Biologically Inspired Adaptive Control of Quadcopter Flight

by

Brent Komer

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2015

Abstract

This thesis explores the application of a biologically inspired adaptive controller to quadcopter flight control. This begins with an introduction to modelling the dynamics of a quadcopter, followed by an overview of control theory and neural simulation in Nengo. The Virtual Robotics Experimentation Platform (V-REP) is used to simulate the quadcopter in a physical environment. Iterative design improvements leading to the final controller are discussed. The controller model is run on a series of benchmark tasks and its performance is compared to conventional controllers. The results show that the neural adaptive controller performs on par with conventional controllers on simple tasks but exceeds far beyond these controllers on tasks involving unexpected external forces in the environment.

Acknowledgements

First off, I would like to thank my supervisor, Chris Eliasmith, for his guidance and support throughout my research. My thanks extends to the rest of the members of the Centre for Theoretical Neuroscience, for all of their help and for making this lab a great environment to work in. I would also like to thank Jeff Orchard and Edith Law for taking the time to read through this thesis. Finally, I'd like to thank my friends and family. In particular Sally Siu for all of her support and encouragement.

Dedication

To my family, for their unending support and love.

Table of Contents

List of Tables

List of Figures

Glossary

Connection A link between the output of one Ensemble or Node to the input of another Ensemble or Node.

Decoders Weightings applied to neural activities to produce a vector.

Encoders Functions applied to a vector to produce neural activities.

Ensemble A group of neurons representing a single vector.

Nengo A Python software package that implements algorithms from the Neural Engineering Framework.

Network A group of Ensembles, Nodes, Connections, and other Networks within a Nengo model.

Neural Engineering Framework A set of methods for performing computations with simulated ensembles of neurons.

Node A component of a Nengo network that executes Python code rather than representing a value.

Synapse A filter applied to a connection between two representational components in Nengo.

Tuning Curve Response characteristics of a neuron.

Chapter 1

Introduction

1.1 Motivation

Humans have an exceptional ability to be able to adapt to their surroundings. In particular the human motor control system is able to compensate for changes in forces, torques, and inertial effects on the body. For example, when picking up an object such as a hammer, the weight of the hammer will apply external forces to the hand. This will change the dynamic properties of the hand and arm movements, yet the human motor control system is able to easily compensate for these changes and accurately control movement with the object. Even if an object has never been encountered before, the human brain is able to calculate the correct changes in timing and muscle tensions in order to skilfully manipulate the object. The predictive capabilities of the brain, along with the plasticity of neural connections in the motor area help guide these sophisticated behaviours.

This ability for quick and easy adaptation to new dynamic properties of a system would be extremely useful in robotics. Applying similar methods of control that have been developed over millions of years of evolution in the brain to a robotic control system could result in major improvements. This is especially useful now that the demands of many robotic systems are now more general purpose than they were in the past. Robots started out mainly performing simple and repetitive tasks in stable environments, such as automation in manufacturing [15]. Now they are being used increasingly in more complex situations requiring a diverse amount of control, such as search and rescue missions, performing medical procedures, and assisting the elderly [15, 20, 25, 22]. When the precise environment that the robot will operate in is not fully known, it is useful for any control system that the robot uses to be adaptable to those environments.

Another advantage the brain has when it comes to control, is that it uses very little power, about 20 Watts on average [18]. Hardware inspired by the brain is being designed to take advantage of this low power paradigm. This style of hardware, known as neuromorphic hardware, is typically massively parallel and consumes much less power than traditional hardware. The algorithms explored in this thesis focus on being adaptive and biologically inspired in order to take advantage of such hardware.

1.2 Quadcopters

Quadcopters are very versatile aerial vehicles, typically consisting of a central body with four upright rotors equally spaced around the body. This configuration allows them to be lightweight and simplifies construction. They also have the ability to take off and land vertically without external assistance, meaning they can be used in a lot more situations than aircraft that require a runway or other particular conditions to take off. They can also change directions fairly easily mid flight, and have the ability to hover at a particular location. Due to their low cost, light weight, and ease of use, they are an excellent aircraft to use for research purposes.

Despite all of these positive qualities, one of the main weaknesses of quadcopters is their short battery life. Small and simple quadcopters used by hobbyists typically last only 5 to 10 minutes before they run out of power [2]. The more expensive industrial grade quadcopters typically last up to 30 minutes before they need to be recharged, but this duration is still not long enough for some applications. For example, if a quadcopter is being used in a search and rescue mission it may need to remain flying for a long time without recharging, especially if it is operating in a remote area. Battery weight is one of the main issues hindering the maximum flight time of quadcopters. If a larger battery is used in attempts to increase the maximum flight time, the quadcopter will be heavier and therefore require more power to fly, which in turn will drain the battery faster. Two possible solutions to this problem are lighter batteries and more energy-efficient operation. The latter can be achieved by the use of neuromorphic hardware to run the flight control system. While the majority of the power consumed by a quadcopter goes towards the rotors, some is still used by the on-board control system. As the sophistication of the control system increases, the computational demands will follow, leading to less overall flight time. Computational efficiency improvements in traditional digital computation is beginning to stagnate and is expected to soon approach a limit where minimal improvement is expected. On the contrary, computational power efficiency for biological systems is 8-9 orders of magnitude better than the power efficiency for digital computation [19]. Reducing the power required for the control system can allow more of the power from the batteries to be used for the rotors.

In addition to the low power consumption of neuromorphic hardware, they also have the advantage of using various types of neural algorithms that can be extremely useful for control. In particular, the learning capabilities of brain-like algorithms can be used to develop controllers that are able to adapt to unknown environments with non-linear dynamics. This thesis explores an adaptive control algorithm for quadcopter flight constructed using simulated biologically plausible neurons.

Chapter 2 gives an overview of how quadcopters are able to fly, followed by a section detailing the mathematics underlying the dynamics of a quadcopter system. Following the mathematical characterization is a description of how the quadcopter is modelled in a simulation environment. Chapter 3 gives an overview of the control theory used in developing the control system, including standard PID control, adaptive control, and adaptive control in a simulated biological neural system. Chapter 4 covers the initial design of the controller model, as well as many of the iterations of improvements leading to the final design. Chapter 5 describes the set of experiments performed and metrics used to quantify performance of the various controllers. The final section discusses the results and outlines areas of future work.

Chapter 2

Background Information

2.1 Flight Control

The system state of a quadcopter is six dimensional: three for position (x, y, and z) and three for orientation (roll, pitch, and yaw). There are four control inputs, which are typically taken to be the rotational velocity of each of the four rotors. For a physical quadcopter, these inputs are the voltages applied to each of these rotors. A transformation in these voltages can be obtained using the physical rotor parameters to estimate the rotor velocity. For simplicity of the simulation, the velocity is used directly as the control input in this thesis.

The rotors will always generate a thrust that is perpendicular to the body of the aircraft. In addition to this thrust, torque is generated by each rotor based on their spin direction as well as their distance from the center of the body. For the quadcopter to be able to control its orientation, half of the rotors spin clockwise and the other half spin counter-clockwise. The pairs diagonally opposite each other across the body spin in the same direction, allowing the torques to balance one another out and the quadcopter to maintain a steady orientation. By varying the relative speeds of each rotor, the quadcopter is able to create a net torque in a specific direction, causing a rotation about any axis.

A quadcopter performs four common actions to move around in its environment, with a distinct pattern of rotor actuation for each one: tilt forward/backward, tilt left/right, rotate, and move up/down. A quadcopter can perform any combination of these actions, and with different magnitudes of each. The relative rotor speeds required for each are shown in Figure 2.1. This set of actions make up what is referred to as the 'task space' of the quadcopter in the remainder of this thesis.

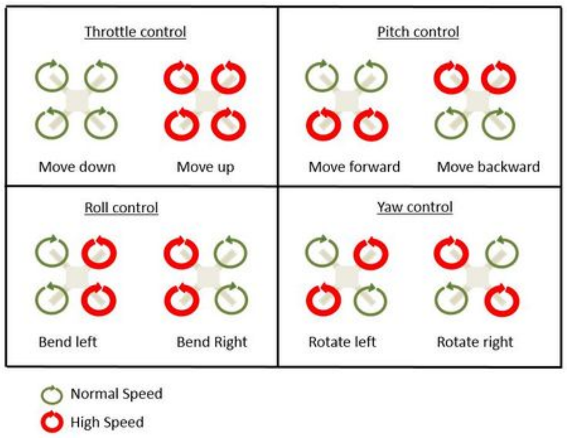


Figure 2.1: Four Primary Quadcopter Movements. Taken from [17]

2.2 Dynamics

The structure of a quadcopter is shown in Figure 2.2, along with the coordinate frame and relevant forces and torques created by the rotors. Two important frames of reference are involved when studying quadcopters: the inertial frame is used to represent the state of the quadcopter relative to the outside world, and the body frame is fixed around the center of mass of the quadcopter and used to represent local effects on the quadcopter. Typically, the inertial frame is the focus when researchers are interested in the state of the quadcopter, but the body frame is more useful when considering control of the quadcopter.

A rotation matrix (2.1) can be used to convert coordinates from the body frame to the inertial frame. Here, C_x represents $\cos x$ and S_x represents $\sin x$. To convert the other way, from inertial frame to body frame, the inverse of the matrix is used. Since this matrix is orthogonal, its inverse is equal to its transpose.

$$R = \begin{bmatrix} C_\psi C_\theta & C_\psi S_\theta S_\phi - S_\psi C_\phi & C_\psi S_\theta C_\phi + S_\psi S_\phi \\ S_\psi C_\theta & S_\psi S_\theta S_\phi + C_\psi C_\phi & S_\psi S_\theta C_\phi - C_\psi S_\phi \\ -S_\theta & C_\theta S_\phi & C_\theta C_\phi \end{bmatrix} \tag{2.1}$$

The quadcopter is assumed to be symmetrical with equal length arms for each rotor. Aligning the arms along the body's x and y axes gives the diagonal inertia matrix in (2.2). Due to symmetry, $I_{xx} = I_{yy}$.

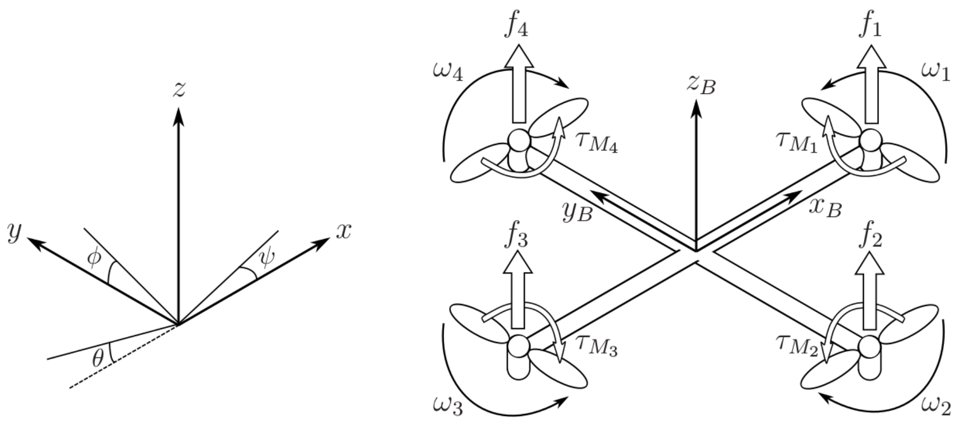


Figure 2.2: The Inertial and Body Frames of a Quadcopter. Taken from [23]

$$\begin{bmatrix} I_{xx} & 0 & 0 \\ 0 & I_{yy} & 0 \\ 0 & 0 & I_{zz} \end{bmatrix} \tag{2.2}$$

The angular velocity of each rotor generates a thrust force perpendicular to the body frame, as well as a torque about the rotor axis. This relationship is shown in (2.3) and (2.4), where ω is the angular velocity of the ith rotor, k is the lift constant, b is the drag constant, and I_M is the rotor's moment of inertia.

$$f_i = kw_i^2 \tag{2.3}$$

$$\tau_{M_i} = b\omega_i^2 + I_M\dot{\omega}_i \tag{2.4}$$

Typically the effect of the angular acceleration is considered small and is omitted in most analyses, so it will not be present in the remainder of the dynamics derivations. This is because during steady state flight the rotors will be maintaining a constant (or almost constant) velocity and will have approximately zero acceleration [16].

Note that the forces and torques generated are always proportional to the square of the rotor's angular velocity, thus working with this term instead of the angular velocity itself is simpler. The set of values making up the square of the rotor angular velocities is referred to as the 'rotor space' in the remainder of this thesis.

