodate. With the goal of simulation-to-reality transfer, this controller development strategy will be applied to a real hexacopter platform in the next chapter.

Chapter 7

Experiment: Control of Height

7.1 Introduction

A noted lack of works in existing studies on evolutionary UAV controllers are the transfer of these control techniques onto physical hardware, where two main challenges are: i) traversal of the reality gap, and ii) the redeployment of learned controllers onto embedded platforms. To address these problems, in Chapter 6 an adaptive spiking neurocontroller is presented for the height control of a simulated hexacopter model, which has proven the plausibility of employing synaptic plasticity to cross the reality gap between different model representations. In Chapter 3 the `eSpinn` SNN software package is introduced to facilitate the simulation-to-real-world development of flight controllers, providing seamless reproductions of the learned controllers on embedded hardware, together with their algorithms that allow further optimisation to continue on-board the platform.

In this chapter the proposed methodology is applied to the hexacopter platform described in Chapter 4. Neuroevolution with Hebbian plasticity will be utilised to control the height of the vehicle. Similar to Chapter 6, the process begins from the identification of the heave dynamics, to the offline training of the controllers with regard to the network configurations and Hebbian plastic rules, and finally to the deployment of the controller where online adaptation takes place.

Organisation of the rest of this chapter is as follows. In Section 7.2 a brief recapitulation of the systems and a description of the battery discharge effect is provided. The control strategy is presented in Section 7.3. Each of the aspects are detailed in Section 7.4 and Section 7.5. Flight test results are given in Section 7.6. Finally conclusions are presented in Section 7.7.

7.2 System Overview

7.2.1 Flight Controller Setup

Details of the platform and systems used to conduct actual flight tests are given in Chapter 4. To recap, flight control is separated into two components as shown in Fig. 4.2:

- A Pixhawk Cube: the lower level controller that is responsible for data acquisition from onboard sensors (e.g. IMUs, a battery reader) and a remote radio controller, as well as signal mixing, manipulation and forwarding of control commands to rotors.

- An Odroid XU4 companion computer: the higher level module that runs an Ubuntu system with the ROS environment (refer to Section 4.4). It is responsible for the implementation of attitude control, position control as well as other higher-level motion planning tasks (e.g., trajectory planning, obstacle avoidance). It is also in charge of the communication with the ground control station (GCS) and the Vicon system (refer to Section 4.3) via the MAVROS network.

7.2.2 `eSpinn` in ROS

The robot operating system (ROS) is a popular development environment for robot control applications [224, 151, 233, 225, 234, 226] and is supported by a large variety of hardware. In this work ROS is utilised to facilitate controller implementation with the `eSpinn` library. The control node is realised in the `uav_ctrl` package (refer to Section 4.5.3) where spiking controllers are constructed from a text file in the `boost serialization` format. This text archive contains the saved controller configurations trained during the offline period, which consists of:

- the network topology and parameters of the neuron model and connection weights (types of neurons can be configured independently in the input, hidden and output layers);

- characteristics of the plastic rules that will incur online weight modifications.

7.2.3 Hover Value Estimation with Battery Compensation

As discussed earlier in Section 4.2.2, the Pixhawk autopilot will mix the movement control commands (i.e. roll, pitch, yaw and throttle) to compute the PWM outputs for the rotors. This is done by introducing a linear transformation matrix $\boldsymbol{Q}$ as shown in Eq. (7.1) that is established based on the position of each rotor and the contributions each rotor therefore makes to the total rolling, pitching and yawing moments acting on the UAV.

$$\boldsymbol{W} = \boldsymbol{Q}\boldsymbol{U} \tag{7.1}$$

where $\boldsymbol{W} = [W_1 \quad W_2 \quad W_3 \quad W_4 \quad W_5 \quad W_6]^T$ is the vector of rotor control commands and $\boldsymbol{U} = [\phi_r \quad \theta_r \quad \psi_r \quad t_r]^T$ is the vector of movement control commands. More details can be referred to Section 5.4.1.

In the simulations carried out in previous chapters, rotor speeds Ω are then converted linearly from the motor control commands W, as per Eq. (7.2).

$$\Omega = k_T W \tag{7.2}$$

where k_T is a constant scalar.

In the case of hovering when ϕ_r, θ_r and ψ_r are zero ideally, the PWM duty cycle (η) of the motor control signal is proportional to the throttle command (t_r):

$$\eta \propto t_r \tag{7.3}$$

However in reality, mapping between the PWM control signal and the rotor speed (and consequently the thrust) is not fixed over time, i.e. k_T in Eq. (7.2) is not a constant. From the motor control diagram shown in Fig. 4.7, we can know their relation is also affected by the supply voltage, which in almost every simulation appeared in the literature is not considered as an affecting factor; yet in reality, the battery voltage will decrease gradually during the course of flight due to discharge causing significant changes to k_T.

Therefore, the throttle command should be compensated based on the battery remaining capacity. In this work the hover throttle value h_r is estimated as a linear function of the battery voltage V_b:

$$h_r = k_h V_b + b_h \tag{7.4}$$

where k_h and b_h are constant coefficients. This is only a basic estimation based on the assumption that the battery current stays the same. In my test, the relationship between the hover throttle value and the battery voltage is more like an inverse proportional function. However, as the voltage V_b does not vary too much, it is safe to model it as linear.

7.3 Approach

The strategy used in this chapter is similar to the process proposed in Chapter 6 that is composed of three steps. It begins from identification of the height dynamics from measured data obtained from actual flight runs. The identification process is carried out based on the assumption that the vertical response of the vehicle is instantaneous given the input throttle command.

More thoroughly, the hover throttle value h_r is first estimated as a linear function of the battery voltage. As the voltage will drop due to load increase and there is also sensing noise that incurs measurement corruption, the captured battery voltage is filtered using a moving average algorithm. The hover throttle value is then modelled with the following equation:

$$h_r = k_h \bar{V}_b + b_h \tag{7.5}$$

where $\bar{V}_b$ is the averaged battery voltage over a moving time window.

Then the throttle command captured during the flight t_r is trimmed by the hover value. Together with the vertical speed v_z it is used to approximate the vertical acceleration a_z in the identified model. The resulting model is given as:

$$\begin{aligned} a_z &= k_t(t_r - h_r) + k_v v_z + b \\ v_z &= \int a_z \end{aligned} \tag{7.6}$$

k_t, k_v and b in Eq. (7.6), as well as k_h and b_h in Eq. (7.5) are all constant coefficients needed to be identified. This identified model does not incorporate servo dynamics as modelled in Section 4.2.4, nor communication lags with sensors and actuators. Further, it is only a basic approximation of the aerodynamics of rotors and flight dynamics (i.e. aerodynamic force drag acting upon the fuselage).

Secondly, to construct a feasible controller, the development process is conducted against the identified model, with the support of `eSpinn`. At the first stage, MoNEAT is used to determine beneficial network configurations. The champion network is then spawned into a population with randomly initialised plastic rules, which are determined by further evolution. The final compositions of the controller is then saved as a text file in the `boost serialization` format.

Finally, the text archive is directly duplicated on Odroid, from which a spiking neurocontroller can be constructed and allowed to optimise on-board the platform.

7.4 Identification of Heave Dynamics

7.4.1 Flight Data Collection

Training data for the identified model is collected from manual control flights. Two sets of flights are carried out to gather the data:

- The first scenario is to hover the drone at a setpoint for a couple of minutes, during which minimal variations of input commands are required. The objective of this flight is to take the input throttle command as the hover value and a linear mapping is approximated between it and the battery voltage. k_h and b_h in Eq. (7.5) are then determined.

- In the second scenario, the drone is manoeuvred up and down continuously with varying input throttle commands, whilst the attitude commands (roll, pitch and yaw) are kept minimal (as close to zero as possible), yet is still enough to hold the vehicle vertical. The objective of this flight is to model a mapping between the vertical acceleration and the combination of throttle command and vertical velocity, so that k_t, k_v and b in Eq. (7.6) can be identified.

A ROS package is created on the GCS for data logging with a sample rate of 10 ms. Flight data are gathered via the MAVROS network, then saved as a text file that can be read later in MATLAB to carry out the identification process. Messages from the following ROS topics are recorded for identification:

```
/mavros/battery
/mavros/actuator_control
/drone/viconraw
```

Battery Voltage

The power source of the Pixhawk Cube is provided by an adaptor that converts the battery voltage to its operating power supply (5 V). Additionally, the power converter comes with a battery monitor that is able to measure the battery voltage and current with an on-chip analog-to-digital converter (ADC) (available on the ARM-based STM32 processors). Battery states are then published to the ROS topic `/mavros/battery`. In the data logging instantiation on the GCS, voltage values are subscribed from this topic.

Throttle Value

Throttle control commands are subscribed from the ROS topic `/mavros/actuator_control`, in which control signals are arranged in the following order:

0. roll (-1..1)
1. pitch (-1..1)
2. yaw (-1..1)
3. throttle (0..1)

4–7. reserved

The ranges of these values are indicated inside the parentheses.

Height

Height of the hexacopter is subscribed from the `/drone/viconraw` topic, in which the message is a user-defined data structure that includes:

```
Header header
float32[3] pos
float32[3] vel
float32[3] rates
float32[3] angles
float32[3] acc
```

For better visualisation, height values are transformed positive above ground.

7.4.2 Process of Identification

Hover vs. Battery Voltage

The identification process is carried out in MATLAB using a linear regression model[1]. The resulting estimation of hover throttle is given in Fig. 7.1. The approximated hover throttle value is then plotted with the actual captured data against time, which is shown in Fig. 7.2.

Vertical Acceleration vs. Throttle and Velocity

Another flight is used to model the vertical acceleration. Take-off and landing of this flight is cut off to eliminate the ground effect. The `fitlm` regression function is used to model the relationship between the acceleration a_z and the combination of trimmed throttle command ($t_r - \tilde{h_r}$, where $\tilde{h_r}$ is the estimated

[1] `https://au.mathworks.com/help/stats/fitlm.html`

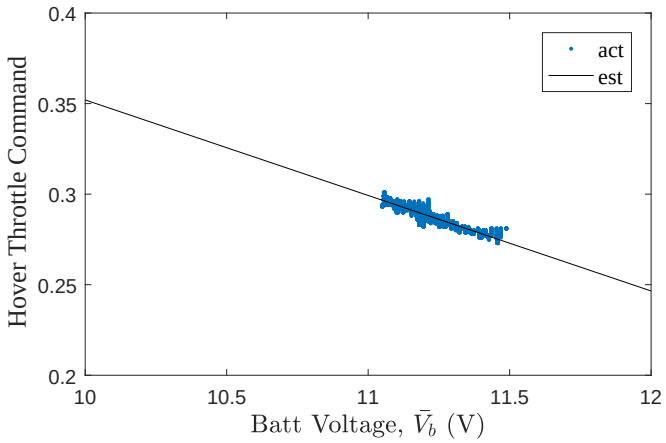


Figure 7.1: Estimation of hover throttle against battery voltage. Hover throttle value h_r is estimated as a linear function of remaining battery voltage $\bar{V}_b$.

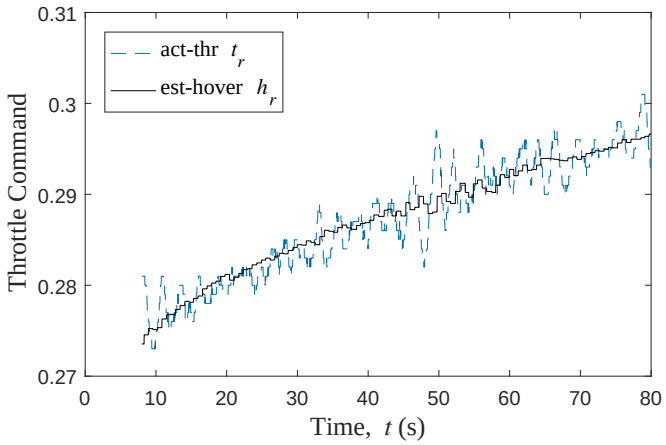


Figure 7.2: Approximated hover throttle value against time.

hover value inferred from the battery voltage) and velocity v_z. The resulting fitting curve of acceleration is given in Fig. 7.3.

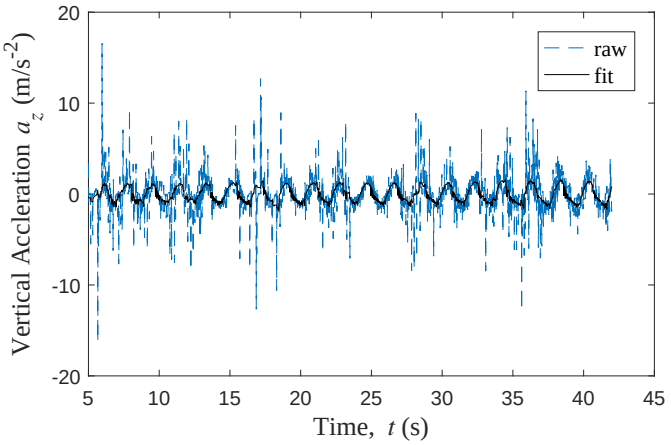


Figure 7.3: Approximation of vertical acceleration a_z.

7.5 Controller Development

7.5.1 Evolution of Non-plastic Controllers

The training process as well as the identified model is implemented in C++ to increase the training speed. The controller takes two hexacopter states as inputs: the position error in the z-axis e_z and the vertical speed v_z. A uniform distribution of white noise is added to the inputs to simulate sensing noise. Hidden neurons are configured as Izhikevich models. The output of the controller is a throttle command that will be added a hover value before being fed to the plant model.

The development process is similar to that in Section 5.6.1. A population of 150 networks are initialised with minimal structure, i.e., two input nodes, one bias, one output and zero hidden nodes. Each network is evaluated iteratively for a flight of 50 s at a rate of 10 ms. Fitness for feasible solutions is defined based on the mean normalised absolute error $|\bar{e_z^n}|$ during the flight. Infeasible simulations will be terminated when any of the drone states is out of boundary. Fitness is then defined based on the time that the drone stays within the bounded area.

MoNEAT is first carried out to evolve the network topology and connection weights. Probabilities of the three types of mutations, i.e. "AddConnection", "AddNode" and "AddFullyConnectedNode" as in Fig. 3.7, are 0.02,

0.01 and 0.005 respectively. Evolution will terminate either when the population has been stagnant for 12 consecutive generations, or if it has reached a threshold of 50 generations.

7.5.2 Hover Throttle Estimation

The final throttle command is obtained with Eq. (7.7), as the sum of the network output and the hover throttle estimate, which is calculated from the measured battery voltage:

$$\begin{aligned} t_r &= t_{snn} + \tilde{h_r} \\ \tilde{h_r} &= k_h \tilde{V_b} + b_h \end{aligned} \tag{7.7}$$

where t_{snn} denotes the output from the SNN controller; $\tilde{h_r}$ denotes the hover throttle estimate; and $\tilde{V_b}$ denotes the measured battery voltage.

Note that Eq. (7.5) marks the mapping between the hover value and the battery voltage when the drone *is* hovering. However in reality, the actual voltage ($\tilde{V_b}$ in Eq. (7.7)) will vary based on current loads. For instance, an increase in rotor speeds that happens when the throttle command becomes larger will lead to a battery voltage drop due to internal resistance. Therefore, the measured voltage $\tilde{V_b}$ at a timestep is very likely to differ from the value it should be when the drone is hovering. This coupling effect is modelled as well in the identified model during the controller training process, so that the SNN controller will be evolved to provide extra redundancy.

7.5.3 Evolution of Plastic Rules

Similar to the step in Section 6.5.2, artificial evolution is used to determine the Hebbian rule parameters, while topological and weight evolution is disabled. Weight changes during the simulation are induced by Hebbian plasticity only. Beneficial plastic rules are selected to reproduce. Each network is able to develop its own plastic rule. Finally the controller information is serialized and saved into a text file.

7.5.4 Controller Deployment

The final controller solution is directly copied to the onboard Odroid computer. An additional reward signal γ is introduced to the Hebbian rules shown in Eq. (6.6):

$$\Delta w = \gamma \Delta w_H \tag{7.8}$$

in which γ is set based on the mean standard error in height over a moving time window of 0.1 s:

$$\gamma \propto |\bar{e}| \tag{7.9}$$

Here $|\bar{e}|$ denotes the current mean absolute error between the actual and reference height. The reward signal is used interchangeably in all connections and is updated every 1 s.

7.6 Flight Tests

Flight tests are conducted using PID control, non-plastic and plastic SNN control. The mission is to maintain the height of the drone at a setpoint after take-off. The vehicle is armed on the ground at initialisation, then a take-off signal is sent from the radio controller so that height control is taken over by the SNN or PID. Meanwhile, orientations are controlled manually using the radio controller.

Heave responses of these runs are plotted below (Fig. 7.4, Fig. 7.5 and Fig. 7.6). The PID has an overall mean absolute error of 1.723 cm when stabilised (after 20 s), with a standard deviation of 2.180 cm. An apparent overshoot can be observed at take-off. The non-plastic SNN run has a mean absolute error of 3.321 cm due to the steady state error. However its deviation (0.776 cm) is much smaller than the value of the PID run. Finally, the plastic run has the best performance overall. It has a mean error of 0.919 cm, with a deviation of 0.867 cm. A weight watcher is instantiated to monitor the changes of connection weights during the course of flight. The weights are able to converge in a few dozen seconds as shown in Fig. 7.7.

7.7 Conclusion

A complete pathway to the design of height control is presented in this chapter, beginning from identification of the height dynamics from measured data obtained from actual flight runs, to controller development against the identified model by integrating Hebbian plasticity with artificial evolution, and finally to controller deployment together with the algorithm that allows further optimisation. Flight tests conducted on a hexacopter UAV have shown that the proposed plastic spiking controller is able to accommodate model variations when transferred from a simplified simulated model, and consequently exhibit better performance than its non-plastic counterpart.

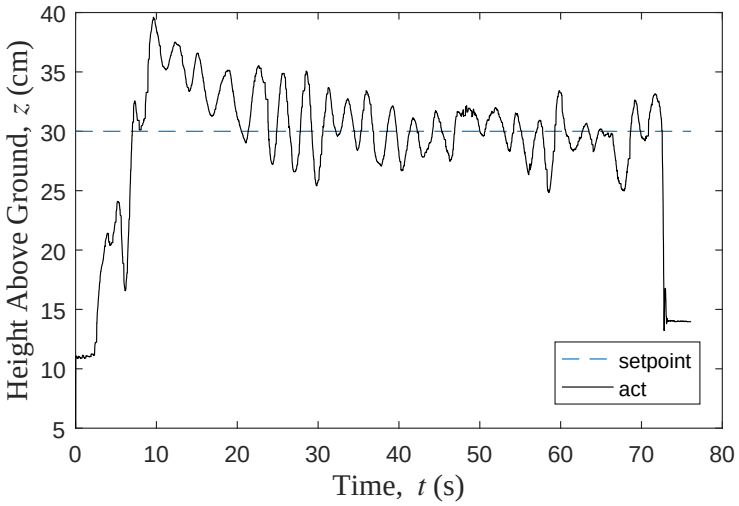


Figure 7.4: Heave Response using PID control.

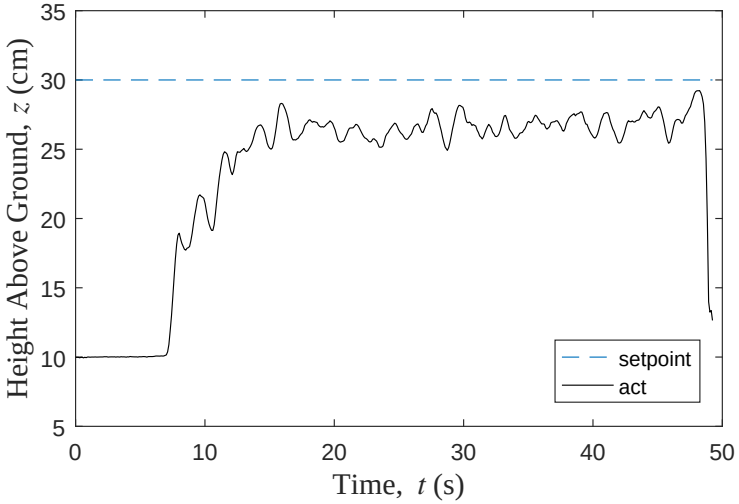


Figure 7.5: Heave Response using non-plastic SNN control.