Combining the forces of all four rotors leads to (2.5), where T is the thrust in the direction of the z-axis of the body. Combining the torques of all four rotors leads to (2.6), where the vector τ_B represents the torques across each of the principal body axes (τ_ϕ for roll, τ_θ for pitch, and τ_ψ for yaw). The distance between the rotor and the center of mass of the quadcopter is denoted by l.

$$T = \sum_{i=1}^{4} f_i = k \sum_{i=1}^{4} w_i^2, T_B = \begin{bmatrix} 0 \\ 0 \\ T \end{bmatrix} \tag{2.5}$$

$$\tau_B = \begin{bmatrix} \tau_\phi \\ \tau_\theta \\ \tau_\psi \end{bmatrix} = \begin{bmatrix} lk(-w_2^2 + w_4^2) \\ lk(-w_1^2 + w_3^2) \\ \sum_{i=1}^{4} \tau_{M_i} \end{bmatrix} \tag{2.6}$$

The governing equation for translational motion is (2.7); where ξ is the linear position vector, T_B is the thrust from (2.5), R is the rotation matrix from (2.1), $A(\dot{\xi})$ is a matrix containing the drag force, and G is the gravitational force.

$$m\ddot{\xi} = T_B R - A(\dot{\xi})\dot{\xi} - G \tag{2.7}$$

Expanding (2.7) and setting the equation in terms of translational acceleration leads to (2.8).

$$\begin{bmatrix} \ddot{x} \\ \ddot{y} \\ \ddot{z} \end{bmatrix} = \frac{T}{m} \begin{bmatrix} C_\psi S_\theta C_\phi + S_\psi S_\phi \\ S_\psi S_\theta C_\phi - S_\psi S_\phi \\ C_\theta C_\phi \end{bmatrix} - \frac{1}{m} \begin{bmatrix} A_x|\dot{x}| & 0 & 0 \\ 0 & A_y|\dot{y}| & 0 \\ 0 & 0 & A_z|\dot{z}| \end{bmatrix} \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{z} \end{bmatrix} - \begin{bmatrix} 0 \\ 0 \\ g \end{bmatrix} \tag{2.8}$$

The governing equation for rotational motion is (2.10); where η is the Euler angle vector, I is the inertia matrix from (2.2), τ_B is the applied torque from (2.6), and $C(\dot{\eta})\dot{\eta}$ is the centripetal force.

$$I\ddot{\eta} = C(\dot{\eta})\dot{\eta} + \tau_B \tag{2.9}$$

Expanding (2.9) and setting the equation in terms of angular acceleration leads to (2.10).

$$\begin{bmatrix} \ddot{\phi} \\ \ddot{\theta} \\ \ddot{\psi} \end{bmatrix} = \begin{bmatrix} (I_{yy} - I_{zz})\dot{\theta}\dot{\psi}/I_{xx} \\ (I_{zz} - I_{xx})\dot{\phi}\dot{\psi}/I_{yy} \\ (I_{xx} - I_{yy})\dot{\phi}\dot{\theta}/I_{zz} \end{bmatrix} + \begin{bmatrix} \tau_\phi/I_{xx} \\ \tau_\theta/I_{yy} \\ \tau_\psi/I_{zz} \end{bmatrix} \tag{2.10}$$

Equations (2.8) and (2.10) are used throughout the remainder of this thesis to model the dynamics of the quadcopter during flight.

Table 2.1: Robotic Simulator Candidates

Feature	**V-REP**	**MORSE**	**Gazebo**
Python Support	****	***	*
Ease of Model Creation	***	****	**
Quality of Available Quadcopter Models	***	**	****
Timing Control	****	*	**
Interactivity with Simulation	****	**	**
Community Support	****	****	****

The number of stars indicate the extent to which the simulator demonstrates a particular feature. Descriptions for each feature are given below. *Python Support*: the extent in which the simulator supports control from an external Python script. *Ease of Model Creation*: how easy it is to build a virtual environment and add sensors and actuators to robots in this environment. *Quality of Available Quadcopter Models*: how realistic the publicly available quadcopter models are. *Timing Control*: how much control the programmer has on the physics simulation steps and synchronization with Nengo. *Interactivity with Simulation*: degree in which the simulation can be interacted with once it is built and running. *Community Support*: available online resources to get help with using the simulator.

2.3 Simulation

The quadcopter model was validated and tested using a computer simulation environment. The advantage of starting with a simulation rather than going straight to physical hardware is that it is quicker and easier to prototype new control algorithms, and it is much less costly to do so.

There are three robot simulators that were considered: Virtual Robotics Experimentation Platform (V-REP)[10], Modular Open Robotics Simulator Engine (MORSE)[11], and Gazebo[21]. All three simulators were experimented with during the beginning of this work and a qualitative summary of the relative strengths of each approach is shown in Table 2.1.

For this project the Virtual Robotics Experimentation Platform (V-REP) was chosen as the physical simulation environment. This open-source software package is free to use for educational purposes; contains models of various robotic platforms, including a model of a quadcopter; and allows the construction of intricate 3D virtual environments for the quadcopter to interact with, as well as many virtual sensors with which to equip the quadcopter, such as cameras.

The simulation environment in V-REP is set up to be a large open space with small obstacles and walls along the ground. The quadcopter model was initialized to start in the center of the space, 0.5 meters above the ground. This model is the one provided

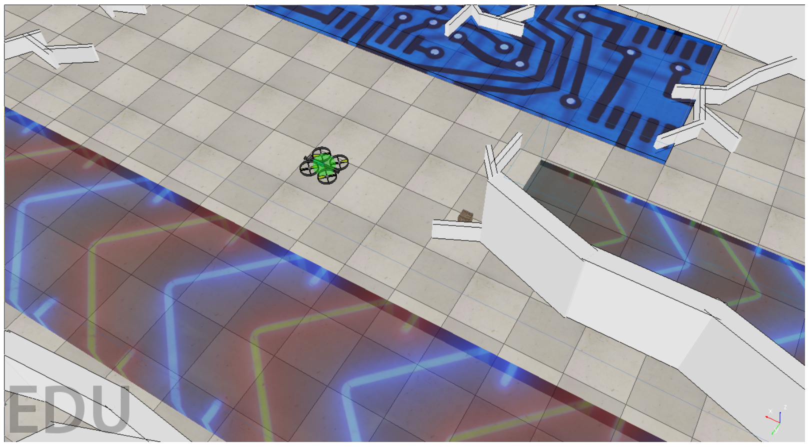


Figure 2.3: Simulation Environment in V-REP

Screen capture of an example environment in V-REP. The coloured areas with arrows represent regions of space where a wind force is applied to the quadcopter in the direction of the arrows. The blue region resembling a circuit board represents an area where an unknown non-linear force will be applied to the quadcopter. The semi-transparent green sphere is the target location. The quadcopter has reached its target in this example.

with the V-REP software courtesy of Eric Rohmer and Lyall Randell. The quadcopter's target is a small semi-transparent green sphere. The goal of the control system is to make the quadcopter move to the target's location with the target's orientation. For this thesis, an environment was created which includes various wind tunnels, which are zones of space which will exert an external force on the quadcopter in a particular direction if the quadcopter is within the zone. There are also zones that can exert various nonlinear forces on the quadcopter. For example, one zone could push the quadcopter in the z direction with a force proportional to the square of its horizontal velocity. In addition to these zones, the quadcopter can also pick up and drop off boxes of various mass. These wind zones and boxes are used to test how well the controller can adapt to changes in its dynamics. An example of a region within the simulation environment is shown in Figure 2.3.

Chapter 3

Control

3.1 PID Control

The canonical method for building a control system is to use a PID controller. This is a closed loop controller that works by applying gains to an error signal and feeding the result as input to the plant (system being controlled). PID stands for Proportional, Integral, and Derivative, referring to the three types of gains used. One gain is applied directly to the error (proportional); a second gain is applied to the derivative of the error, and the third gain is applied to the integral of the error. Each of these gains serves a unique purpose, and the careful tuning of all three is required to get the desired performance of the controller. The proportional gain is the main driver of the system and responds directly to the error to bring the system towards its target point. The larger the gain, the faster this response will be. However, using only this gain will not account for any inertia in the system and will often cause the system being controlled to overshoot its target, especially when the gain is large. In some cases, the overshoot can be so large that the system becomes unstable and oscillates out of control. Thus, the derivative gain is usually used in addition, because it is sensitive to the rate of change of the error, and attempts to bring this rate towards zero. The faster the system is moving towards its target state, the more this gain acts to slow it down, by effectively providing a form of damping and so reducing the amount of overshoot. Depending on how this parameter is tuned, and the properties of the system at hand, it can remove overshooting entirely. Due to unmodelled disturbances or changing external inputs, there may be a steady-state error in the system, wherein the controller has reached a stable state, yet there is still an error being measured. The integral term is designed to account for this steady-state error by keeping a running sum of the error over time. Eventually, this integral error should be large enough that it will drive the system to close the steady-state error gap.

3.2 Adaptive Control

Sometimes the kinematics and dynamics of the system being controlled are unknown, or change over time in an unknown fashion. A controller that works well for the system initially may not be well suited when the system undergoes changes. In this situation, it is useful to have a controller that can adapt to these changes.

The foundation of adaptive control is based on parameter estimation. First, a mathematical model of the system to be controlled is generated based on physical laws. This model is typically of the form shown in (3.1), where q is the vector of state variables, $M(q)$ is a mass/inertia matrix, $C(q, \dot{q})$ is the coriolis-drag term, $g(q)$ is the gravitational force, and τ is a vector representing the input force/torque to the system.

$$M(q)\ddot{q} + C(q, \dot{q})\dot{q} + g(q) = \tau \tag{3.1}$$

The goal of the controller is to bring the system to a particular target state. Typically the state as well as its derivative is desired to be controlled. Thus, the second derivative of the state will be zero when the system has arrived at the target state. Setting $\ddot{q}$ to be zero gives the relationship of the inputs to the rest of the state shown in (3.2).

$$\tau = C(q, \dot{q})\dot{q} + g(q) \tag{3.2}$$

An estimate of q and $\dot{q}$ can typically be measured, leaving the only potentially unknown quantities in the right-hand side of the equation to be the physical parameters of the system. If these physical parameters are constant and linear with respect to the system state, the equation can be reorganized as in (3.3). Here θ is a vector of constant system parameters and $Y(q, \dot{q})$ is a known matrix dependent on the system state. τ is the input required to keep the system in a steady state.

$$\tau = C(q, \dot{q})\dot{q} + g(q) = Y(q, \dot{q})\theta \tag{3.3}$$

Often the system parameters are not fully known and an estimate needs to be used instead. Many control applications also require the system to be able to transition to different states, rather than remain at a particular state. The adaptive control law in (3.4) uses an estimate of the parameter vector, $\hat{\theta}$, along with a standard control law to compute the desired input to the system. Here e is the state error and K is a gain matrix. More detail on this style of adaptive control law can be found in [28, 29, 9].

$$\tau = Y(q, \dot{q})\hat{\theta} + Ke \tag{3.4}$$

The parameter estimates can be initialized to any stable value and are updated according to the relationship in (3.5). L is a learning rate parameter that determines how quickly the parameter estimates change over time in proportion to the measured error. The parameter estimates will eventually converge on values that allow the system to be controlled with minimal error. Given sufficient exploration of the state space, the parameter estimates are guaranteed to converge to the real values if the real values are required for optimal control [28].

$$\dot{\hat{\theta}} = LY(q, \dot{q})^T Ke \tag{3.5}$$

Creating a mathematical model of a system with enough detail to account for everything is difficult. Assumptions and approximations must be made if the model is to be tractable. Moreover, external forces from the environment may influence the model, and their form may be unknown as the environment can be largely unknown. One way to overcome this problem is to use a set of basis functions as the model, and the weights applied to each element of the basis as the constant parameters. If the basis is designed such that it can represent any computable function to a reasonable degree of accuracy, it will be effective in the adaptive control problem. Gaussian basis functions are commonly used in adaptive control [26], but neural networks may be used as well [4]. This thesis explores the application of this form of control law using basis functions that are biologically plausible spiking neurons.

3.3 Neural Simulation

The Neural Engineering Framework (NEF)[14] provides a means of representing arbitrary vectors using the properties of neurons as a basis. This is done through a nonlinear encoding mechanism carried out by the tuning curves of the neurons, and a weighted linear decoding of the responses of the neurons to retrieve an approximation of the vector being encoded. A transformation can be applied to the underlying representation by specifying different weights on the linear decoding. Any computable function can be approximated through a transform, and the degree of accuracy of the decoding is dependent on the number of neurons used and the complexity of the function. The neurons themselves are a part of a dynamical system where timing effects and filters across connections play a role in the behaviour of the system. For more detail on the NEF, see [12, 30, 13].

Simulation of biological neurons is carried out by the software package Nengo [5]. This software implements the algorithms in the NEF and provides an easy to use Python interface for building complex models under this framework. The core components of Nengo are networks, nodes, ensembles, and connections. A network is a container for all of the components, it can contain any number of nodes, ensembles, connections, and even other networks. There is always one base network from which the rest of the model is built. Ensembles are groups of neurons representing a single vector. The dimensionality of this vector can be any positive integer. Nodes are used when a particular part of the network is doing a computation without using neurons as the underlying representation. Typically nodes are used as the inputs and outputs to a neural system. Connections specify transformations between representational components (ensembles and nodes) through one-way links where the information flows from the output of the first representational component (origin) to the input of the second representational component (termination). Connections may have a synapse model applied to them, where the information from one end of the connection is delayed by a time-step before reaching the other end and a filter with a particular time constant may be applied. If no synaptic filter is applied, the value from the origin of the connection is sent directly to the termination of the connection during the same time step.

Nengo supports a variety of underlying neuron models for its Ensembles. The most commonly used is the Leaky Integrate-and-Fire (LIF) neuron [8]. Ensembles can also be run in direct mode, in which functions are computed explicitly rather than with neurons. However, models in this mode are not implementable directly on neuromorphic hardware, and the behaviour of the system can be significantly different. Nevertheless, it can be useful for constructing working prototypes in simulation before converting the entire system to an underlying neural model.

Once the network structure has been specified, Nengo can build and run a simulation of this network for either a specified duration of time, or until a stop signal is generated. All timing is measured as 'simulated time' with a specific time-step. If the time-step is short, the simulation can capture minute timing details more accurately, but the system as a whole will run slower with respect to real-time. A default time step of 1ms is typically used in Nengo models, which provides a reasonable trade-off between accuracy and run-time.

3.4 Adaptive Control in Nengo

In this thesis, we use the adaptive control methods described above to build a quadcopter controller using Nengo. An ensemble of simulated neurons is used as the set of basis functions for the physical model, and the decoders of these neurons are used as the vector of unknown constant parameters. A biologically plausible learning rule known as the Prescribed Error Sensitivity (PES) rule is used to update the decoder values [6].

This learning method works by first creating a connection from an ensemble of spiking neurons representing the state of the system to an ensemble or node representing the output of the controller. This is known as the 'learned connection' and can be initially set to perform any transformation, but is typically initialized for the output to be random or zero. If the designer has an approximation of what the final learned transformation should look like, they can set this as the initial transformation. Doing so will allow the system to converge to the final transformation more quickly.

The learned connection will be modulated by an error signal, which can come from anywhere in the network. The PES learning rule will attempt to reduce the error signal by changing the value of the decoders on the learned connection. The direction in which the decoder values change is dependent on the sign of the error. The magnitude of the change in decoder values at each time step is dependent on both the magnitude of the error and a learning-rate parameter. The learning rate is a dimensionless parameter that needs to be tuned for the specific application. It is dependent on the simulation time step, the number of neurons in the state ensemble, as well as how responsive the model needs to be to changes. A larger learning rate will cause larger reactions to error, effectively making the system trust its current measurements more than historical ones. A smaller time-step means that these changes will occur more frequently, so the net change over time will be greater. A larger number of neurons means that the changes will be greater, as there will be more decoders changing. The overall transformation is a sum of these decoders.

Chapter 4

Design

4.1 Derivation of Non-Neural Adaptive Controller

Based on the dynamics equations in section 2.2 and the adaptive control theory presented in section 3.2, an adaptive controller for a quadcopter can be generated using the methods described in [9]. The first step is to set up the governing equation for the system (4.1), and rearrange it so that the right side is in terms of the second derivative of the state (4.2).

$$M(q)\ddot{q} + C(q,\dot{q})\dot{q} + g(q) = B(q)u \tag{4.1}$$

$$\ddot{q} = M(q)^{-1}(-C(q,\dot{q})\dot{q} - g(q) + B(q)u) \tag{4.2}$$

We can then choose the relationship in (4.3) which allows $\ddot{q}$ to be zero, given the values of the terms in (4.4) and (4.5).

$$B(q)u = C(q,\dot{q})\dot{q} + g(q) \tag{4.3}$$

$$C(q,\dot{q}) = \begin{bmatrix} A_x|\dot{x}| & 0 & 0 & 0 & 0 & 0 \\ 0 & A_y|\dot{y}| & 0 & 0 & 0 & 0 \\ 0 & 0 & A_z|\dot{z}| & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & I_{zz}\dot{\psi} & -I_{yy}\dot{\theta} \\ 0 & 0 & 0 & -I_{zz}\dot{\psi} & 0 & I_{xx}\dot{\phi} \\ 0 & 0 & 0 & I_{yy}\dot{\theta} & -I_{xx}\dot{\phi} & 0 \end{bmatrix} \tag{4.4}$$

$$g(q) = \begin{bmatrix} 0 \\ 0 \\ mg \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{4.5}$$

Here, u is a vector containing the thrust force and the rotational torque in the body frame, $B(q)$ is the transformation matrix of the force and torque from the body frame to the inertial frame, and ω is a vector containing the squared angular velocities of each rotor. Consequently, we can write:

$$u = \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \end{bmatrix} = A\omega = \begin{bmatrix} k & k & k & k \\ 0 & -lk & 0 & lk \\ -lk & 0 & lk & 0 \\ -b & b & -b & b \end{bmatrix} \begin{bmatrix} \omega_1^2 \\ \omega_2^2 \\ \omega_3^2 \\ \omega_4^2 \end{bmatrix} \tag{4.6}$$

$$B(q) = \begin{bmatrix} \cos\phi\sin\theta\cos\psi + \sin\theta\sin\psi & 0 & 0 & 0 \\ \cos\phi\sin\theta\sin\psi - \sin\theta\cos\psi & 0 & 0 & 0 \\ \cos\phi\cos\theta & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.7}$$

Expanding (4.3) and rearranging the equation in terms of ω results in (4.8).

$$\omega = Y(q, \dot{q})\hat{\theta} = A^{-1}B(q)^{-1}[C(q, \dot{q})\dot{q} + g(q)] \tag{4.8}$$

The form of $Y(q, \dot{q})$ can be determined as in (4.9) by pulling out a vector of the unknown constant parameters. The true value of these parameters assuming the model is perfect is shown in (4.11).

$$Y(q,\dot{q})\hat{\theta} = \begin{bmatrix} a|\dot{x}|\dot{x} & b|\dot{y}|\dot{y} & c|\dot{z}|\dot{z} & c & 0 & -\dot{\phi}\dot{\psi} & -\dot{\phi}\dot{\theta} \\ a|\dot{x}|\dot{x} & b|\dot{y}|\dot{y} & c|\dot{z}|\dot{z} & c & -\dot{\theta}\dot{\psi} & 0 & \dot{\phi}\dot{\theta} \\ a|\dot{x}|\dot{x} & b|\dot{y}|\dot{y} & c|\dot{z}|\dot{z} & c & 0 & \dot{\phi}\dot{\psi} & -\dot{\phi}\dot{\theta} \\ a|\dot{x}|\dot{x} & b|\dot{y}|\dot{y} & c|\dot{z}|\dot{z} & c & \dot{\theta}\dot{\psi} & 0 & \dot{\phi}\dot{\theta} \end{bmatrix} \begin{bmatrix} \hat{\theta}_1 \\ \hat{\theta}_2 \\ \hat{\theta}_3 \\ \hat{\theta}_4 \\ \hat{\theta}_5 \\ \hat{\theta}_6 \\ \hat{\theta}_7 \end{bmatrix} \tag{4.9}$$

$$d = (\cos\phi\sin\theta\cos\psi + \sin\theta\sin\psi)^2 + (\cos\phi\sin\theta\sin\psi - \sin\theta\cos\psi)^2 + (\cos\phi\cos\theta)^2 \tag{4.10a}$$

$$a = (\cos\phi\sin\theta\cos\psi + \sin\theta\sin\psi)/d \tag{4.10b}$$

$$b = (\cos\phi\sin\theta\sin\psi - \sin\theta\cos\psi)/d \tag{4.10c}$$

$$c = (\cos\phi\cos\theta)/d \tag{4.10d}$$

$$\theta = \begin{bmatrix} A_x/4k \\ A_y/4k \\ A_z/4k \\ mg/4k \\ (I_{zz} - I_{yy})/2kl \\ (I_{xx} - I_{zz})/2kl \\ (I_{yy} - I_{xx})/2kl \end{bmatrix} \tag{4.11}$$

The adaptive controller and parameter update equations are shown in (4.12) and (4.13) respectively. K is the control gain matrix (4.14), T_R is the transformation from task space to rotor space (4.15), e is the state error, and L is the learning rate. The value of L can either be a constant or a 4x4 matrix. For simplicity a constant value of one is used.

$$\omega = Y(q,\dot{q})\hat{\theta} + T_R K e \tag{4.12}$$

$$\dot{\hat{\theta}} = L Y^T T_R K e \tag{4.13}$$

4.2 Controller Implementation

The goal of the controller is to actuate the quadcopter in such a way that it travels to a desired position in a reasonable amount of time. This position is specified as a set of x, y, and z coordinates and a particular yaw direction. The controller is given the state error of the quadcopter and its target, as well as the current linear and angular velocities of the quadcopter. It needs to use this information to generate a suitable control signal.

The adaptive controller here consists of two parts. The first is a standard PD controller that generates a four-dimensional output signal in the space of possible quadcopter motions (task space). The gain matrix that performs this operation can be seen in (4.14). This signal is then multiplied by the matrix in (4.15) to transform it into the four dimensional rotor velocity space, providing the actuation associated with the desired movement commands. The design of this matrix depends upon the orientation of the rotor blades to the x and y axes. This transformation matrix assumes that the rotor axes are offset from the x and y axes by 45 degrees as the V-REP model used here.

$$K = \begin{bmatrix} 0 & 0 & k_2 & 0 & 0 & -k_4 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & k_1 & 0 & 0 & -k_3 & 0 & -k_5 & 0 & 0 & k_7 & 0 & 0 \\ -k_1 & 0 & 0 & k_3 & 0 & 0 & 0 & -k_5 & 0 & 0 & k_7 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -k_6 & 0 & 0 & k_8 \end{bmatrix} \tag{4.14}$$

$$T_R = \begin{bmatrix} 1 & -1 & 1 & 1 \\ 1 & -1 & -1 & -1 \\ 1 & 1 & -1 & 1 \\ 1 & 1 & 1 & -1 \end{bmatrix} \tag{4.15}$$

$$u = \begin{bmatrix} w_1^2 \\ w_2^2 \\ w_3^2 \\ w_4^2 \end{bmatrix} = T_R K \begin{bmatrix} x \\ y \\ z \\ \dot{x} \\ \dot{y} \\ \dot{z} \\ \phi \\ \theta \\ \psi \\ \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} \tag{4.16}$$

Translation in the x and y directions is dependent on the states of both x and y as well as roll and pitch, because a roll in the quadcopter causes a component of thrust to be applied in the y direction, and a pitch causes a component of thrust to be applied in the x direction. A delicate balance needs to be found between each of the gains in order to create a stable, functioning controller. The setpoint for each of the velocities as well as for roll and pitch is zero.

The state error is measured relative to the body frame because most sensors on a real quadcopter would return measurements relative to the sensor device itself, which is located on the quadcopter. Absolute measurements could still be obtained with a GPS device, but would typically be much less accurate and such devices are harder to use for fine-tuned control. Using localized sensors, the state of the quadcopter can be defined relative to its target, making the state in the same coordinates as the error.

Here the controller is implemented with a Nengo network, in which a 12-dimensional ensemble representing the state error can be projected to a 4-dimensional ensemble representing the desired control command. The transformation done through this projection will be by the 12x4 PD gain matrix of the controller (4.14). This 4-dimensional ensemble is then projected to another 4-dimensional ensemble which represents the four desired angular velocities of the quadcopter's rotors. This projection is done through a transformation by the 4x4 rotor matrix (4.15). This rotor velocity ensemble is connected to a node representing the physical quadcopter, which in turn feeds back into the state error ensemble. The network diagram is shown in Figure 4.1. The network can be simplified further by multiplying the gain matrix with the rotor matrix to give a single transformation matrix from state error to rotor velocity. This simplified and functionally equivalent network is shown in Figure 4.2. The larger network is used in the remainder of this thesis because it is more explicit regarding how each signal is used, and allows greater flexibility for design improvements and system debugging.

4.3 Iterations

The first version of the controller uses an adaptive ensemble to influence the task space command. A projection from the adaptive ensemble to the task ensemble is modulated by the state error undergoing the control transform. The PES learning rule is applied to this connection, which seeks to build a transformation that minimizes the error coming in from this modulatory connection. The effect is similar to that of the I term in a PID controller, except that it uses all state information to come up with the integral gain, and can perform nonlinear transforms to accomplish this. Without this adaptive component,