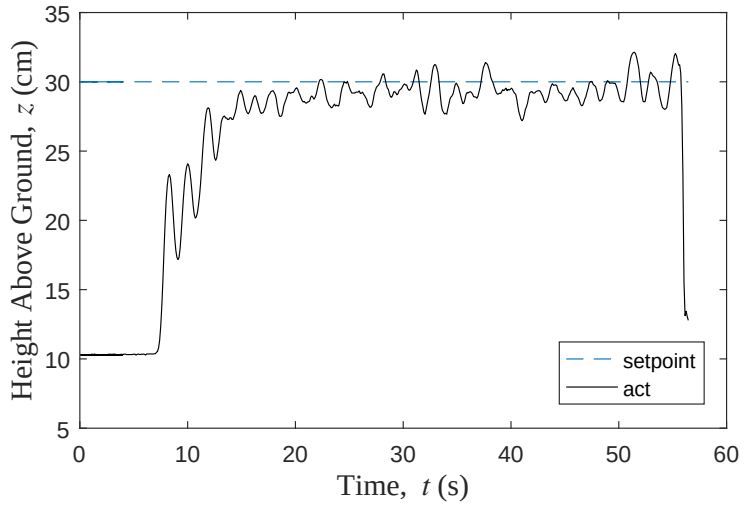


Figure 7.6: Heave Response using plastic SNN control.

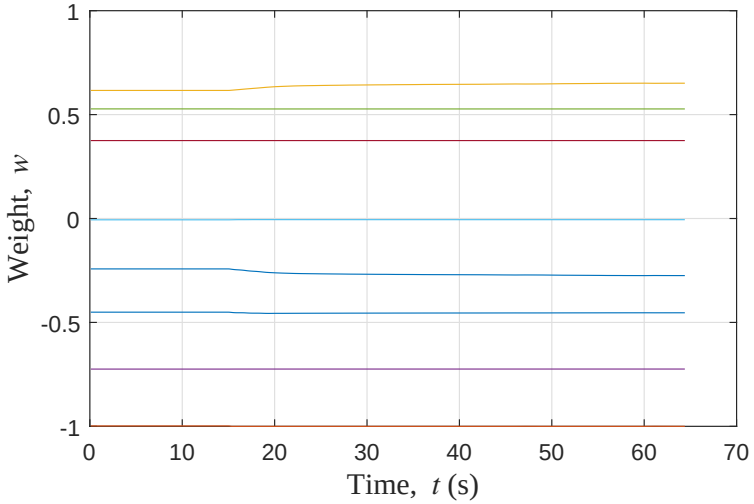


Figure 7.7: Weight adaptation of the plastic run.

Chapter 8

Conclusions & Discussions

8.1 Summary of Achievements

The primary contribution of this work is the implementation of a procedural methodology towards the autonomous control of a hexacopter UAV, by learning spiking neural networks with Hebbian synaptic plasticity via artificial evolution. A complete pathway is created for the development of flight control for physical UAV platforms, by beginning from identification of the system dynamics, to decomposing the controller into modules and evolving them incrementally, and finally to integrating Hebbian plasticity for online adaptation to model variations from simulation to on-board the platform.

8.1.1 Evolution of Network Structure and Plasticity

The `eSpinn` SNN library package is presented in the thesis with a focus on offline-online hybrid training for robotic applications. Simulations in the MATLAB Simulink and Python environments are supported by MATLAB S-functions and `pybind11` respectively, allowing rapid prototyping and validation of control strategies in simulation. It also incorporates seamless reproductions of models on embedded platforms, with the dependency of `Boost Serialization`. In Chapter 3 a Flappy Bird game was demonstrated as a simple example in the Python environment. Neuroevolution was used to learn a functional SNN that is able to self-play the game. In future work, it would also be worth interfacing `eSpinn` to the OpenAI Gym simulation [235], which includes a variety of benchmark problems that have attracted growing interests in the reinforcement learning world. These games and control problems can be employed to investigate the performance of SNNs.

A varying-topological neuroevolution algorithm based on NEAT [140] is utilised for SNN learning in this work (i.e. MoNEAT), which involves the

usual search for connection weights as well as determination of network topology. MoNEAT employs an incremental topological growth mechanism to discover the (near) minimal effective network structure. The basis of MoNEAT (and NEAT) is the trace of historical markings, which are essentially gene IDs. They are used as a measurement of the genetic similarity of network topology, with which network diversities are preserved via speciation. MoNEAT complements the original NEAT with an additional mutation (i.e. 'AddFullyConnectedNode' as in Fig. 3.7), so that the population can escape from less favourable spaces more quickly. A comparison between MoNEAT and NEAT is provided in Chapter 3 on the pole-balancing problem, which is a widely recognised benchmark. It is demonstrated that MoNEAT are significantly faster and more effective in solving the stated problem.

In this work, synaptic plasticity is in the form of a rate-based Hebbian model of the nearest neighbour STDP implementation [143]. The correlation between the pre- and postsynaptic firing is determined by artificial evolution. Such methods are not not uncommon in the literature. For instance, the Hebbian ABCD model described in Eq. (2.11) is similarly evolved to optimise [152, 153, 154]. In particular, HyperNEAT is used in [153], which is an indirect encoding method derived from NEAT.

8.1.2 Flight Control

A mathematical hexacopter model was constructed in MATLAB Simulink that incorporates rigid body dynamics and nonlinear aerodynamics. Simulated development of spiking controllers for all degrees of freedom is presented in Chapter 5, which is carried out in an incremental manner from the inner loop attitude control to the outer loops. Control of each DoF is developed independently. Fitness evaluations of the controller candidates are categorised into feasible and infeasible regions. Statistic analyses showed that the evolved SNN controllers have better performance than PID control, in which the roll and pitch control are most improved. This indicates SNNs are more able to provide optimised control in the presence of system nonlinearities.

Plastic height control is conducted on an actual hexacopter UAV (Chapter 7). The controller is able to adapt to the plant in a few dozen seconds and maintain the height of the drone at a setpoint after take-off. During flight tests some oscillations in heave response were observed as shown in Fig. 7.5 and Fig. 7.6. These oscillations may well have been caused by delays in the control system. In practice, sensing data is first filtered using a moving average algorithm, which incurs a delay that affects the performance of the designed controller. Servo delay is another significant factor. The servo dynamics in the hexacopter platform can be modelled as a first order transfer function

(as in Eq. (4.2)), with a time constant of 0.1 s that is roughly estimated from a step response. Communication lags in wireless data transmission also contribute to reduced controller performance and potential instability. Time delay due to these effects can be as much as 0.36 s as observed in a commercial Parrot AR.Drone [236] and a MikroKopter running ROS [237].

8.2 Areas for Future Study

This section addresses the limitation of this thesis, and discusses some future work that will advance the study beyond the stated contributions outlined in Chapter 1. The following topics provided for future research are in the field of autonomous robotics, bioinspired and brain-like computing.

8.2.1 Control of Flight

Although full control of the hexacopter in six DoF is accomplished in simulation, flight tests have only been conducted with regard to the height control on the real platform. Future work on the control of other DoF can be carried out using a similar methodology. Existing research on SNNs has primarily focused on mission-level objectives [199, 174, 184] such as waypoint following, target approaching and obstacle avoidance. It would be interesting to investigate the performance of spiking controllers in the lower-level domain, i.e. attitude control [49].

In addition, learning and adaptation should be studied in online learning scenarios where training data streamed online can be changing over time [238]. Investigations should be carried out in terms of how SNNs learn throughout the course of the flight. The adaptation ability should also be studied under uncertain conditions such as wind disturbance. Further, as system delays in sensors and actuators as well as in communications can degenerate the flight performance, modelling and analysis of such effects should be considered in the design of robust control. Lastly, the battery discharge effect may be more effectively handled by incorporating the measured battery voltage and current as extra inputs to the SNN controller.

8.2.2 Exploiting Spike Timing

Chapter 3 has discussed information coding in spikes. Although rate coding has been common in robotic applications, existing studies suggest that spike-based coding can provide richer contents and faster signal processing

(hence less reaction time) than rate-based approaches [126, 239], as computation with firing rates has ignored the timing of spikes and must be acquired over a longer period of spiking activities. For high-bandwidth control applications such as that required for accurate UAV control, reducing delays in computation should result in improved performance. In addition, learning with spike timing can be carried out via spike-based synaptic plasticity (e.g. STDP) [159], where reward modulation is found to be associated with the TD algorithm and neural activities could be explained in a reinforcement learning paradigm [25].

8.2.3 Low-cost Hardware Implementation

Although specialised neuromorphic chips have brought forth massive brain-like computing [116, 240, 120], such solutions are too expensive and in many cases not available to the majority. A low-cost alternative is to instantiate networks on programmable hardware, e.g. field programmable gate arrays (FPGA) [241, 165, 242, 243]. FPGAs are reconfigurable hardware and capable of massive parallel data processing which would not be possible on general purpose system-on-chips (SoCs), such as the ARM-based chips on the Odroid and Pixhawk autopilot used in this work. Such systems can be integrated with vision-based navigation applications.

8.3 Concluding Remarks

Bioinspired computational models and learning mechanisms hold a great deal of promise for providing autonomous systems like UAVs with adaptable, dynamic, and highly-tuned closed loop performance. Artificial evolution of the innate properties of a spiking neurocontroller with synaptic plasticity can be used to empower a robotic system to adapt in-mission to variations in the platform.

In nature, evolution is the path that cultivates intelligence across millions of years. However, intelligence is not directly imposed by evolution itself, but is acquired from the learning activities during the occurrence of neuromodulation based on synaptic plasticity. Evolution and learning shape intelligence together, from a generational and an individual scale respectively. Artificial evolution and bioinspired computation articulated in machines offer us a fascinating insight into the learning and generalisation ability of human brains, in return advancing AI technologies in a way that we have not previously witnessed.

Bibliography

[1] P. R. Chandler and M. Pachter. Research issues in autonomous control of tactical UAVs. In *Proceedings of the 1998 American Control Conference. ACC (IEEE Cat. No.98CH36207)*, volume 1, pages 394–398 vol.1, 1998.

[2] Stefano Nolfi. *Behavioral and Cognitive Robotics: An Adaptive Perspective*. Institute of Cognitive Sciences and Technologies, National Research Council, Roma, Italy, 2021.

[3] Agoston Restas. Drone applications for supporting disaster management. *World Journal of Engineering and Technology*, Vol.03No.03, 2015.

[4] Hamid Menouar, Ismail Guvenc, Kemal Akkaya, A. Selcuk Uluagac, Abdullah Kadri, and Adem Tuncer. UAV-enabled intelligent transportation systems for the smart city: Applications and challenges. *IEEE Communications Magazine*, 55(3):22–28, March 2017.

[5] F. Santoso, M. A. Garratt, and S. G. Anavatti. State-of-the-art intelligent flight control systems in unmanned aerial vehicles. *IEEE Transactions on Automation Science and Engineering*, 15(2):613–627, April 2018.

[6] Hongwei Mo and Ghulam Farid. Nonlinear and adaptive intelligent control techniques for quadrotor UAV – a survey. *Asian Journal of Control*, 21(2):989–1008, 2019.