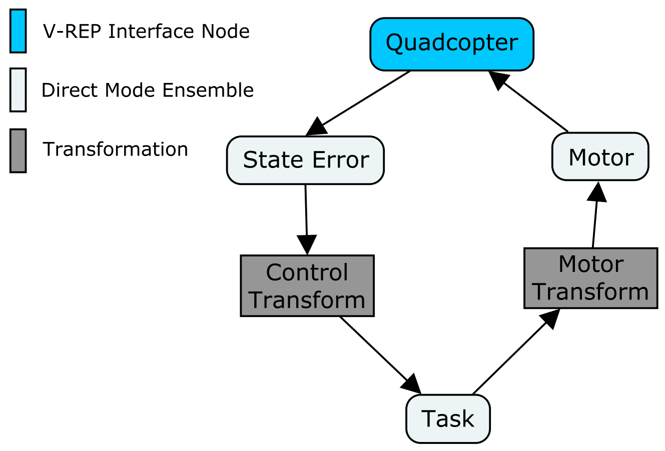


Figure 4.1: Basic Quadcopter Controller Network

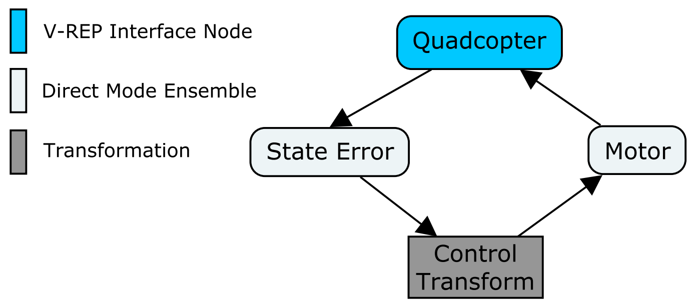


Figure 4.2: Simplified Quadcopter Controller Network

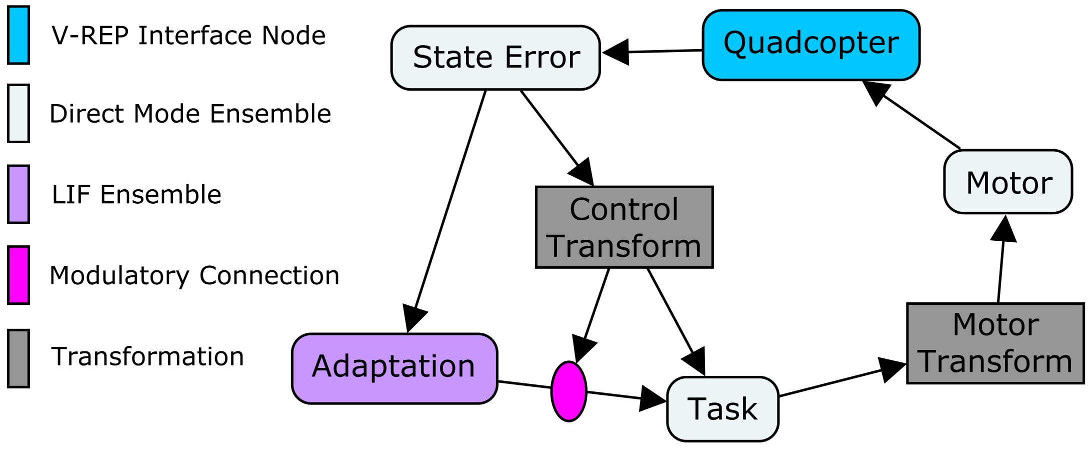


Figure 4.3: Quadcopter Controller Network with Adaptation

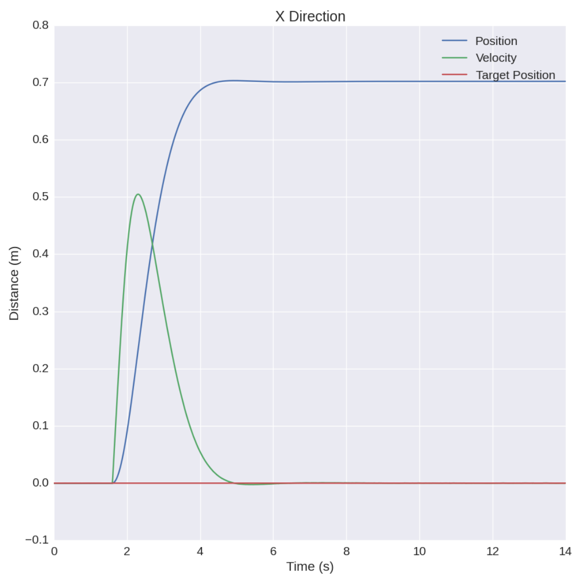
the quadcopter will have a large steady state error in the z direction, and will fall to the ground as soon as the simulation starts.

The adaptive ensemble is the only component of the network that takes advantage of a neural representation, so that ensemble is constructed using spiking LIF neurons. The other ensembles are run in direct mode and store their exact mathematical values. The network diagram is shown in Figure 4.3.

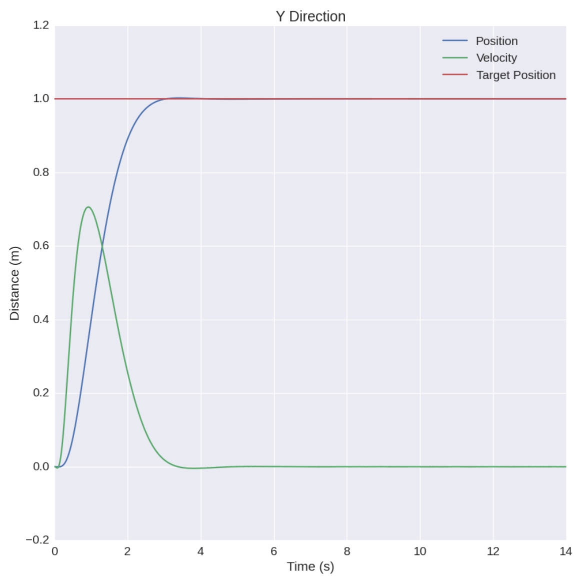
This model is very effective at learning the unknown mass of the quadcopter and adapting to any external forces in the z direction. Gains were initially chosen manually by surveying other quadcopter models and trying gains until reasonable results were found. The starting gains were based on those present in the quadcopter PD controller provided with V-REP [10].

When an external force is applied in the horizontal plane, this controller does a very bad job of correcting for it. The quadcopter will settle to a stable point in space with a constant offset from the target location, as can be seen in Figure 4.4a. This behaviour results from how the error is specified: since both position and angle gains are being combined into the same dimension to produce the task space error, multiple pairs of measurements will produce the same error. That is, the adaptive ensemble is being given an effective error reading of zero even though the quadcopter is not at the desired position. For example, if wind is producing a force in the x direction, in order for the quadcopter to remain stationary, it must have a pitch angle that allows its thrust to compensate for both the gravitational force and the translational force.

Since the quadcopter has a non-zero pitch, the control signal produced by this pitch



(a) Displacement by Wind



(b) Trajectory to Target

Figure 4.4: Performance of Original Adaptive Controller

multiplied by its gain will be non-zero. In order to have zero error, the product of the x position error and its gain must match this value. The quadcopter will consequently move away from the target x position in order to maintain zero control error, an undesirable side-effect of this controller design. In fact, there is a whole space of angle-position measurement pairs that produce zero error. A visualization of this space is depicted in Figure 4.5. The reason this controller works so well under normal operation despite this flaw is that only one point in that space is ever stable at any particular time. When the only force acting on the quadcopter is gravity, that point is at the target location with roll and pitch angles of zero radians. As external forces are applied along the horizontal plane, this stable point shifts to new locations.

One way to overcome the above problem is to modify the error signal that the adaptive ensemble uses. The only important state variables that need to match the target are the x, y, z position and yaw angle. The state velocities should all be zero as well. Since roll and pitch are not supposed to be controlled, they should really be left out of the error that the adaptive ensemble uses since they do not correspond to an actual error in the desired state. This information is still important in allowing the quadcopter to fly properly, as the controller needs to know roll and pitch information for stable flight. If this information is removed from the controller entirely, it cannot fly.

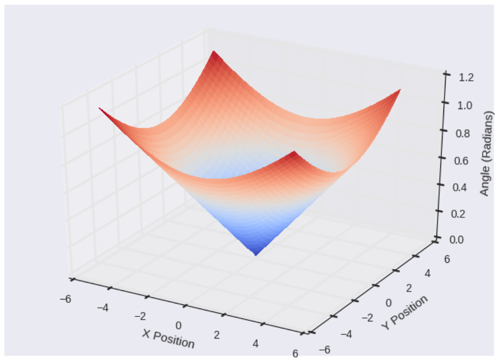


Figure 4.5: Zero Control Signal Region

This surface indicates the region of the quadcopter state where the control output is zero. The target position is at [0,0] and the angle shown is a combination of the pitch and roll angles where $angle = \sqrt{\phi^2 + \theta^2}$. The quadcopter is only ever stable at one point on this surface. The location of that point is dependent on the external forces acting on the quadcopter. Under only the influence of gravity, the stable point is at the center of [0,0], but when an external force is applied along the horizontal plane, the stable point shifts in the direction of that force by an amount proportional to the magnitude of that force. The slope of this surface is the ratio of the linear and angular P gains of the controller, k_1/k_5.

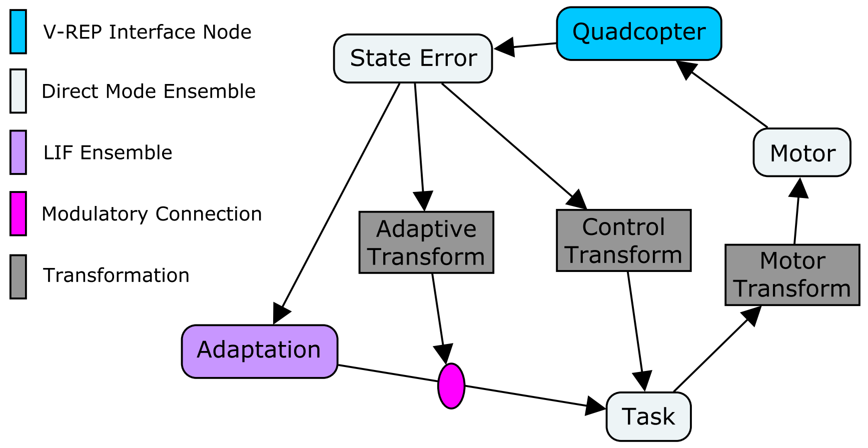


Figure 4.6: Quadcopter Controller Network with Adaptive Transformation

Consequently, the first iteration of the original controller omits the roll and pitch information only from the adaptive ensemble, while still using it for the baseline controller. This approach involves generating a separate gain matrix to use on the modulatory connection to the learned transformation. The new network diagram of the controller is shown below in Figure 4.6. The form of the gain matrix for the modulatory connection is shown in (4.17).

$$K_a = \begin{bmatrix} 0 & 0 & k_{a2} & 0 & 0 & -k_{a4} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & k_{a1} & 0 & 0 & -k_{a3} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -k_{a1} & 0 & 0 & k_{a3} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -k_6 & 0 & 0 & k_8 \end{bmatrix} \tag{4.17}$$

4.3.1 Gain Tuning

This controller is now able to adapt to external horizontal forces, but normal flight is much less stable and prone to overshooting targets. Possibly because the gains need to be retuned to work with this new controller setup. However, as there are 12 different gains in this controller, tuning them manually is a difficult and laborious task, one well-suited for automated parameter optimization, as described next.

The tool that I chose to use for this optimization is Hyperopt [7]. Hyperopt is a python package designed to perform parameter optimization over a search space for a given function. As long as a problem can be set as a function that takes any number of parameters and returns a single error metric to be minimized, that problem can be used with Hyperopt. Hyperopt uses a technique known as Sequential Model-Based Optimization (SMBO) for function optimization. SMBO methods are typically used when the goal is to optimize a function that is costly to evaluate as they invest more time between function evaluations than other methods, such as conjugate gradient descent, in order to reduce the overall number of evaluations [24, 7]. Since each set of gains to evaluate requires a controller model to be generated and physics simulation to be run for a fixed amount of time (on the order of seconds), the evaluations are costly to compute time-wise, leading to SMBO as the preferred choice of optimization method.

4.3.2 Using Hyperopt

In order to use Hyperopt, the controller model must be encapsulated in a function that returns a useful metric of how well the controller works. This was done by creating a set of target points for the quadcopter to move through over a period of time and an additional ensemble computing a scalar measure of the error of the quadcopter from the target. This error metric was taken to be a weighted Euclidean norm of each of the dimensions of the state. States that were deemed more important (such as the x, y, and z position) were given higher weights in this calculation. States that were less important (such as roll and pitch) were given lower weights. Paired with this error was a status signal of whether the quadcopter should be at the target at a given time or en-route. As the quadcopter cannot be expected to move instantaneously between targets, it should not be penalized for having an error when it has just been told to move to a different location. This status signal is set to 0 right after a new target is introduced, and then set to 1 again about the time when the quadcopter is expected to have reached the new target. The error at each time step is multiplied by this status signal before being recorded. The delay in switching back on the status signal can be used to indicate how important it is for the quadcopter to reach the target quickly. A longer delay means the optimization will find parameters that allow the quadcopter to reach the target very precisely with little oscillation, but reaching the target could take a long time. A shorter delay will prefer controllers that get to the target quickly, but may overshoot, have a steady state error, or jitter once they reach the target.

Model creation, running the physics simulator, sending the target commands, stopping the physics simulator, and returning the error metric were all encapsulated in a single Python function. This function was then given to Hyperopt, which ran it many times and kept track of the parameters used for the best result. There were 3000 evaluations of different parameter sets, and each run took about 30 seconds to complete. Hyperopt is able to go through a set number of evaluation points in one run, and then pick up where

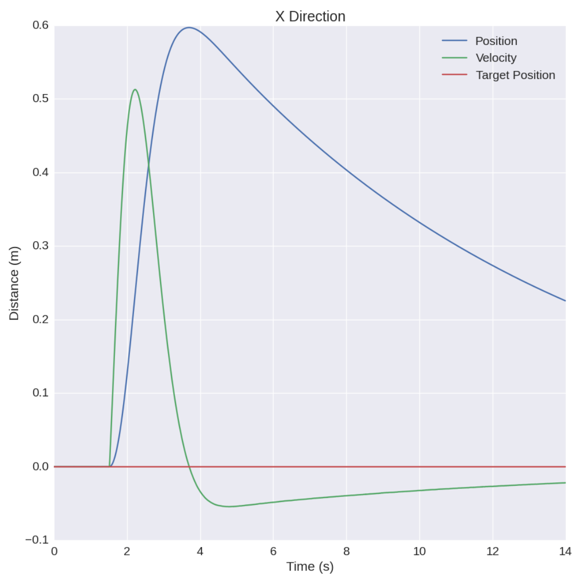
Table 4.1: Gain Values Obtained Through Hyperopt

Gain	**Original**	**Hyperopt**	**Hybrid**
k_1	0.22	0.4335	0.4335
k_2	2.00	3.8617	8.0000
k_3	0.47	0.5388	0.5388
k_4	1.65	4.6702	6.6000
k_5	3.80	2.5995	2.5995
k_6	10.0	0.8029	6.4230
k_7	1.95	0.5990	0.5990
k_8	3.16	2.8897	11.5589
k_{a1}	0.0063	0.0262	0.0262
k_{a2}	10.0	48.2387	26.00
k_{a3}	0.0150	0.0276	0.0276
k_{a4}	8.25	34.9626	21.45

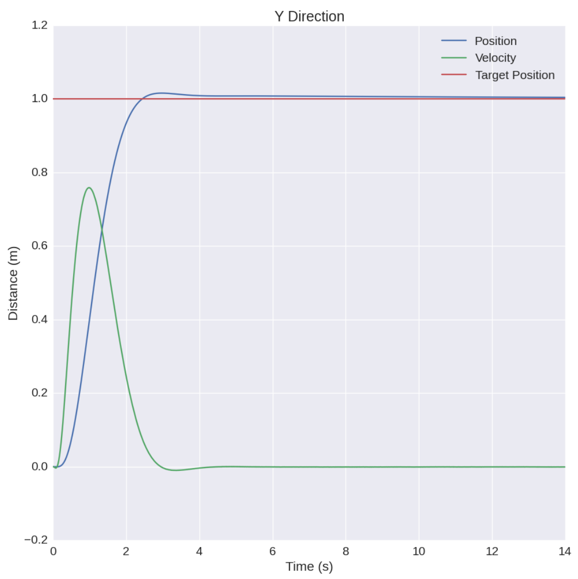
it left off in a separate run. This capability allowed these runs to be completed overnight over the course of multiple nights to get the final results, shown in the Hyperopt column of Table 4.1. These are by no means the optimal gain parameters, as the function being optimized does not represent the full operational space of the controller, and the method for generating the error metric was very simplistic. The more complex the set of targets in the objective function, the longer each run will take to complete, meaning less of the parameter space can be explored in the same amount of time. A tradeoff had to be made between the amount of data to evaluate and the quality of each data point. Nevertheless, these gains turned out to produce a controller that works exceptionally well and is faster, more responsive, and more accurate than the previous controller. The performance is further improved by augmenting the gains found through Hyperopt with hand-tuned gains. These tweaks to the gains were done empirically to produce stronger performance in particular areas of flight. These final gains are shown in the last column of Table 4.1.

4.3.3 Improving Adaptation to Horizontal Forces

There is still much room for improvement of this controller. While it can adapt to horizontal forces, it does so quite slowly, as can be seen in Figure 4.7. This is because the adaptive component of the controller is always fighting against the command given by the standard



(a) Displacement by Wind



(b) Trajectory to Target

Figure 4.7: Performance of Basic Adaptive Controller

component. There is an implicit target roll and pitch angle of zero radians that the controller is trying to achieve even if that is not the correct angle to be at. The second iteration of the controller applied an approach to overcoming this problem that employs a second adaptive ensemble that tries to learn what roll and pitch angle will allow the quadcopter to be stationary, and then feed these angles to the basic controller. Thus, the controller will no longer be actively trying to bring the quadcopter away from its setpoint, yet still has the angle information required for it to be able to fly. A network diagram of this controller is shown in Figure 4.8 and the performance of this controller is shown in Figure 4.9. As can be seen, control in the x direction is significantly improved, with only slightly slower control in the y direction.

4.3.4 Shortcomings of the Error Signal

Even with the above adaptation, a problem remains with the implementation of this controller. Since the adaptive component is driven by state error, as soon as the target is moved to a new location, a large error will be produced. This error will cause the adaptive transformation to change, even if it was already at its optimal configuration. This change can cause problems in control, such as the overshoot and oscillations seen in Figure 4.9b.

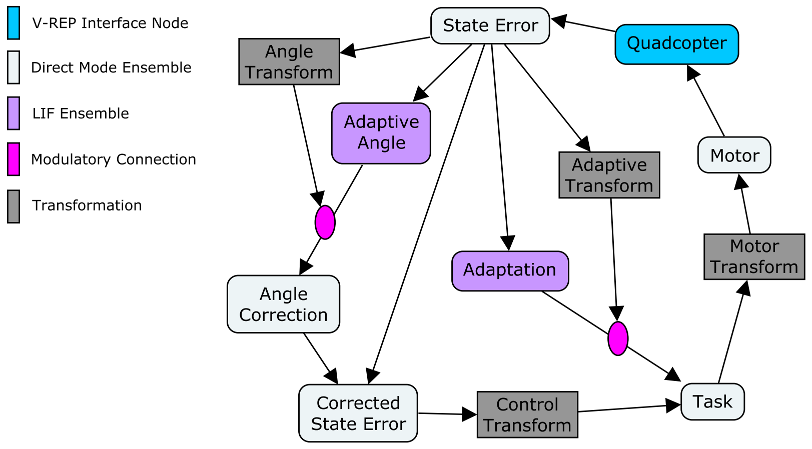
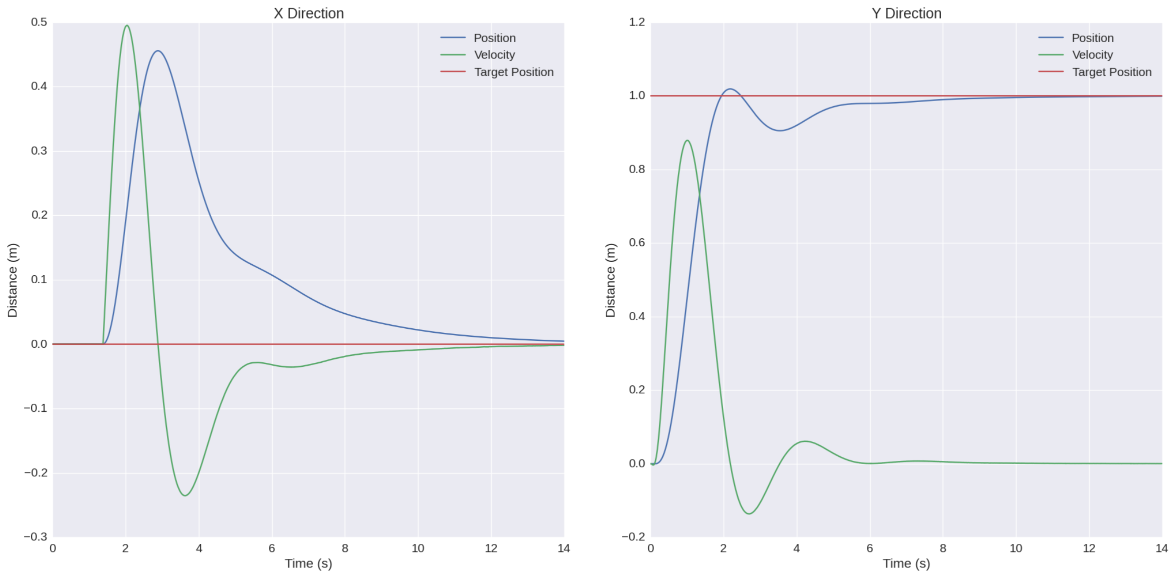


Figure 4.8: Quadcopter Controller Network with Angle Correction



(a) Displacement by Wind

(b) Trajectory to Target

Figure 4.9: Performance of Angle Correction Controller

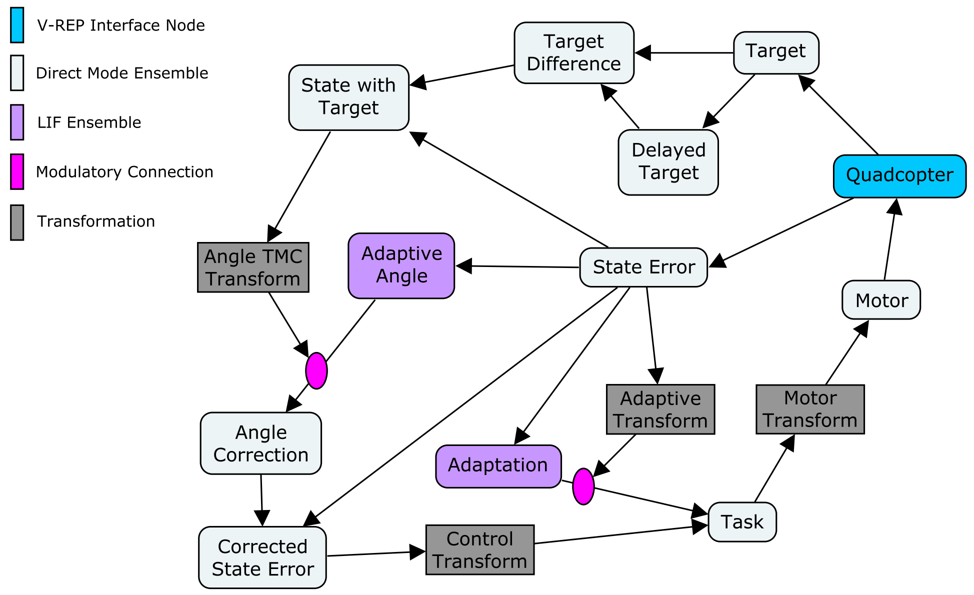
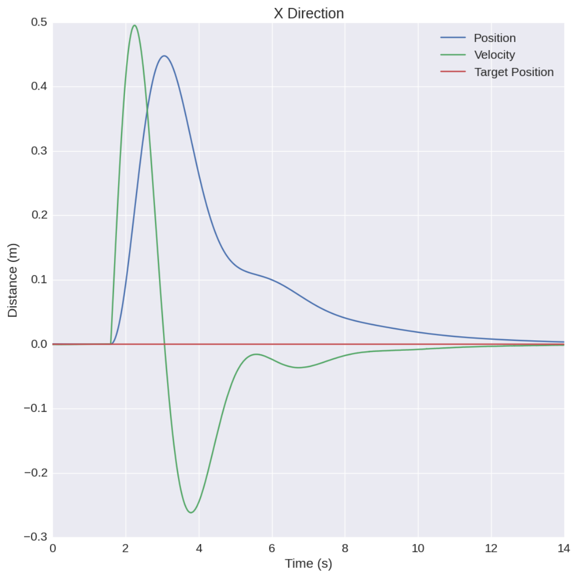


Figure 4.10: Quadcopter Controller Network with Target Modulated Control

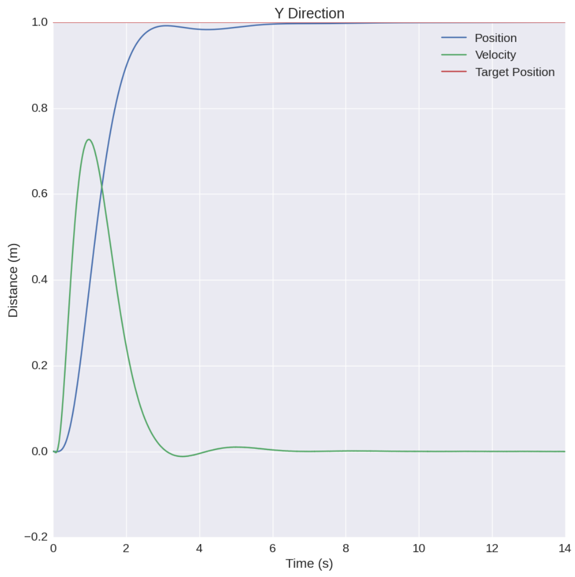
The summation of the control signal of the underlying controller and the signal from the adaptation to the error creates a control signal that is too large for the desired performance.

To correct this behaviour, the third iteration of the controller being developed here includes a time-delayed filter that is first applied to an ensemble representing the target location. This filtered location is then subtracted from an unfiltered location and stored in a new ensemble. The signal coming from this new ensemble can be used to inhibit the adaptive ensembles. If the target has not moved recently, the value being projected from this filtered-difference ensemble will be close to zero, meaning there is no inhibition. If the target is suddenly moved, this filtered-difference ensemble will begin to output the difference between the two target positions, thereby inhibiting the adaptation in any dimension that has a difference. Over time the filtered target will start to match the unfiltered target, and the extent to which the adaptive ensembles can adapt to any errors increases. The network diagram for this model can be seen in Figure 4.10 and the performance of this controller is shown in Figure 4.11. In this case the filtered-difference ensemble feeds into just the angle correction system.

This controller has the best performance of those tested both for operation under normal conditions and in the presence of unknown external forces.



(a) Displacement by Wind



(b) Trajectory to Target

Figure 4.11: Performance of Filtered Angle Correction Controller

4.3.5 Additional State Information

The state information represented by the adaptive ensemble does not have to be the same as the state information fed into the controller. This allows the adaptation to take advantage of any sensory information available. For example, a low battery could affect how the quadcopter flies or the accuracy of the sensor measurements. If the adaptive controller is aware of the battery state, it could learn to adapt its control signal to account for the kinds of dynamic changes that occur in this regime. Similarly, if there is a sensor that can detect weather information (strong winds, rain, snow, etc), this sensor can be used to provide context for the controller to learn how to best adapt to these situations. In the controllers described in the previous sections, the error with respect to the target is the only context used as this was all that was being measured. One useful piece of additional information available in the simulation is absolute position. If there are dynamic effects that vary with the position of the quadcopter, such as strong winds in a particular region, the controller will be able to learn the association of the dynamic effects with a particular location in order to adapt faster to those effects when the quadcopter enters that location. The final iteration of the controller design here has the network diagram shown in Figure 4.12. It is similar to the previous iteration, but the target modulation is applied to both adaptive

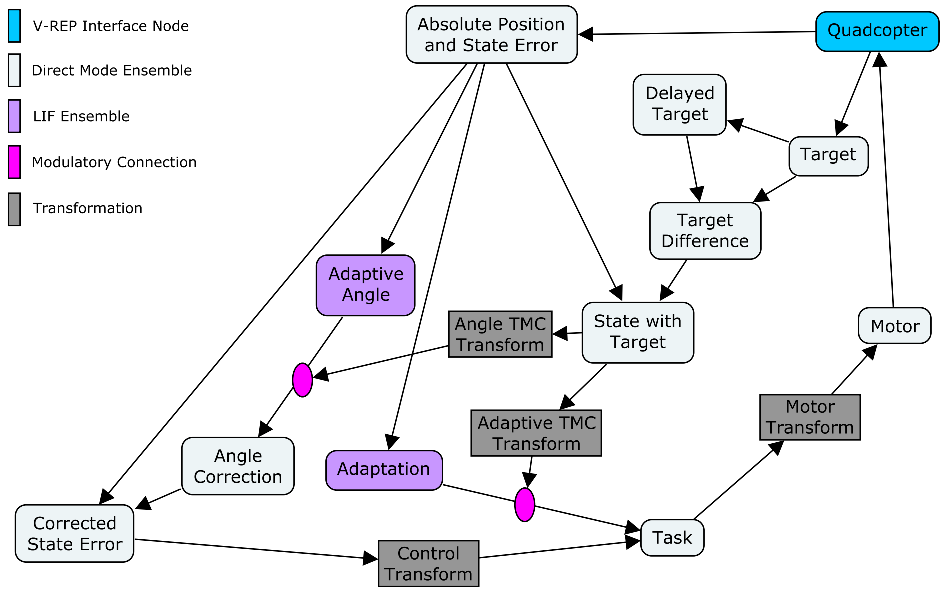


Figure 4.12: Quadcopter Controller Network with Allocentric Target Modulated Control

populations and additional allocentric information is contained in the state and adaptation ensembles. This allocentric information is not passed on through the control transform.