[7] Hyunchul Shim. *Hierarchical flight control system synthesis for rotorcraft -based unmanned aerial vehicles*. PhD thesis, Mechanical Engineering, University of California, Berkeley, 2000. Copyright - Database copyright ProQuest LLC; ProQuest does not claim copyright in the individual underlying works; Last updated - 2016-05-24.

[8] R. Mahony, V. Kumar, and P. Corke. Multirotor aerial vehicles: Modeling, estimation, and control of quadrotor. *IEEE Robotics Automation Magazine*, 19(3):20–32, 2012.

[9] Wulfram Gerstner and Werner Kistler. *Spiking Neuron Models: An Introduction*. Cambridge University Press, New York, NY, USA, 2002.

[10] Abigail Morrison, Markus Diesmann, and Wulfram Gerstner. Phenomenological models of synaptic plasticity based on spike timing. *Biological Cybernetics*, 98(6):459–478, Jun 2008.

[11] T. V. P. Bliss and G. L. Collingridge. A synaptic model of memory: long-term potentiation in the hippocampus. *Nature*, 361(6407):31–39, Jan 1993.

[12] Alain Artola and Wolf Singer. Long-term depression of excitatory synaptic transmission and its relationship to long-term potentiation. *Trends in Neurosciences*, 16(11):480–487, 1993.

[13] Eugene M. Izhikevich. Solving the Distal Reward Problem through Linkage of STDP and Dopamine Signaling. *Cerebral Cortex*, 17(10):2443–2452, 01 2007.

[14] Wolfgang Maass. Networks of spiking neurons: The third generation of neural network models. *Neural Networks*, 10(9):1659–1671, 1997.

[15] Egidio Falotico, Lorenzo Vannucci, Alessandro Ambrosano, Ugo Albanese, Stefan Ulbrich, Juan Camilo Vasquez Tieck, Georg Hinkel, Jacques Kaiser, Igor Peric, Oliver Denninger, Nino Cauli, Murat Kirtay, Arne Roennau, Gudrun Klinker, Axel Von Arnim, Luc Guyot, Daniel Peppicelli, Pablo Martínez-Cañada, Eduardo Ros, Patrick Maier, Sandro Weber, Manuel Huber, David Plecher, Florian Röhrbein, Stefan Deser, Alina Roitberg, Patrick van der Smagt, Rüdiger Dillman, Paul Levi, Cecilia Laschi, Alois C. Knoll, and Marc-Oliver Gewaltig. Connecting artificial brains to robots in a comprehensive simulation framework: The neurorobotics platform. *Frontiers in neurorobotics*, 11:2–2, Jan 2017. 28179882[pmid].

[16] Yongqiang Cao, Yang Chen, and Deepak Khosla. Spiking deep convolutional neural networks for energy-efficient object recognition. *International Journal of Computer Vision*, 113(1):54–66, May 2015.

[17] Amirhossein Tavanaei, Masoud Ghodrati, Saeed Reza Kheradpisheh, Timothée Masquelier, and Anthony S. Maida. Deep learning in spiking neural networks. *CoRR*, abs/1804.08150, 2018.

[18] Dario Floreano and Claudio Mattiussi. *Evolution of Spiking Neural Controllers for Autonomous Vision-Based Robots*, pages 38–61. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001.

[19] David Howard and Alberto Elfes. Evolving spiking networks for turbulence-tolerant quadrotor control. *Artificial Life 14*, 2014.

[20] Erik Steur, Thijs Vromen, and Henk Nijmeijer. *Adaptive Training of Neural Networks for Control of Autonomous Mobile Robots*, pages 387–405. Springer International Publishing, Cham, 2017.

[21] Zhenshan Bing, Claus Meschede, Florian Röhrbein, Kai Huang, and Alois C. Knoll. A survey of robotics control based on learning-inspired spiking neural networks. *Frontiers in Neurorobotics*, 12:35, 2018.

[22] Nikola K. Kasabov. Time-space, spiking neural networks and brain-inspired artificial intelligence. *Springer-Verlag Berlin Heidelberg*, 2019.

[23] Paul Tonelli and Jean-Baptiste Mouret. On the relationships between generative encodings, regularity, and learning abilities when evolving plastic artificial neural networks. *PLOS ONE*, 8(11):1–12, 11 2013.

[24] E. Nichols, L. J. McDaid, and N. Siddique. Biologically inspired SNN for robot control. *IEEE Transactions on Cybernetics*, 43(1):115–128, Feb 2013.

[25] Bernardo Fichera. *Spiking neural network controller for flight stabilization of a microrobotic bee*. PhD thesis, POLITesi, 2017.

[26] Maxence Bouvier, Alexandre Valentian, Thomas Mesquida, Francois Rummens, Marina Reyboz, Elisa Vianello, and Edith Beigne. Spiking neural networks hardware implementations and challenges: A survey. *J. Emerg. Technol. Comput. Syst.*, 15(2):22:1–22:35, April 2019.

[27] Sylvain Koos, Jean-Baptiste Mouret, and Stéphane Doncieux. Crossing the reality gap in evolutionary robotics by promoting transferable controllers. In *Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation*, GECCO '10, page 119–126, New York, NY, USA, 2010. Association for Computing Machinery.

[28] Jean-Baptiste Mouret and Konstantinos Chatzilygeroudis. 20 Years of Reality Gap: A Few Thoughts about Simulators in Evolutionary Robotics. In *Workshop "Simulation in Evolutionary Robotics", Genetic and Evolutionary Computation Conference*, Berlin, Germany, 2017.

[29] Wulfram Gerstner and Werner M. Kistler. Mathematical formulations of hebbian learning. *Biological Cybernetics*, 87(5):404–415, Dec 2002.

[30] Dario Floreano, Phil Husbands, and Stefano Nolfi. *Evolutionary Robotics*, pages 1423–1451. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[31] Peng Cheng, James Keller, and Vijay Kumar. Time-optimal UAV trajectory planning for 3D urban structure coverage. In *2008 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2750–2757, Sep. 2008.

[32] Matthias Müller, Guohao Li, Vincent Casser, Neil Smith, Dominik L. Michels, and Bernard Ghanem. Learning a controller fusion network by online trajectory filtering for vision-based UAV racing. *CoRR*, abs/1904.08801, 2019.

[33] Shubhani Aggarwal and Neeraj Kumar. Path planning techniques for unmanned aerial vehicles: A review, solutions, and challenges. *Computer Communications*, 149:270–299, 2020.

[34] Jung-Woo Park, Hyon-Dong Oh, and Min-Jea Tahk. UAV collision avoidance based on geometric approach. In *2008 SICE Annual Conference*, pages 2122–2126, Aug 2008.

[35] Nils Gageik, Paul Benz, and Sergio Montenegro. Obstacle detection and collision avoidance for a UAV with complementary low-cost sensors. *IEEE Access*, 3:599–609, 2015.

[36] Mitch Bryson and Salah Sukkarieh. Building a robust implementation of bearing-only inertial SLAM for a UAV. *Journal of Field Robotics*, 24(1-2):113–143, 2007.

[37] Patrik Schmuck and Margarita Chli. Multi-UAV collaborative monocular SLAM. In *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pages 3863–3870, May 2017.

[38] K. J. Åström and Thomas J. McAvoy. Intelligent control. *Journal of Process Control*, 2(3):115–127, 1992.

[39] Panos J Antsaklis. Intelligent control. *Encyclopedia of Electrical and Electronics Engineering*, 10:493–503, 1997.

[40] Bruce T Clough. Metrics, schmetrics! how the heck do you determine a UAV's autonomy anyway? Technical report, Air Force Research Lab Wright-Patterson AFB OH, 2002.

[41] Farid Kendoul. Survey of advances in guidance, navigation, and control of unmanned rotorcraft systems. *Journal of Field Robotics*, 29(2):315–378, 2012.

[42] Karl Johan Åström and Richard M Murray. *Feedback systems: an introduction for scientists and engineers*. Princeton University Press, 2nd edition, 2021.

[43] A. Knoll and F. Walter. Neurorobotics – a unique opportunity for ground breaking research. Technical report, Chair of Robotics, Artificial Intelligence and Real-Time Systems, 2019.

[44] Stefano Nolfi and Dario Floreano. *Evolutionary Robotics: The Biology,Intelligence,and Technology*. MIT Press, Cambridge, MA, USA, 2000.

[45] Jean-Christophe Zufferey, Dario Floreano, Matthijs van Leeuwen, and Tancredi Merenda. *Evolving Vision-Based Flying Robots*, pages 592–600. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002.

[46] D. Howard. A platform that directly evolves multirotor controllers. *IEEE Transactions on Evolutionary Computation*, 21(6):943–955, Dec 2017.

[47] Gerard David Howard. On self-adaptive rate restarts for evolutionary robotics with real rotorcraft. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '17, page 123–130, New York, NY, USA, 2017. Association for Computing Machinery.

[48] R. de Nardi, J. Togelius, O. E. Holland, and S. M. Lucas. Evolution of neural networks for helicopter control: Why modularity matters. In *2006 IEEE International Conference on Evolutionary Computation*, pages 1799–1806, 2006.

[49] William Koch, Renato Mancuso, Richard West, and Azer Bestavros. Reinforcement learning for UAV attitude control. *ACM Trans. Cyber-Phys. Syst.*, 3(2), February 2019.

[50] P. Pounds, R. Mahony, and P. Corke. Modelling and control of a large quadrotor robot. *Control Engineering Practice*, 18(7):691 – 699, 2010. Special Issue on Aerial Robotics.

[51] A. Alaimo, V. Artale, C. Milazzo, A. Ricciardello, and L. Trefiletti. Mathematical modeling and control of a hexacopter. In *2013 International Conference on Unmanned Aircraft Systems (ICUAS)*, pages 1043–1050, May 2013.

[52] Matthew Garratt and Sreenatha Anavatti. Non-linear control of heave for an unmanned helicopter using a neural network. *Journal of Intelligent & Robotic Systems*, 66(4):495–504, Jun 2012.

[53] Juan Cristóbal Zagal, Javier Ruiz del Solar, and Paul Vallejos. Back to reality: Crossing the reality gap in evolutionary robotics. *IFAC Proceedings Volumes*, 37(8):834 – 839, 2004. IFAC/EURON Symposium on Intelligent Autonomous Vehicles, Lisbon, Portugal, 5-7 July 2004.

[54] Andrew Y. Ng, H. J. Kim, Michael I. Jordan, and Shankar Sastry. Autonomous helicopter flight via reinforcement learning. In S. Thrun, L. K. Saul, and P. B. Schölkopf, editors, *Advances in Neural Information Processing Systems 16*, pages 799–806. MIT Press, 2004.

[55] Andrew Y. Ng, Adam Coates, Mark Diel, Varun Ganapathi, Jamie Schulte, Ben Tse, Eric Berger, and Eric Liang. *Autonomous Inverted Helicopter Flight via Reinforcement Learning*, pages 363–372. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[56] Nathan V. Hoffer, Calvin Coopmans, Austin M. Jensen, and YangQuan Chen. A survey and categorization of small low-cost unmanned aerial vehicle system identification. *Journal of Intelligent & Robotic Systems*, 74(1):129–145, Apr 2014.

[57] A. Punjani and P. Abbeel. Deep learning helicopter dynamics models. In *2015 IEEE International Conference on Robotics and Automation (ICRA)*, pages 3223–3230, 2015.

[58] Greg Welch, Gary Bishop, et al. An introduction to the kalman filter. Technical report, Department of Computer Science, University of North Carolina at Chapel Hill, 1995.

[59] Steven W Smith et al. *The scientist and engineer's guide to digital signal processing*, volume 14. California Technical Pub. San Diego, 1997.

[60] Hugues Garnier and Liuping Wang, editors. *Identification of Continuous-time Models from Sampled Data*. Springer, 2008.

[61] Md Meftahul Ferdaus, Sreenatha G. Anavatti, Matthew A. Garratt, and Mahardhika Pratama. Fuzzy clustering based nonlinear system identification and controller development of pixhawk based quadcopter. In *2017 Ninth International Conference on Advanced Computational Intelligence (ICACI)*, pages 223–230, 2017.

[62] Mahendra Kumar Samal, Sreenatha Anavatti, and Matthew Garratt. Neural network based system identification for autonomous flight of an eagle helicopter. *IFAC Proceedings Volumes*, 41(2):7421 – 7426, 2008. 17th IFAC World Congress.

[63] Syariful S. Shamsudin and XiaoQi Chen. Identification of an unmanned helicopter system using optimised neural network structure. *International Journal of Modelling, Identification and Control*, 17(3):223–241, 2012.

[64] Mahendra Samal. *Neural network based identification and control of an unmanned helicopter*. PhD thesis, Engineering & Information Technology, Australian Defence Force Academy, UNSW, 2009.

[65] S. Bennett. Development of the PID controller. *IEEE Control Systems Magazine*, 13(6):58–62, 1993.

[66] K. J. Åström and T. Hägglund. Revisiting the Ziegler-Nichols step response method for PID control. *Journal of Process Control*, 14(6):635–650, 2004.

[67] S. Bouabdallah, A. Noth, and R. Siegwart. PID vs LQ control techniques applied to an indoor micro quadrotor. In *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (IEEE Cat. No.04CH37566)*, volume 3, pages 2451–2456 vol.3, Sept 2004.

[68] A. L. Salih, M. Moghavvemi, H. A. F. Mohamed, and K. S. Gaeid. Modelling and PID controller design for a quadrotor unmanned air vehicle. In *2010 IEEE International Conference on Automation, Quality and Testing, Robotics (AQTR)*, volume 1, pages 1–5, May 2010.

[69] Jack F Shepherd III and Kagan Tumer. Robust neuro-control for a micro quadrotor. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1131–1138, Portland, OR, July 2010.

[70] Nguyen Khoi Tran. Modeling and control of a quadrotor in a wind field. Master's thesis, McGill University, 2016.

[71] F. Santoso, M. A. Garratt, and S. G. Anavatti. A self-learning ts-fuzzy system based on the c-means clustering technique for controlling the altitude of a hexacopter unmanned aerial vehicle. In *2017 International Conference on Advanced Mechatronics, Intelligent Manufacture, and Industrial Automation (ICAMIMIA)*, pages 46–51, Oct 2017.

[72] Vu Phi Tran, Fendy Santoso, Matthew A. Garratt, and Ian R. Petersen. Adaptive second-order strictly negative imaginary controllers based on the interval type-2 fuzzy self-tuning systems for a hovering quadrotor with uncertainties. *IEEE/ASME Transactions on Mechatronics*, 25(1):11–20, 2020.

[73] Lorenz Meier, Dominik Honegger, and Marc Pollefeys. Px4: A node-based multithreaded open source robotics framework for deeply embedded platforms. In *2015 IEEE International Conference on Robotics and Automation (ICRA)*, pages 6235–6240, 2015.

[74] Jianhua Yang, Wei Lu, and Wenqi Liu. *PID Controller Based on the Artificial Neural Network*, pages 144–149. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[75] Abdel Badie Sharkawy. Genetic fuzzy self-tuning PID controllers for antilock braking systems. *Engineering Applications of Artificial Intelligence*, 23(7):1041 – 1052, 2010.

[76] Batıkan E Demir, Raif Bayir, and Fecir Duran. Real-time trajectory tracking of an unmanned aerial vehicle using a self-tuning fuzzy proportional integral derivative controller. *International Journal of Micro Air Vehicles*, 8(4):252–268, 2016.

[77] D. Howard and T. Merz. A platform for the direct hardware evolution of quadcopter controllers. In *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 4614–4619, Sept 2015.

[78] Daewon Park, Hyona Yu, Nguyen Xuan-Mung, Junyong Lee, and Sung Kyung Hong. Multicopter PID attitude controller gain autotuning through reinforcement learning neural networks. In *Proceedings of the 2019 2nd International Conference on Control and Robot Technology*, ICCRT 2019, page 80–84, New York, NY, USA, 2019. Association for Computing Machinery.

[79] Alberto Bemporad, Manfred Morari, Vivek Dua, and Efstratios N. Pistikopoulos. The explicit linear quadratic regulator for constrained systems. *Automatica*, 38(1):3 – 20, 2002.

[80] Dingyü Xue, YangQuan Chen, and Derek P. Atherton. *Linear Feedback Control: Analysis and Design with MATLAB*. Society for Industrial and Applied Mathematics, 2007.

[81] Chingiz Hajiyev; Sitki Yenal Vural;. LQR controller with Kalman estimator applied to UAV longitudinal dynamics. *Positioning*, Vol.04No.01, 2013.

[82] Elias Reyes-Valeria, Rogerio Enriquez-Caldera, Sergio Camacho-Lara, and Jose Guichard. Lqr control for a quadrotor using unit quaternions: Modeling and simulation. In *CONIELECOMP 2013, 23rd International Conference on Electronics, Communications and Computing*, pages 172–178, 2013.

[83] M. W. Mueller and R. D'Andrea. Stability and control of a quadrocopter despite the complete loss of one, two, or three propellers. In *2014 IEEE International Conference on Robotics and Automation (ICRA)*, pages 45–52, May 2014.

[84] T. S. Clawson, S. Ferrari, S. B. Fuller, and R. J. Wood. Spiking neural network (SNN) control of a flapping insect-scale robot. In *2016 IEEE 55th Conference on Decision and Control (CDC)*, pages 3381–3388, Dec 2016.

[85] Ming Chen and Mihai Huzmezan. A combined MBPC/2 DOF H ∞ controller for a quad rotor UAV. In *AIAA Guidance, Navigation, and Control Conference and Exhibit*, 2003.

[86] J. Gadewadikar, Frank Lewis, Kamesh Subbarao, and Ben M. Chen. Structured h-infinity command and control-loop design for unmanned helicopters. *Journal of Guidance, Control, and Dynamics*, 31(4):1093–1102, 2008.

[87] J. López, R. Dormido, S. Dormido, and J. P. Gómez. A robust H ∞ controller for an UAV flight control system. *TheScientificWorldJournal*, 2015:403236–403236, 2015. 26221622[pmid].

[88] Kai Masuda and Kenji Uchiyama. Robust control design for quad tiltwing UAV. *Aerospace*, 5(1), 2018.

[89] Holger Voos. Nonlinear control of a quadrotor micro-UAV using feedback-linearization. In *2009 IEEE International Conference on Mechatronics*, pages 1–6, 2009.

[90] Qing-Li Zhou, Youmin Zhang, Camille-Alain Rabbath, and Didier Theilliol. Design of feedback linearization control and reconfigurable control allocation with application to a quadrotor UAV. In *2010 Conference on Control and Fault-Tolerant Systems (SysTol)*, pages 371–376, 2010.

[91] Bilal Ahmed, Hemanshu R. Pota, and Matt Garratt. Flight control of a rotary wing UAV using backstepping. *International Journal of Robust and Nonlinear Control*, 20(6):639–658, 2010.

[92] S. Bouabdallah and R. Siegwart. Backstepping and sliding-mode techniques applied to an indoor micro quadrotor. In *Proceedings of the 2005 IEEE International Conference on Robotics and Automation*, pages 2247–2252, April 2005.

[93] Kostas Alexis, Christos Papachristos, Roland Siegwart, and Anthony Tzes. Robust model predictive flight control of unmanned rotorcrafts. *Journal of Intelligent & Robotic Systems*, 81(3):443–469, Mar 2016.

[94] Bourhane Kadmiry. Fuzzy control for an unmanned helicopter, 2002. Report code: LiU-Tek-Lic-2002:11. The format of the electronic version of this thesis differs slightly from the printed one: this is due mainly to font compatibility. The figures and body of the thesis are remaining unchanged.

[95] Arbab Nighat Khizer, Dai Yaping, Syed Amjad Ali, and Xu Xiangyang. Stable hovering flight for a small unmanned helicopter using fuzzy control. *Mathematical Problems in Engineering*, 2014, 2014.

[96] M. M. Ferdaus, M. Pratama, S. G. Anavatti, and M. Garratt. A generic self-evolving neuro-fuzzy controller based high-performance hexacopter altitude control system. In *2018 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pages 2784–2791, Oct 2018.

[97] A. Sarabakha, C. Fu, E. Kayacan, and T. Kumbasar. Type-2 fuzzy logic controllers made even simpler: From design to deployment for UAVs. *IEEE Transactions on Industrial Electronics*, 65(6):5069–5077, 2018.

[98] Ayad Al-Mahturi, Fendy Santoso, Matthew A. Garratt, and Sreenatha G. Anavatti. A robust self-adaptive interval type-2 ts fuzzy

logic for controlling multi-input-multi-output nonlinear uncertain dynamical systems. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, pages 1–12, 2020.

[99] Jie Xu, Tao Du, Michael Foshey, Beichen Li, Bo Zhu, Adriana Schulz, and Wojciech Matusik. Learning to fly: Computational controller design for hybrid UAVs with reinforcement learning. *ACM Trans. Graph.*, 38(4):42:1–42:12, July 2019.

[100] Boumediene Selma and Samira Chouraqui. Neuro-fuzzy controller to navigate an unmanned vehicle. *SpringerPlus*, 2(1):188, 2013.