4.4 Software Simulation

Each of these controllers is implemented as a Nengo model in Python. The main communication channel between V-REP and Nengo is the Quadcopter Node. This node contains a callable Python class that manages the communication between the two systems at every time step and, from the Nengo network's point of view, represents the entirety of the physics simulation. Connections can be made to and from this node just like any other node in a Nengo network. A connection to the V-REP remote API server is established when this node is created. This node outputs the 6 dimensional state of the quadcopter, the 6 dimensional derivative of the state of the quadcopter, as well as the 6 dimensional state of the target. These values are obtained by making a remote API call to read these values from the current state of the simulation. The input to the Quadcopter Node is a 4 dimensional signal representing the velocity commands to give to each of the four rotors. These commands are packed into a string signal and sent to the V-REP simulation. A Lua script is run within V-REP each time step, and this script unpacks these velocity commands. It then issues them to the physical quadcopter model. The physics engine calculates the appropriate forces and torques that will be applied to the quadcopter as well as the resulting changes in state (position, velocity, orientation, and angular velocity) after one time-step. This new state will now be read by the Nengo script in its next time-step.

Nengo is run with a time-step of 1ms, as this is the standard time-step for most models and is sufficient for modelling spike timing effects in ensembles of LIF neurons while still running at a reasonable speed on most computer processors. The V-REP simulation on the other hand is run at a 10ms time-step. Ideally it would also be run at 1ms, but the simulator has trouble rendering and making calculations that quickly. To account for this, Nengo only communicates with V-REP every 10 of its time-steps. A synchronization trigger command is also issued every 10 time-steps from Nengo. The V-REP simulation can move forward only after a trigger is issued, and only by one time-step. Nengo pauses simulation until V-REP has completed this time-step. This sequence ensures that the timing of each simulation is always synchronized. Some processing overhead is incurred in ensuring the synchronization of these two systems, but the improved accuracy of the overall simulation justifies this expense.

Chapter 5

Simulations and Results

5.1 Experiments

To get a sense of how well the neural adaptive controller performs, some reference implementations were created to use as benchmarks. Five different non-neural controllers were used: a standard PD controller, a standard PID controller, an improved PID controller, an improved PID controller with a faster integral gain, and an adaptive controller. Several iterations of the neural adaptive controller are used in the benchmarking: the basic neural adaptive controller, the angle corrective controller, the target modulated controller (TMC), and the target modulated controller with allocentric information (TMCA). The controllers used in the benchmarking are listed in Table 5.1

For each non-adaptive reference implementation, a gravity compensation term had to be calculated and applied to the controllers in order to obtain reasonable performance. The magnitude of this term was determined empirically and is proportional to the mass of the quadcopter. The rotor velocity signal is the sum of the gravity compensation term and the output of the controller. The adaptive controllers do not need this extra term as they are able to learn how to compensate for the effects of gravity very quickly.

5.1.1 Metrics

Three metrics are used to evaluate performance on these tasks. The first is the Root Mean Squared (RMS) error of the difference between the quadcopter's current state and its target state. The quadcopter's state consists of position, velocity, orientation, and angular

Table 5.1: Controller Models

Controller Name	**Description**
Neural Adaptive	Simple adaptive neural controller. Uses an ensemble of LIF neurons to augment a PD control signal to adapt to an unknown environment. The network diagram can be seen in Figure 4.6.
Neural Adaptive Angle Correction	Adaptive neural controller with an additional adaptive neural ensemble for determining stable roll and pitch. The network diagram can be seen in Figure 4.8.
Neural Adaptive TMC	The difference between the current target state and a time delayed target modulates the error signal that drives the learning. The network diagram can be seen in Figure 4.10.
Neural Adaptive TMCA	The same as above, but absolute position and orientation information is projected into the adaptive ensembles along with the relative error to the target. The delayed target affects the error signal for both adaptive ensembles as well. The network diagram can be seen in Figure 4.12.
Non-Neural Adaptive	Adaptive controller using an analytical model without neurons. Derivation shown in section 4.1.
PD	Standard PD controller with gravity compensation term.
PID	Standard PID controller with gravity compensation term.
PIDt	PID controller where the error signal for the I term is in task space rather than state space. Uses the adaptive transform to calculate this error.
PIDtf	PIDt controller with a greater integral gain.

velocity. These state variables are combined by computing the length of the resulting 12-dimensional error vector to produce a single quantity. This value is calculated at each time-step for the duration of the run and then averaged by the number of time-steps in the run. For point to point control this error is sometimes not the most informative because as soon as the target point has changed a large error value will be recorded even if the quadcopter is moving optimally towards its target.

The second metric is a modified version of the RMS error designed to take the desired trajectory into account. This works by ignoring any error along the direction to the target as long as the current velocity is also in that direction. It also ignores any error caused by the velocity in the correct direction as long as the quadcopter is not currently at its target. There are also weights placed on specific types of errors to reflect an increased desire to minimize those errors. For example, errors caused by overshooting the target are weighted more heavily.

The third metric is the time taken for the quadcopter to reach its target within a particular tolerance. This metric favours controllers that can reach the target quickly. This metric does not worry about overshoot and non-optimal trajectories as long as steady state is achieved at the target in the end. The particular tolerance chosen for these experiments is maintaining an RMS error of less than 0.001 for a duration of one second.

The majority of the benchmarks performed in this thesis will report results using the trajectory RMS error and the time-to-target metric. The RMS metrics were recorded over a 30 second simulation time. The time-to-target trials were also run for a total of 30 seconds and a value of 30 is recorded if the quadcopter never reaches its target within tolerance over that duration.

5.1.2 Benchmarks for Simple Environments

A series of simple point-to-point control tasks were used to give an indication of performance: movement in the vertical direction, movement in the horizontal direction, rotation about the yaw axis, and horizontal movement into a wind tunnel. These tasks are summarized in Table 5.2 below. Each task with translational motion was completed across four different target distances (every meter from 1m to 4m) and the task with rotational motion was completed across three different target angles (every 45 degrees from 45° to 135°). Experimental results (mean and standard error) obtained from the reference controllers and the neural adaptive controllers are shown for each of these tasks in Figures 5.2 to 5.5. The legend for all of the bar plots in the remainder of this thesis is shown in Figure 5.1.

Table 5.2: Benchmark Tasks

Task	**Description**
Vertical	Fly upwards (between one and four meters)
Horizontal	Fly in the x direction (between one and four meters)
Rotation	Rotate about the yaw axis (between 45 and 135 degrees)
Wind	Fly in the y direction (between one and four meters). Enter a wind tunnel after 0.75 meters. This wind tunnel exerts a force of 0.6 N in the x direction

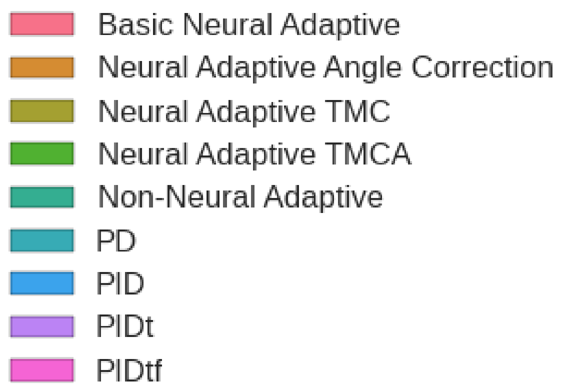


Figure 5.1: Controllers used in Benchmarking

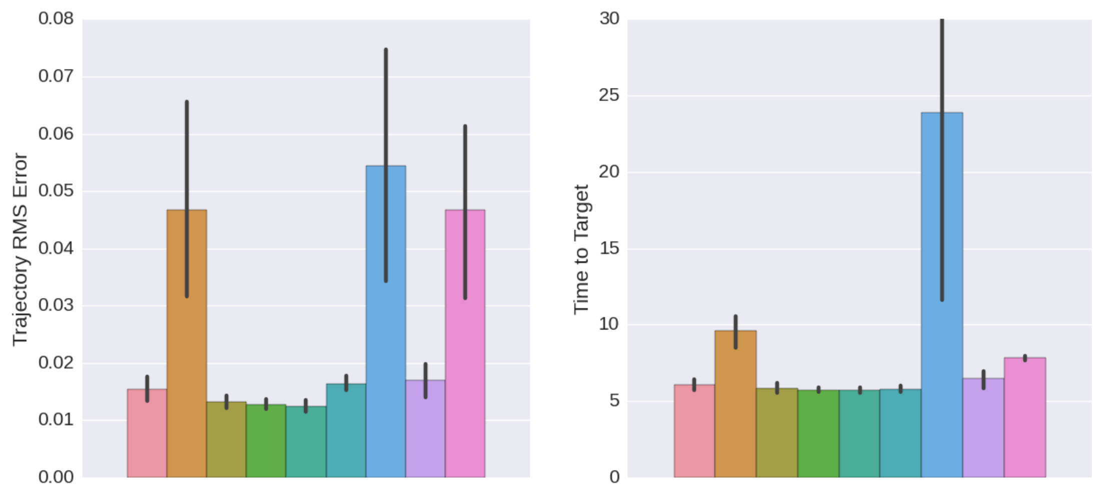


Figure 5.2: Performance on Horizontal Movement Tasks

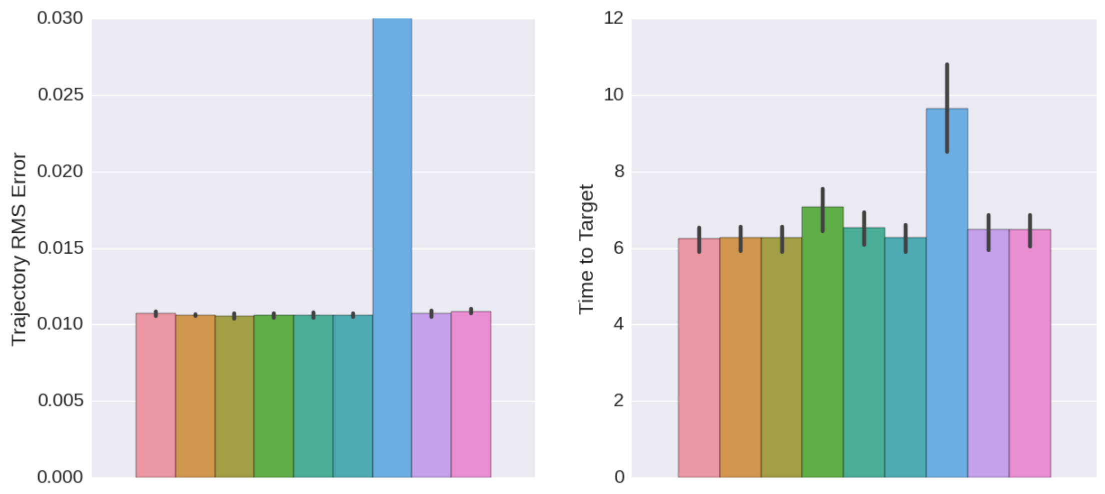


Figure 5.3: Performance on Vertical Movement Tasks

The PID controller fared quite poorly on this task because it would overshoot the target on every trial. The trajectory RMS error is truncated in the plot to allow the performance of the other controllers to be easily seen.