[101] Yesim Oniz and Okyay Kaynak. Control of a direct drive robot using fuzzy spiking neural networks with variable structure systems-based learning algorithm. *Neurocomputing*, 149:690 – 699, 2015.

[102] João Pedro Carvalho de Souza, André Luís Marques Marcato, Eduardo Pestana de Aguiar, Marco Aurélio Jucá, and Alexandre Menezes Teixeira. Autonomous landing of UAV based on artificial neural network supervised by fuzzy logic. *Journal of Control, Automation and Electrical Systems*, 30(4):522–531, Aug 2019.

[103] M. M. Ferdaus, M. A. Hady, M. Pratama, H. Kandath, and S. G. Anavatti. Redpac: A simple evolving neuro-fuzzy-based intelligent control framework for quadcopter. In *2019 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE)*, pages 1–7, 2019.

[104] Kevin M. Passino and Stephen Yurkovich. *Fuzzy Control*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1997.

[105] G. Buskey, G. Wyeth, and J. Roberts. Autonomous helicopter hover using an artificial neural network. In *Proceedings 2001 ICRA. IEEE International Conference on Robotics and Automation (Cat. No.01CH37164)*, volume 2, pages 1635–1640 vol.2, 2001.

[106] J. Dunfied, M. Tarbouchi, and G. Labonte. Neural network based control of a four rotor helicopter. In *Industrial Technology, 2004. IEEE ICIT '04. 2004 IEEE International Conference on*, volume 3, pages 1543–1548 Vol. 3, Dec 2004.

[107] Howard B. Demuth, Mark H. Beale, Orlando De Jess, and Martin T. Hagan. *Neural Network Design*. Martin Hagan, USA, 2nd edition, 2014.

[108] Warren S. McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943.

[109] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning internal representations by error propagation. Technical report, DTIC Document, 1985.

[110] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 61:85–117, 2015.

[111] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25:1097–1105, 2012.

[112] Sepp Hochreiter and Jürgen Schmidhuber. Long Short-Term Memory. *Neural Computation*, 9(8):1735–1780, 11 1997.

[113] W. Maass. Fast sigmoidal networks via spiking neurons. *Neural Computation*, 9(2):279–304, Feb 1997.

[114] D. I. Perrett, E. T. Rolls, and W. Caan. Visual neurones responsive to faces in the monkey temporal cortex. *Experimental Brain Research*, 47(3):329–342, Sep 1982.

[115] Z. F. Mainen and T. J. Sejnowski. Reliability of spike timing in neocortical neurons. *Science*, 268(5216):1503–1506, 1995.

[116] Paul A. Merolla, John V. Arthur, Rodrigo Alvarez-Icaza, Andrew S. Cassidy, Jun Sawada, Filipp Akopyan, Bryan L. Jackson, Nabil Imam, Chen Guo, Yutaka Nakamura, Bernard Brezzo, Ivan Vo, Steven K. Esser, Rathinakumar Appuswamy, Brian Taba, Arnon Amir, Myron D. Flickner, William P. Risk, Rajit Manohar, and Dharmendra S. Modha. A million spiking-neuron integrated circuit with a scalable communication network and interface. *Science*, 345(6197):668–673, 2014.

[117] X. Jin, M. Lujan, L. A. Plana, S. Davies, S. Temple, and S. B. Furber. Modeling spiking neural networks on SpiNNaker. *Computing in Science Engineering*, 12(5):91–97, Sept 2010.

[118] S. B. Furber, D. R. Lester, L. A. Plana, J. D. Garside, E. Painkras, S. Temple, and A. D. Brown. Overview of the SpiNNaker system architecture. *IEEE Transactions on Computers*, 62(12):2454–2467, Dec 2013.

[119] Giacomo Indiveri, Bernabe Linares-Barranco, Tara Hamilton, Andrè van Schaik, Ralph Etienne-Cummings, Tobi Delbruck, Shih-Chii Liu, Piotr Dudek, Philipp Häfliger, Sylvie Renaud, Johannes Schemmel, Gert Cauwenberghs, John Arthur, Kai Hynna, Fopefolu Folowosele, Sylvain SAÏGHI, Teresa Serrano-Gotarredona, Jayawan Wijekoon, Yingxue Wang, and Kwabena Boahen. Neuromorphic silicon neuron circuits. *Frontiers in Neuroscience*, 5:73, 2011.

[120] Jing Pei, Lei Deng, Sen Song, Mingguo Zhao, Youhui Zhang, Shuang Wu, Guanrui Wang, Zhe Zou, Zhenzhi Wu, Wei He, Feng Chen, Ning Deng, Si Wu, Yu Wang, Yujie Wu, Zheyu Yang, Cheng Ma, Guoqi Li, Wentao Han, Huanglong Li, Huaqiang Wu, Rong Zhao, Yuan Xie, and Luping Shi. Towards artificial general intelligence with hybrid Tianjic chip architecture. *Nature*, 572(7767):106–111, 2019.

[121] L. Deng, G. Wang, G. Li, S. Li, L. Liang, M. Zhu, Y. Wu, Z. Yang, Z. Zou, J. Pei, Z. Wu, X. Hu, Y. Ding, W. He, Y. Xie, and L. Shi. Tianjic: A unified and scalable chip bridging spike-based and continuous neural computation. *IEEE Journal of Solid-State Circuits*, 55(8):2228–2246, 2020.

[122] Patrick Lichtsteiner, Christoph Posch, and Tobi Delbruck. A 128 × 128 120 db 15 μs latency asynchronous temporal contrast vision sensor. *IEEE Journal of Solid-State Circuits*, 43(2):566–576, Feb 2008.

[123] Christian Brandli, Raphael Berner, Minhao Yang, Shih-Chii Liu, and Tobi Delbruck. A 240 × 180 130 db 3 μs latency global shutter spatiotemporal vision sensor. *IEEE Journal of Solid-State Circuits*, 49(10):2333–2341, Oct 2014.

[124] Ran Cheng, Khalid B. Mirza, and Konstantin Nikolic. Neuromorphic robotic platform with visual input, processor and actuator, based on spiking neural networks. *Applied System Innovation*, 3(2), 2020.

[125] Riccardo Massa, Alberto Marchisio, Maurizio Martina, and Muhammad Shafique. An efficient spiking neural network for recognizing gestures with a DVS camera on the Loihi neuromorphic processor. In *2020 International Joint Conference on Neural Networks (IJCNN)*, pages 1–9, July 2020.

[126] Hélène Paugam-Moisy and Sander Bohte. *Computing with Spiking Neuron Networks*, pages 335–376. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[127] Eugene M Izhikevich. *Dynamical systems in neuroscience*. MIT press, 2007.

[128] Dominik Thalmeier, Marvin Uhlmann, Hilbert J. Kappen, and Raoul-Martin Memmesheimer. Learning universal computations with spikes. *PLOS Computational Biology*, 12(6):1–29, 06 2016.

[129] S. J. Martin, P. D. Grimwood, and R. G. M. Morris. Synaptic plasticity and memory: An evaluation of the hypothesis. *Annual Review of Neuroscience*, 23(1):649–711, 2000. PMID: 10845078.

[130] André Grüning and Er M. Bohte. Spiking neural networks: Principles and challenges, 2014.

[131] Sander M. Bohte, Joost N. Kok, and Han La Poutré. Spikeprop: Backpropagation for networks of spiking neurons. *ResearchGate*, 2000.

[132] Filip Ponulak. *Supervised learning in spiking neural networks with ReSuMe method*. PhD thesis, Poznan University of Technology, 2006.

[133] Peter O'Connor and Max Welling. Deep spiking networks. *CoRR*, abs/1602.08323, 2016.

[134] Jun Haeng Lee, Tobi Delbruck, and Michael Pfeiffer. Training deep spiking neural networks using backpropagation. *Frontiers in Neuroscience*, 10:508, 2016.

[135] Eric Hunsberger and Chris Eliasmith. Spiking deep networks with LIF neurons. *CoRR*, abs/1510.08829, 2015.

[136] Dario Floreano, Peter Dürr, and Claudio Mattiussi. Neuroevolution: from architectures to learning. *Evolutionary Intelligence*, 1(1):47–62, Mar 2008.

[137] Agoston E Eiben, James E Smith, et al. *Introduction to Evolutionary Computing*. Natural Computing Series. Springer, Berlin, Heidelberg, second edition, 2015.

[138] A. Vandesompele, F. Walter, and F. Röhrbein. Neuro-evolution of spiking neural networks on SpiNNaker neuromorphic hardware. In *2016 IEEE Symposium Series on Computational Intelligence (SSCI)*, pages 1–6, Dec 2016.

[139] Abdulrazak Yahya Saleh, Siti Mariyam Shamsuddin, and Haza Nuzly Abdull Hamed. A hybrid differential evolution algorithm for parameter tuning of evolving spiking neural network. *International Journal of Computational Vision and Robotics*, 7(1-2):20–34, 2017.

[140] Kenneth O. Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary Computation*, 10(2):99–127, 2002.

[141] Stefan Schliebs and Nikola Kasabov. Evolving spiking neural network—a survey. *Evolving Systems*, 4(2):87–98, Jun 2013.

[142] Donald Olding Hebb. *The organization of behavior: A neuropsychological theory*. John Wiley & Sons, 1949.

[143] Eugene M. Izhikevich and Niraj S. Desai. Relating STDP to BCM. *Neural Computation*, 15(7):1511–1523, 2003.

[144] Eugene M. Izhikevich. Polychronization: Computation with spikes. *Neural Computation*, 18(2):245–282, 2006. PMID: 16378515.

[145] Michael E. Hasselmo. Neuromodulation: acetylcholine and memory consolidation. *Trends in Cognitive Sciences*, 3(9):351–359, 1999.

[146] Michael Pfeiffer, Bernhard Nessler, Rodney J. Douglas, and Wolfgang Maass. Reward-modulated hebbian learning of decision making. *Neural Comput.*, 22(6):1399–1444, June 2010.

[147] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[148] A. Soltoggio, P. Durr, C. Mattiussi, and D. Floreano. Evolving neuromodulatory topologies for reinforcement learning-like problems. In *2007 IEEE Congress on Evolutionary Computation*, pages 2471–2478, Sep. 2007.

[149] Andrea Soltoggio, John A. Bullinaria, Claudio Mattiussi, Peter Dürr, and Dario Floreano. Evolutionary advantages of neuromodulated plasticity in dynamic, reward- based scenarios. *Proceedings of the 11th International Conference on Artificial Life (Alife XI)*, pages 569–576, 2008.

[150] P. Dürr, C. Mattiussi, A. Soltoggio, and D. Floreano. Evolvability of neuromodulated learning for robots. In *2008 ECSIS Symposium on*

Learning and Adaptive Behaviors for Robotic Systems (LAB-RS), pages 41–46, Aug 2008.

[151] C. Muhammad and B. Samanta. Neuromodulation based control of autonomous robots in ROS environment. In *2014 IEEE Symposium on Computational Intelligence, Cognitive Algorithms, Mind, and Brain (CCMB)*, pages 16–23, Dec 2014.

[152] Yael Niv, Daphna Joel, Isaac Meilijson, and Eytan Ruppin. Evolution of reinforcement learning in uncertain environments: A simple explanation for complex foraging behaviors. *Adaptive Behavior*, 10(1):5–24, 2002.

[153] Sebastian Risi and Kenneth O. Stanley. Indirectly encoding neural plasticity as a pattern of local rules. In Stéphane Doncieux, Benoît Girard, Agnès Guillot, John Hallam, Jean-Arcady Meyer, and Jean-Baptiste Mouret, editors, *From Animals to Animats 11*, pages 533–543, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[154] Elias Najarro and Sebastian Risi. Meta-learning through hebbian plasticity in random networks. *CoRR*, abs/2007.02686, 2020.

[155] R.J.C. Bosman, W.A. van Leeuwen, and B. Wemmenhove. Combining hebbian and reinforcement learning in a minibrain model. *Neural Networks*, 17(1):29–36, 2004.

[156] Jens Kober and Jan Peters. *Reinforcement Learning in Robotics: A Survey*, pages 579–610. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[157] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, Jan 2016.

[158] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of go without human knowledge. *Nature*, 550(7676):354–359, Oct 2017.

[159] Michael A. Farries and Adrienne L. Fairhall. Reinforcement learning with modulated spike timing–dependent synaptic plasticity. *Journal of Neurophysiology*, 98(6):3648–3665, 2007.

[160] Wiebke Potjans, Abigail Morrison, and Markus Diesmann. A Spiking Neural Network Model of an Actor-Critic Learning Agent. *Neural Computation*, 21(2):301–339, 02 2009.

[161] Karim El-Laithy and Martin Bogdan. A hebbian-based reinforcement learning framework for spike-timing-dependent synapses. In Konstantinos Diamantaras, Wlodek Duch, and Lazaros S. Iliadis, editors, *Artificial Neural Networks – ICANN 2010*, pages 160–169, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[162] Nicolas Frémaux, Henning Sprekeler, and Wulfram Gerstner. Reinforcement learning using a continuous time actor-critic framework with spiking neurons. *PLOS Computational Biology*, 9(4):1–21, 04 2013.

[163] Anil Yaman, Giovanni Iacca, Decebal Constantin Mocanu, George Fletcher, and Mykola Pechenizkiy. Learning with delayed synaptic plasticity. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '19, pages 152–160, New York, NY, USA, 2019. ACM.

[164] Pawel Ladosz, Eseoghene Ben-Iwhiwhu, Yang Hu, Nicholas Ketz, Soheil Kolouri, Jeffrey L. Krichmar, Praveen K. Pilly, and Andrea Soltoggio. Deep reinforcement learning with modulated hebbian plus Q network architecture. *CoRR*, abs/1909.09902, 2019.

[165] K. L. Rice, M. A. Bhuiyan, T. M. Taha, C. N. Vutsinas, and M. C. Smith. FPGA implementation of Izhikevich spiking neural networks for character recognition. In *2009 International Conference on Reconfigurable Computing and FPGAs*, pages 451–456, Dec 2009.

[166] Z. Wang, Y. Ma, Z. Dong, N. Zheng, and P. Ren. Spiking locality-sensitive hash: Spiking computation with phase encoding method. In *2018 International Joint Conference on Neural Networks (IJCNN)*, pages 1–7, July 2018.

[167] G. Srinivasan, C. Lee, A. Sengupta, P. Panda, S. S. Sarwar, and K. Roy. Training deep spiking neural networks for energy-efficient neuromorphic computing. In *ICASSP 2020 - 2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 8549–8553, 2020.

[168] L. F. Abbott, Brian DePasquale, and Raoul-Martin Memmesheimer. Building functional networks of spiking model neurons. *Nature Neuroscience*, 19:350–355, Feb 2016. Perspective.

[169] Dario Floreano, Nicolas Schoeni, Gilles Caprari, and Jesper Blynel. Evolutionary bits'n'spikes. In *Proceedings of the eighth international conference on Artificial life*, pages 335–344, 2003.

[170] R. Batllori, C.B. Laramee, W. Land, and J.D. Schaffer. Evolving spiking neural networks for robot control. *Procedia Computer Science*, 6:329 – 334, 2011. Complex adaptive sysytems.

[171] Gerard David Howard, Larry Bull, and Pier Luca Lanzi. A spiking neural learning classifier system. *CoRR*, abs/1201.3249, 2012.

[172] Victor Erokhin, Gerard David Howard, and Andrew Adamatzky. Organic memristor devices for logic elements with memory. *International Journal of Bifurcation and Chaos*, 22(11):1250283, 2012.

[173] Xiuqing Wang, Zeng-Guang Hou, Anmin Zou, Min Tan, and Long Cheng. A behavior controller based on spiking neural networks for mobile robots. *Neurocomputing*, 71(4):655 – 666, 2008. Neural Networks: Algorithms and Applications 50 Years of Artificial Intelligence: a Neuronal Approach.

[174] Xiuqing Wang, Zeng-Guang Hou, Feng Lv, Min Tan, and Yongji Wang. Mobile robots' modular navigation controller using spiking neural networks. *Neurocomputing*, 134:230 – 238, 2014. Special issue on the 2011 Sino-foreign-interchange Workshop on Intelligence Science and Intelligent Data Engineering (IScIDE 2011) Learning Algorithms and Applications.

[175] Hédi Soula, Aravind Alwan, and Guillaume Beslon. Learning at the edge of chaos : Temporal coupling of spiking neurons controller for autonomous robotic. In *Proceedings of the AAAI spring symposia on developmental robotics*. AAAI Palo Alto, CA, 2005.

[176] Răzvan V. Florian. Reinforcement learning through modulation of spike-timing-dependent synaptic plasticity. *Neural Computation*, 19(6):1468–1502, 2007.

[177] S. Ferrari, B. Mehta, G. Di Muro, A. M. J. VanDongen, and C. Henriquez. Biologically realizable reward-modulated hebbian training for spiking neural networks. In *2008 IEEE International Joint Conference*

on Neural Networks (IEEE World Congress on Computational Intelligence), pages 1780–1786, June 2008.

[178] G. Foderaro, C. Henriquez, and S. Ferrari. Indirect training of a spiking neural network for flight control via spike-timing-dependent synaptic plasticity. In *49th IEEE Conference on Decision and Control (CDC)*, pages 911–917, Dec 2010.

[179] X. Zhang, Z. Xu, C. Henriquez, and S. Ferrari. Spike-based indirect training of a spiking neural network-controlled virtual insect. In *52nd IEEE Conference on Decision and Control*, pages 6798–6805, Dec 2013.

[180] Di Hu, Xu Zhang, Ziye Xu, S. Ferrari, and P. Mazumder. Digital implementation of a spiking neural network (SNN) capable of spike-timing-dependent plasticity (STDP) learning. In *14th IEEE International Conference on Nanotechnology*, pages 873–876, Aug 2014.

[181] P. Mazumder, D. Hu, I. Ebong, X. Zhang, Z. Xu, and S. Ferrari. Digital implementation of a virtual insect trained by spike-timing dependent plasticity. *Integration, the VLSI Journal*, 54(Supplement C):109 – 117, 2016.

[182] Kevin Y. Ma, Pakpong Chirarattananon, Sawyer B. Fuller, and Robert J. Wood. Controlled flight of a biologically inspired, insect-scale robot. *Science*, 340(6132):603–607, 2013.

[183] Taylor S. Clawson, Terrence C. Stewart, Chris Eliasmith, and Silvia Ferrari. An adaptive spiking neural controller for flapping insect-scale robots. In *2017 IEEE Symposium Series on Computational Intelligence, SSCI 2017, Honolulu, HI, USA, November 27 - Dec. 1, 2017*, pages 1–7, 2017.

[184] Zhenshan Bing, Ivan Baumann, Zhuangyi Jiang, Kai Huang, Caixia Cai, and Alois Knoll. Supervised learning in SNN via reward-modulated spike-timing-dependent plasticity for a target reaching vehicle. *Frontiers in Neurorobotics*, 13:18, 2019.

[185] M.C. Schut, E. Haasdijk, and A.E. Eiben. What is situated evolution? In *2009 IEEE Congress on Evolutionary Computation*, pages 3277–3284, 2009.

[186] Dario Floreano, Jean-Christophe Zufferey, and Jean-Daniel Nicoud. From wheels to wings with evolutionary spiking circuits. *Artificial Life*, 11(1-2):121–138, 2005.

[187] H. Hagras, A. Pounds-Cornish, M. Colley, V. Callaghan, and G. Clarke. Evolving spiking neural network controllers for autonomous robots. In *Robotics and Automation, 2004. Proceedings. ICRA '04. 2004 IEEE International Conference on*, volume 5, pages 4620–4626 Vol.5, April 2004.

[188] Peter Trhan. The application of spiking neural networks in autonomous robot control. *Computing and Informatics*, 29(5):823–847, 2012.

[189] Patrick Rocke, Brian McGinley, Fearghal Morgan, and John Maher. *Reconfigurable Hardware Evolution Platform for a Spiking Neural Network Robotics Controller*, pages 373–378. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[190] Stefano Nolfi and Dario Floreano. Learning and evolution. *Autonomous Robots*, 7(1):89–113, Jul 1999.

[191] Andrea Soltoggio, Kenneth O. Stanley, and Sebastian Risi. Born to learn: The inspiration, progress, and future of evolved plastic artificial neural networks. *Neural Networks*, 108:48 – 67, 2018.

[192] Joseba Urzelai and Dario Floreano. Evolution of adaptive synapses: Robots with fast adaptive behavior in new environments. *Evolutionary Computation*, 9(4):495–524, 2001.

[193] Paul Tonelli and Jean-Baptiste Mouret. On the relationships between synaptic plasticity and generative systems. In *Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation*, GECCO '11, pages 1531–1538, New York, NY, USA, 2011. Association for Computing Machinery.

[194] G. Howard, E. Gale, L. Bull, B. de Lacy Costello, and A. Adamatzky. Evolution of plastic learning in spiking networks via memristive connections. *IEEE Transactions on Evolutionary Computation*, 16(5):711–729, Oct 2012.