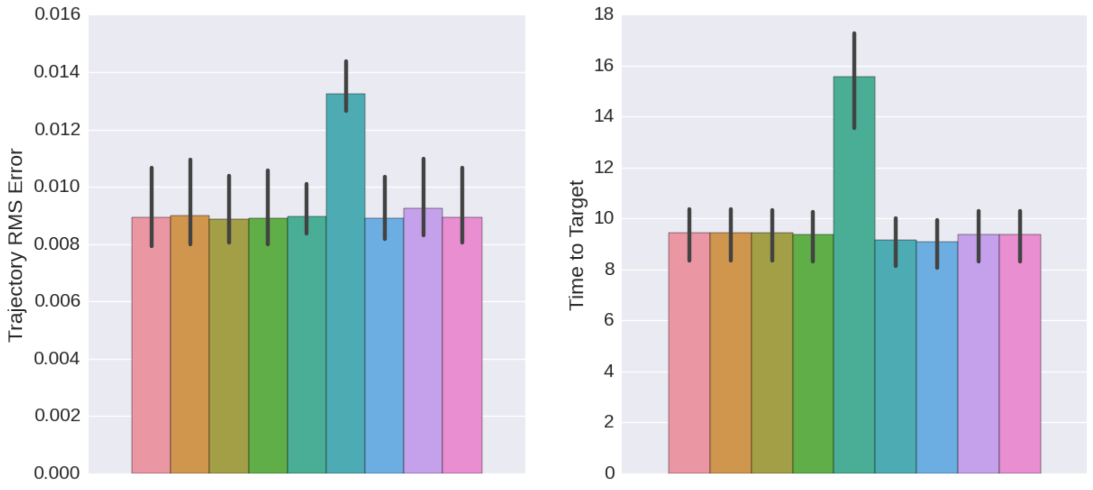


Figure 5.4: Performance on Rotational Movement Tasks

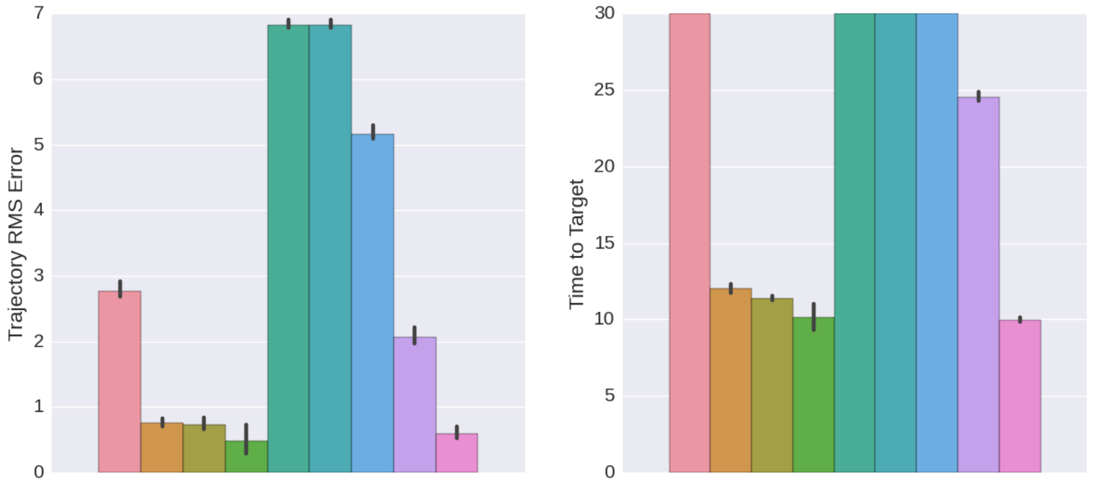


Figure 5.5: Performance on Horizontal Movement Through Wind Tasks

Four of the controllers tested were not able to reach the target within the 30 second time limit. A value of 30 seconds is reported in these cases. This was due to these controllers being unable to compensate for the steady-state error in the horizontal direction caused by the wind.

Table 5.3: Forcing Functions

Name	**Description**
None	No external forces applied
Constant	A constant horizontal force is applied in a direction perpendicular to the quadcopter's desired trajectory
Vertical Velocity	A downward force is applied proportional to the quadcopter's horizontal velocity
Horizontal Position	A force is applied in the y direction proportional to the quadcopter's x position
Horizontal Velocity	A force is applied in the y direction proportional to the quadcopter's x velocity

Overall, the performance of all of the controllers tested are quite similar on the horizontal, vertical, and rotational tasks. On the task that involves movement in the presence of wind, the iterations of the neural adaptive controller fare very well. The PID controller with a fast integral gain also does well on this task, so it is not clear from these tests if an adaptive controller has an advantage over a well tuned PID controller.

The performance of the neural adaptive controller is quite strong, but its performance is still relatively close to that of the modified PID controllers. Where the neural adaptive controller really excels is when there are external forces being applied that are functions of the system state. The neural adaptive controller is designed to be able to account for both linear and nonlinear functions of the system state.

5.1.3 Benchmarks for Complex Environments

To quantify the performance of the quadcopter controllers under the influence of these more interesting forces, a new set of benchmark tasks is used. These tasks are listed in Table 5.3 below.

Performance of the neural adaptive controllers is compared to the reference controllers on these tasks. The results are displayed in Figure 5.6. The tasks that involved external forces in the horizontal direction tended to be the most difficult for the controllers tested here, likely due to the fact that these forces are more complicated to correct for. In these

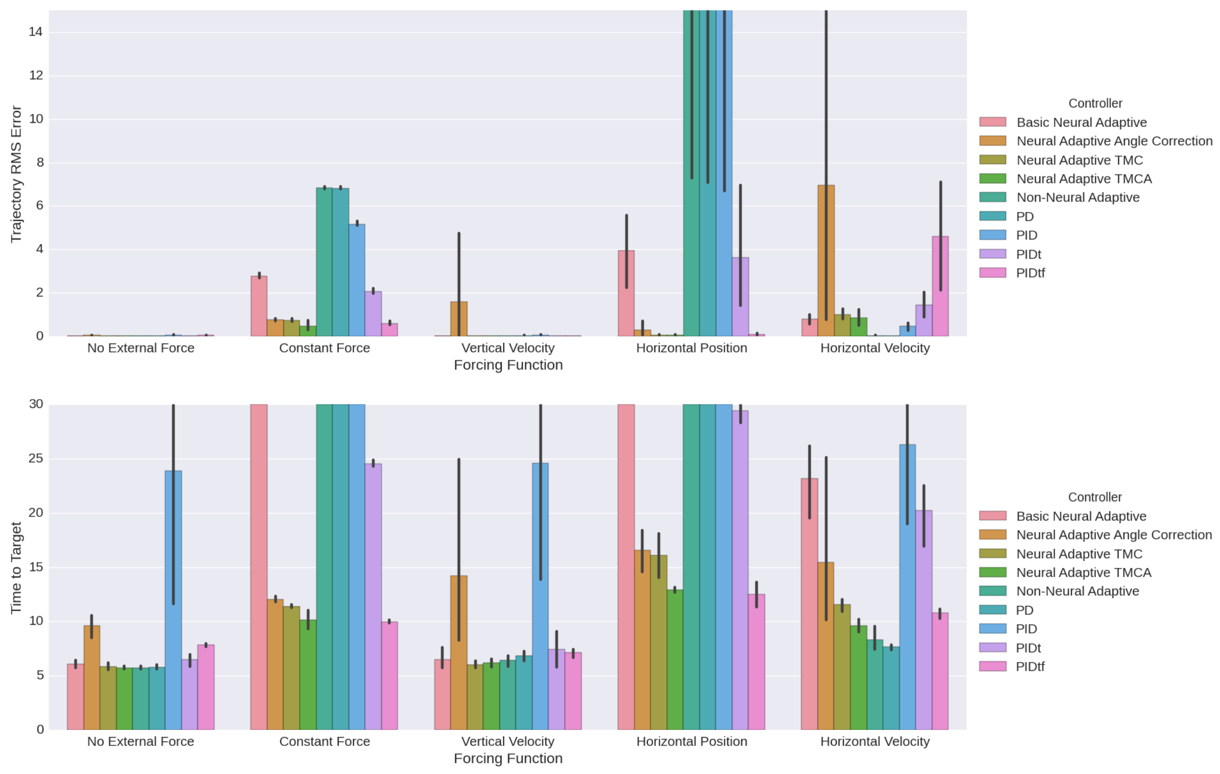


Figure 5.6: Performance on Benchmarks under different External Force Conditions

cases the final iteration of the neural adaptive controller (TMCA) obtained a relatively low RMS error and was the fastest to reach the target in most trials. In the horizontal velocity case this controller was not the fastest to reach its target, but this can be attributed to the fact that once the target is close and the quadcopter velocity is low, there will be little external force (as it is a function of velocity) in this case. A controller that does not adapt at all or adapts slowly will be able to reach steady state more quickly, explaining why the PD controller reached the target the fastest. The non-neural adaptive controller also did well here, which could be caused by a low learning rate. Performance on the task that involved a vertical force was almost the same as if there was no force at all.

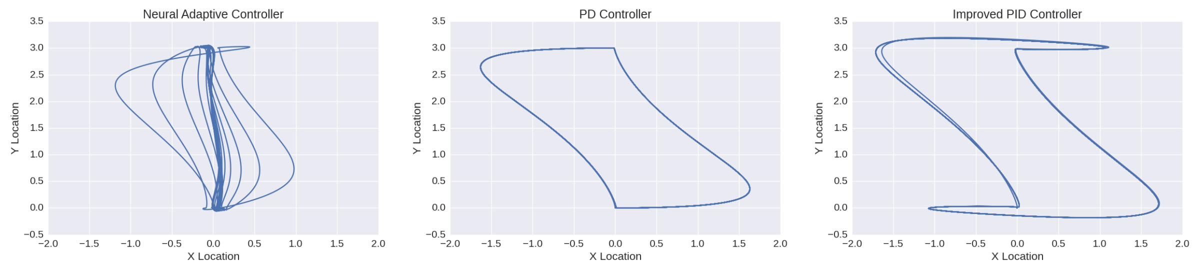


Figure 5.7: Path of Quadcopter between Two Points with External Forces

RMS values for the Neural Adaptive TMCA, PD, and PIDtf controllers are 0.6729, 1.2827, and 1.7662 respectively. These values were calculated using the deviation from the line formed by the two target points as the error.

5.1.4 Improvement over Time

The previous experiments only track performance for a single short run. The neural adaptive controllers are able to learn how to move throughout their environment better over time. An example of this ability is shown in Figure 5.7. The quadcopter is commanded to move back and forth between two points on the x-y plane ([0,0] and [3,0]) with a 5 second delay between each new command. An external force is being applied to the quadcopter along the y direction proportional to its x direction velocity. This causes the path of the quadcopter to become curved rather than a straight line. Over time, the neural adaptive TMCA controller begins to compensate for this external force, and the path between the two points starts to converge towards a straight line. The non-adaptive controllers show no such improvement over time.

5.2 Results

Several ranking methods are presented in order to determine which controller had the best overall performance. The first assigns a numerical ranking to each controller for each metric (RMS error and time-to-target) on each of the seven tasks and computes the sum of those rankings. A rank of 1 is given to the controller with the best performance, a rank of 2 to the second best, a rank of 3 to the third best, and a rank of 4 to all others. As can be seen in Figure 5.8, the neural adaptive TMCA controller obtains the best overall rank.

This ranking method can be useful, but it does not do a good job at taking into account results that are in close proximity or far apart from one another. One approach to

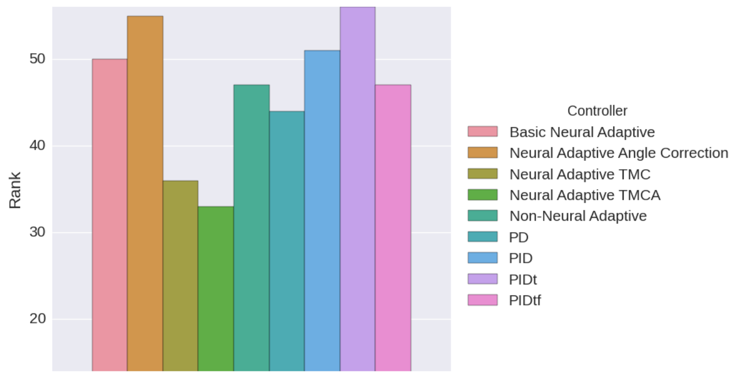


Figure 5.8: Controller Performance Ranking

Final rank is obtained from the sum of the controller's rank on each task. Lower ranks indicate better performance. There are seven tasks with two performance metrics each, resulting in fourteen tasks in total.

overcoming this issue is to compute the mean across all tasks and use that as the ranking. Directly using the mean is sometimes not informative, as the results of each task can be on very different scales. By normalizing the data before taking the mean, the different scales become less of an issue. These results are shown in Table 5.4. The neural adaptive TMCA controller performs the best in all categories except for the normalized RMS error, in which it is tied with the neural adaptive TMC controller.

While the final iteration of the neural adaptive controller did not perform strictly the best in all tasks, it had the strongest overall performance. For the benchmarks that covered simple environments with no external forces, the neural controller performed on par with the reference controllers. For the benchmarks involving external forces, the neural controller achieved the lowest RMS error and the fastest time to reach the target for the majority of the tasks.

Table 5.4: Mean Controller Performance

Controller Name	**Time-to-Target**		**Trajectory RMS**	
	Direct	**Normalized**	**Direct**	**Normalized**
Basic Neural Adaptive	6.77	0.29	0.54	0.12
Neural Adaptive Angle Correction	5.88	0.27	0.69	0.26
Neural Adaptive TMC	4.36	0.21	0.13	**0.08**
Neural Adaptive TMCA	**4.11**	**0.20**	**0.10**	**0.08**
Non-Neural Adaptive	7.20	0.32	2.64	0.21
PD	6.78	0.29	1.73	0.21
PID	10.85	0.47	1.61	0.29
PIDt	6.51	0.28	0.51	0.12
PIDtf	4.32	0.21	0.38	0.16

The *Direct* method computes the mean across all tasks by using the mean for each task directly. The *Normalized* method first normalizes the mean for each task by dividing it by the maximum mean in that task and then uses the mean of that result. Displayed values are rounded to two decimal places. The best result for each method is bolded.

Chapter 6

Discussion and Future Work

6.1 Contributions

The main contribution of this thesis is an adaptive control system for a quadcopter using simulated biological neurons. This is a proof-of-concept that an ensemble of spiking neurons can be used to improve quadcopter control in the face of uncertain and changing environments through a biologically realistic learning mechanism. Using this methodology, state of the art control can be implemented on the low power and highly parallel architecture of neuromorphic hardware.

A second contribution is the integration of the Nengo neural simulation software with the robotics and physics simulation capacity of V-REP. This union allows complex neural models to be embodied in a physical environment in a straightforward manner. The various tools developed over the course of this thesis work to support the integration of the simulated quadcopter with Nengo are general purpose enough to be applied to other simulated robotic models. These tools include a standard method for opening and closing the communication channel for the simulators, synchronization of simulation time between the two simulators, an interface for using sensors and actuators from Nengo, and method for displaying data from a Nengo network using V-REP's plotting interface.

6.2 Discussion

While the majority of this work was conducted in an ideal environment with no sensor noise, the neural adaptive controller can still function with noisy measurements. A few

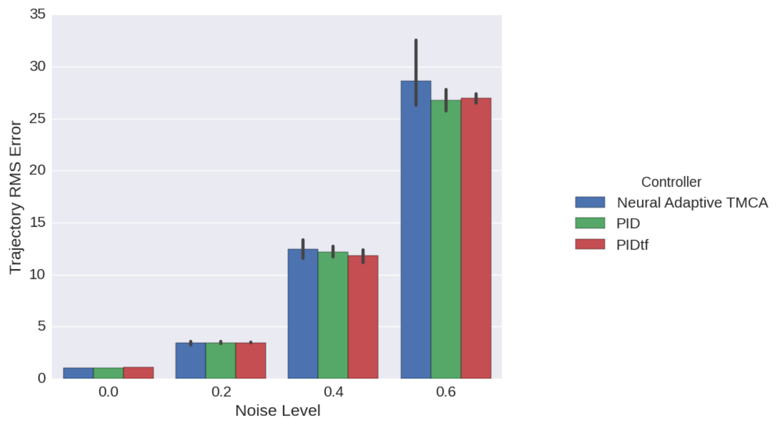


Figure 6.1: Horizontal Motion with Noise

Noise level indicates the variance on the Gaussian noise applied to the state measurements. The noise applied to the angular measurements is one tenth of the linear measurements. A noise level of 0.2 means that the variance on position and velocity measurements is 0.2 and the variance on orientation and angular velocity measurements is 0.02.

experiments were performed with Gaussian noise injected into the sensor measurements and the results compared to the reference PID and PIDtf controllers are shown in Figure 6.1. As expected, all of the controllers perform worse as the amount of noise in the system is increased. Each controller degrades at about the same rate with respect to the increase in noise, with the adaptive controller being affected slightly more than the others. One possible explanation for this is that the adaptive controller is attempting to learn something from the errors caused by noise, but cannot due so correctly because the noise is random. In future work, methods could be implemented to increase robustness to noise, such as smoothing measurements over time.

Another common goal of a control system is path following. This is where instead of the robot being instructed to move to a particular point, it is told to follow a particular path. A series of position/velocity pairs can be sequentially given as set-points rather than a single position set-point. The neural adaptive controller was designed for point-to-point control, but it can also be given a path to follow. A top-view screenshot from V-REP of a simple example demonstrating this capability is shown in Figure 6.2. Here the quadcopter is commanded to continuously fly along a circular path while being subjected to an external force along the y-axis (up/down) proportional to its velocity along the x-axis (left/right). In the beginning, the quadcopter's trajectory is very elliptical due to this external force, but as it continues to fly the adaptive component of the controller learns to compensate

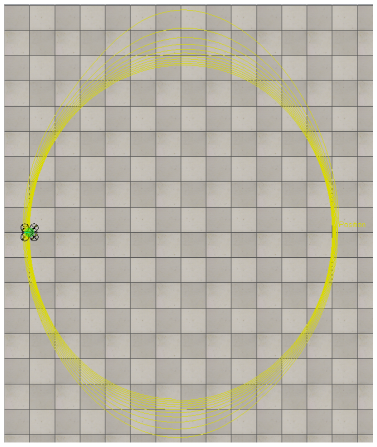


Figure 6.2: Circular Path with External Forces

for this force, eventually producing very circular trajectories.

6.3 Future Work

Further research could usefully explore improvements to this model in terms of both physical accuracy and additional capabilities. The closer the model behaves to a real physical system, the more useful it will be in guiding the design of real world applications. The more capabilities the model displays, the more variety in the applications it can be applied to.

6.3.1 Adaptive Prediction System

In addition to control, the inverse problem of predicting future states is important. An internal model of the system dynamics can be constructed using sensor measurements throughout time and updating its representation based on its own errors of future state prediction. If the environment changes, or the quadcopter or its sensors become damaged, the internal model can adapt to reflect those changes and give predictions for this augmented system. Integrating adaptive control with an adaptive method for system identification could lead to major improvements in overall system performance.

6.3.2 Integrated Navigation and Planning System

A useful extension of the adaptive controller would be to embed it within a larger navigation and planning system. Currently the quadcopter will only fly in a direct path towards a specified point, but it would be useful for it to be able to detect obstacles in its path and calculate a new route to avoid those obstacles. The adaptive ensemble of neurons in the controller could be set up to be able to recognize certain obstacles and learn to avoid them in an effective manner. An internal representation of the environment along with a memory for changes in the environment could be used to allow dynamic planning of optimal trajectories to a target.

6.3.3 Detailed Modelling

Under special conditions, the dynamics of a quadcopter can change dramatically from the ideal model presented in this thesis. One such condition is known as vortex ring state, which can arise when the quadcopter descends too quickly [27]. The quadcopter's rotor blades may enter the turbulent downwash of the air beneath the craft, causing a severe loss of lift. A vortex forms in a circular ring around the rotors' path of rotation, bringing turbulent air from beneath the rotors to above them. Increasing the throttle at this point only makes the vortex stronger, eventually causing a total loss of lift. This is a dangerous situation to encounter, so it could be highly beneficial to include some properties of this state as well as signs of entering this state (typically wobbling of the craft during descent) in the model. With this extra information, the adaptive controller should have an easier time learning how to recover from this state and avoid entering it entirely.

Another condition that was not modelled in simulation is blade flapping. This condition occurs when the rotor blades undergo translational motion; the advancing blade experiences a higher tip velocity while the retreating blade experiences a lower tip velocity [3]. This differential results in an increase in lift in the advancing blade and a decrease in lift in the retreating blade, applying a torque to the rotor disk. The rotor tip path flaps up as the blade advances, and down as the blade retreats to balance the aerodynamic forces. The adaptive controller is designed to be able to account for external non-linear effects, so this would be a good condition to test the controller with in the future.

6.3.4 Realistic Sensors

The current implementation of the simulation is able to directly read the exact state information of the quadcopter. A more realistic scenario would be to use a set of sensors that provide measurements that can be used to estimate the state. Each of these sensors can have different noise properties and accuracy that can change depending on the situation. The robustness of the adaptive control algorithm to changes in the sensor properties and estimation algorithm can be important to characterize when looking into purchasing components to implement the controller on physical hardware.

6.3.5 Running on Physical Hardware

The most practical extension to this work would be to move beyond simulation and run the control system on physical hardware. Many challenges will arise in this undertaking, including selection of the particular hardware platform, sensors to use, and real-time requirements of the control algorithm. Overcoming these challenges to produce an adaptive neural quadcopter that can interact with the real world will be highly beneficial for exploring both practical applications of this work as well as guiding future research direction.

6.4 Conclusion

This project was undertaken to design an adaptive quadcopter controller capable of being implemented on neuromorphic hardware and evaluate its performance with respect to conventional controller design methods. The results obtained in simulation are highly promising and warrant future investigation of more sophisticated neural navigation and planning systems as well as implementation on physical hardware.

APPENDICES

Appendix A

Source Code for Thesis Work

All of the source code used to create the quadcopter controllers presented in this thesis is available for download at: https://github.com/bjkomer/masters-thesis

Instructions for downloading and installing the necessary software to run the simulations and benchmarks are also provided at this link.

References

[1] N. H. El-Amary A. I. Hashad A. Y. Elruby, M. M. El-khatib. Dynamic modeling and control of quadrotor vehicle.

[2] Richard Baguley. Best drones 2015.

[3] Moses Bangura and Robert Mahony. Nonlinear dynamic modeling for high performance control of a quadrotor. In *Australasian conference on robotics and automation*, pages 1–10, 2012.

[4] Andrew G Barto, Richard S Sutton, and Charles W Anderson. Neuronlike adaptive elements that can solve difficult learning control problems. *Systems, Man and Cybernetics, IEEE Transactions on*, (5):834–846, 1983.

[5] Trevor Bekolay, James Bergstra, Eric Hunsberger, Travis DeWolf, Terrence C Stewart, Daniel Rasmussen, Xuan Choo, Aaron Russell Voelker, and Chris Eliasmith. Nengo: a python tool for building large-scale functional brain models. *Frontiers in neuroinformatics*, 7, 2013.

[6] Trevor Bekolay, Carter Kolbeck, and Chris Eliasmith. Simultaneous unsupervised and supervised learning of cognitive functions in biologically plausible spiking neural networks. In *Proceedings of the 35th annual conference of the cognitive science society*, pages 169–174, 2013.

[7] James Bergstra, Dan Yamins, and David D Cox. Hyperopt: A python library for optimizing the hyperparameters of machine learning algorithms. In *Proceedings of the 12th Python in Science Conference*, pages 13–20, 2013.

[8] Anthony N Burkitt. A review of the integrate-and-fire neuron model: I. homogeneous synaptic input. *Biological cybernetics*, 95(1):1–19, 2006.

[9] Chien-Chern Cheah, Chao Liu, and Jean-Jacques E Slotine. Adaptive tracking control for robots with unknown kinematic and dynamic properties. *The International Journal of Robotics Research*, 25(3):283–296, 2006.

[10] M. Freese E. Rohmer, S. P. N. Singh. V-rep: a versatile and scalable robot simulation framework. In *Proc. of The International Conference on Intelligent Robots and Systems (IROS)*, 2013.

[11] Gilberto Echeverria, Nicolas Lassabe, Arnaud Degroote, and Séverin Lemaignan. Modular open robots simulation engine: Morse. In *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, pages 46–51. IEEE, 2011.

[12] Chris Eliasmith. How to build a brain: From function to implementation. *Synthese*, 159(3):373–388, 2007.

[13] Chris Eliasmith. *How to build a brain: A neural architecture for biological cognition.* Oxford University Press, 2013.

[14] Chris Eliasmith and Charles H Anderson. *Neural engineering: Computation, representation, and dynamics in neurobiological systems.* MIT press, 2004.

[15] Elena Garcia, Maria Antonia Jimenez, Pablo Gonzalez De Santos, and Manuel Armada. The evolution of robotics research. *Robotics & Automation Magazine, IEEE*, 14(1):90–103, 2007.

[16] Andrew Gibiansky. Quadcopter dynamics, simulation, and control, 2012.

[17] Sree Harsha, Balaji, and Divya Kamath. Quadcopter, 2014.

[18] Leslie A Hart. *How the brain works: A new understanding of human learning, emotion, and thinking.* Basic Books, 1975.

[19] Jennifer Hasler and Bo Marr. Finding a roadmap to achieve large neuromorphic hardware systems. *Frontiers in neuroscience*, 7, 2013.

[20] NG Hockstein, CG Gourin, RA Faust, and DJ Terris. A history of robots: from science fiction to surgical robotics. *Journal of robotic surgery*, 1(2):113–118, 2007.

[21] Nathan Koenig and Andrew Howard. Design and use paradigms for gazebo, an open-source multi-robot simulator. In *Intelligent Robots and Systems, 2004.(IROS 2004). Proceedings. 2004 IEEE/RSJ International Conference on*, volume 3, pages 2149–2154. IEEE, 2004.

[22] Gerard Lacey and Kenneth M Dawson-Howe. The application of robotics to a mobility aid for the elderly blind. *Robotics and Autonomous Systems*, 23(4):245–252, 1998.

[23] Teppo Luukkonen. Modelling and control of quadcopter. *Independent research project in applied mathematics, Espoo*, 2011.

[24] Jonas Mockus, Vytautas Tiesis, and Antanas Zilinskas. The application of bayesian methods for seeking the extremum. *Towards Global Optimization*, 2(117-129):2, 1978.

[25] Illah R Nourbakhsh, Katia Sycara, Mary Koes, Mark Yong, Michael Lewis, and Steve Burion. Human-robot teaming for search and rescue. *Pervasive Computing, IEEE*, 4(1):72–79, 2005.

[26] Robert M Sanner and Jean-Jacques E Slotine. Gaussian networks for direct adaptive control. *Neural Networks, IEEE Transactions on*, 3(6):837–863, 1992.

[27] Quadcopter Flight School. Vortex ring state (the wobble of death), 2014.

[28] Jean-Jacques E Slotine and Weiping Li. On the adaptive control of robot manipulators. *The international journal of robotics research*, 6(3):49–59, 1987.

[29] Jean-Jacques E Slotine, Weiping Li, et al. *Applied nonlinear control*, volume 199. Prentice-hall Englewood Cliffs, NJ, 1991.

[30] Terrence C Stewart, Trevor Bekolay, and Chris Eliasmith. Neural representations of compositional structures: Representing and manipulating vector spaces with spiking neurons. *Connection Science*, 23(2):145–153, 2011.