[195] Anil Yaman, Giovanni Iacca, Decebal Constantin Mocanu, Matt Coler, George Fletcher, and Mykola Pechenizkiy. Evolving Plasticity for Autonomous Learning under Changing Environmental Conditions. *Evolutionary Computation*, pages 1–25, 12 2020.

[196] Oliver J. Coleman and Alan D. Blair. Evolving plastic neural networks for online learning: Review and future directions. In Michael

Thielscher and Dongmo Zhang, editors, *AI 2012: Advances in Artificial Intelligence*, pages 326–337, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[197] Jean-Baptiste Mouret and Paul Tonelli. *Artificial Evolution of Plastic Neural Networks: A Few Key Concepts*, pages 251–261. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[198] Ezequiel Di Paolo. Spike-timing dependent plasticity for evolved robots. *Adaptive Behavior*, 10(3-4):243–263, 2002.

[199] D. Federici. Evolving developing spiking neural networks. In *2005 IEEE Congress on Evolutionary Computation*, volume 1, pages 543–550 Vol.1, Sep. 2005.

[200] Michael L Hines and Nicholas T Carnevale. The NEURON simulation environment. *Neural computation*, 9(6):1179–1209, 1997.

[201] Marc-Oliver Gewaltig and Markus Diesmann. NEST (NEural Simulation Tool). *Scholarpedia*, 2(4):1430, 2007.

[202] Friedemann Zenke and Wulfram Gerstner. Limits to high-speed simulations of spiking neural networks using general-purpose computers. *Frontiers in Neuroinformatics*, 8:76, 2014.

[203] T. Chou, H. J. Kashyap, J. Xing, S. Listopad, E. L. Rounds, M. Beyeler, N. Dutt, and J. L. Krichmar. Carlsim 4: An open source library for large scale, biologically detailed spiking neural network simulation using heterogeneous clusters. In *2018 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8, July 2018.

[204] Hananel Hazan, Daniel J. Saunders, Hassaan Khan, Devdhar Patel, Darpan T. Sanghavi, Hava T. Siegelmann, and Robert Kozma. Bindsnet: A machine learning-oriented spiking neural networks library in python. *Frontiers in Neuroinformatics*, 12:89, 2018.

[205] D Gamez, A K Fidjeland, and E Lazdins. iSpike: a spiking neural interface for the iCub robot. *Bioinspiration & Biomimetics*, 7(2):025008, 2012.

[206] A. K. Fidjeland, E. B. Roesch, M. P. Shanahan, and W. Luk. Nemo: A platform for neural modelling of spiking neurons using gpus. In *2009 20th IEEE International Conference on Application-specific Systems, Architectures and Processors*, pages 137–144, 2009.

[207] E. M. Izhikevich. Simple model of spiking neurons. *IEEE Transactions on Neural Networks*, 14(6):1569–1572, Nov 2003.

[208] Alan L Hodgkin and Andrew F Huxley. A quantitative description of membrane current and its application to conduction and excitation in nerve. *The Journal of physiology*, 117(4):500, 1952.

[209] Frédéric Theunissen and John P. Miller. Temporal encoding in nervous systems: A rigorous definition. *Journal of Computational Neuroscience*, 2(2):149–162, Jun 1995.

[210] Eric R. Kandel and Robert D. Hawkins. The biological basis of learning and individuality. *Scientific American*, 267(3):78–87, 1992.

[211] David Pardoe, Michael Ryoo, and Risto Miikkulainen. Evolving neural network ensembles for control problems. In *Proceedings of the 7th Annual Conference on Genetic and Evolutionary Computation*, GECCO '05, pages 1379–1384, New York, NY, USA, 2005. ACM.

[212] Huanneng Qiu, Matthew Garratt, David Howard, and Sreenatha Anavatti. Towards crossing the reality gap with evolved plastic neurocontrollers. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference*, GECCO '20, pages 130–138, New York, NY, USA, 2020. Association for Computing Machinery.

[213] Faustino Gomez and Risto Miikkulainen. Incremental evolution of complex general behavior. *Adaptive Behavior*, 5(3-4):317–342, 1997.

[214] H. Qiu, M. Garratt, D. Howard, and S. Anavatti. Evolving spiking neurocontrollers for UAVs. In *2020 IEEE Symposium Series on Computational Intelligence (SSCI)*, pages 1928–1935, 2020.

[215] André Brandão, Pedro Pires, and Petia Georgieva. Reinforcement learning and neuroevolution in flappy bird game. In Aythami Morales, Julian Fierrez, José Salvador Sánchez, and Bernardete Ribeiro, editors, *Pattern Recognition and Image Analysis*, pages 225–236, Cham, 2019. Springer International Publishing.

[216] Madhavun Candadai Vasu and Eduardo J. Izquierdo. Information bottleneck in control tasks with recurrent spiking neural networks. In Alessandra Lintas, Stefano Rovetta, Paul F.M.J. Verschure, and Alessandro E.P. Villa, editors, *Artificial Neural Networks and Machine Learning – ICANN 2017*, pages 236–244, Cham, 2017. Springer International Publishing.

[217] T. S. Kang and A. Banerjee. Learning deterministic spiking neuron feedback controllers. In *2017 International Joint Conference on Neural Networks (IJCNN)*, pages 2443–2450, May 2017.

[218] A. G. Barto, R. S. Sutton, and C. W. Anderson. Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-13(5):834–846, Sept 1983.

[219] Jong-Woo Choi and Seung-Ki Sul. Inverter output voltage synthesis using novel dead time compensation. *IEEE Transactions on Power Electronics*, 11(2):221–227, 1996.

[220] G. Flores Colunga, J. A. Guerrero, J. Escareño, and R. Lozano. *Modeling and Control of Mini UAV*, chapter 6, pages 99–134. John Wiley & Sons, Ltd, 2012.

[221] Pierre Merriaux, Yohan Dupuis, Rémi Boutteau, Pascal Vasseur, and Xavier Savatier. A study of vicon system positioning performance. *Sensors*, 17(7), 2017.

[222] M. Quigley. ROS: An open-source robot operating system. In *ICRA 2009*, 2009.

[223] Jyh-Cheng Jeng, Wan-Ling Tseng, and Min-Sen Chiu. A one-step tuning method for PID controllers with robustness specification using plant step-response data. *Chemical Engineering Research and Design*, 92(3):545–558, 2014.

[224] Johannes Meyer, Alexander Sendobry, Stefan Kohlbrecher, Uwe Klingauf, and Oskar von Stryk. Comprehensive simulation of quadrotor UAVs using ROS and Gazebo. In Itsuki Noda, Noriaki Ando, Davide Brugali, and James J. Kuffner, editors, *Simulation, Modeling, and Programming for Autonomous Robots*, pages 400–411, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[225] Julia Badger, Dustin Gooding, Kody Ensley, Kimberly Hambuchen, and Allison Thackston. *ROS in Space: A Case Study on Robonaut 2*, pages 343–373. Springer International Publishing, Cham, 2016.

[226] Wolfgang Hönig and Nora Ayanian. *Flying Multiple UAVs Using ROS*, pages 83–118. Springer International Publishing, Cham, 2017.

[227] Matthew Adam Garratt. *Biologically Inspired Vision and Control for an Autonomous Flying Vehicle*. PhD thesis, Visual Sciences, Research School of Biological Sciences and The Australian National University, 10 2007.

[228] Tomáš Jiřinec. Stabilization and control of unmanned quadcopter, 2011.

[229] Anthony Robert Southey Bramwell, David Balmford, and George Done. *Bramwell's Helicopter Dynamics*. Butterworth-Heinemann, Oxford, second edition edition, 2001.

[230] H. Glauert. A general theory of the autogyro. In *Royal Aeronautical Society*, 1926.

[231] J. G. Leishman. *Principles of Helicopter Aerodynamics*, volume 2nd Edition. Cambridge University Press, New York, 2006.

[232] Zbigniew Michalewicz and Marc Schoenauer. Evolutionary algorithms for constrained parameter optimization problems. *Evolutionary Computation*, 4(1):1–32, 1996.

[233] Philipp Weidel, Mikael Djurfeldt, Renato C. Duarte, and Abigail Morrison. Closed loop interactions between spiking neural network and robotic simulators based on MUSIC and ROS. *Frontiers in Neuroinformatics*, 10:31, 2016.

[234] J. Kaiser, J. C. V. Tieck, C. Hubschneider, P. Wolf, M. Weber, M. Hoff, A. Friedrich, K. Wojtasik, A. Roennau, R. Kohlhaas, R. Dillmann, and J. M. Zöllner. Towards a framework for end-to-end control of a simulated vehicle with spiking neural networks. In *2016 IEEE International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR)*, pages 127–134, Dec 2016.

[235] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. OpenAI Gym. *CoRR*, abs/1606.01540, 2016.

[236] Stephen K. Armah and Sun Yi. *Analysis of Time Delays in Quadrotor Systems and Design of Control*, pages 299–313. Springer International Publishing, Cham, 2017.

[237] Inkyu Sa and Peter Corke. System identification, estimation and control for a cost effective open-source quadcopter. In *2012 IEEE International Conference on Robotics and Automation*, pages 2202–2209, May 2012.

[238] Jesus L. Lobo, Javier Del Ser, Albert Bifet, and Nikola K. Kasabov. Spiking neural networks and online learning: An overview and perspectives. *CoRR*, abs/1908.08019, 2019.

[239] Romain Brette. Philosophy of the spike: Rate-based vs. spike-based theories of the brain. *Frontiers in Systems Neuroscience*, 9:151, 2015.

[240] F. Akopyan, J. Sawada, A. Cassidy, R. Alvarez-Icaza, J. Arthur, P. Merolla, N. Imam, Y. Nakamura, P. Datta, G. Nam, B. Taba, M. Beakes, B. Brezzo, J. B. Kuang, R. Manohar, W. P. Risk, B. Jackson, and D. S. Modha. Truenorth: Design and tool flow of a 65 mw 1 million neuron programmable neurosynaptic chip. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 34(10):1537–1557, Oct 2015.

[241] D. Roggen, S. Hofmann, Y. Thoma, and D. Floreano. Hardware spiking neural network with run-time reconfigurable connectivity in an autonomous robot. In *NASA/DoD Conference on Evolvable Hardware, 2003. Proceedings.*, pages 189–198, July 2003.

[242] H. Soleimani, A. Ahmadi, and M. Bavandpour. Biologically inspired spiking neurons: Piecewise linear models and digital implementation. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 59(12):2991–3004, Dec 2012.

[243] Hesham Mostafa, Bruno U. Pedroni, Sadique Sheik, and Gert Cauwenberghs. Fast classification using sparsely active spiking networks. In *2017 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–4, May 2017.