Faculty of Engineering
Department of Systems Design Engineering

NengoAR: A Neural Quadcopter Control System

A Report Submitted in Partial Fulfillment of the Requirements For SYDE 556

April 24, 2014
Course Instructor: Dr. Terry Stewart

Chris Vandevelde, #20316110
4B Systems Design Engineering

Introduction

This report describes an effort to use a neural simulation framework as a control system to a commercial-grade quad-rotor helicopter (“quadcopter”), a type of drone which has recently become popular as both hobbyist and commercial or industrial Unmanned Aerial Vehicles (UAVs). The goal of the project was to determine the feasibility of a neural framework for use for control systems, and what impact neural representations of those controls would have in the performance.

A Parrot AR Drone was used for the quadcopter in this project, which allowed for quick, cheap prototyping of the control algorithm as well as a low overhead cost of hardware. It was thankfully durable enough to survive this testing phase, as the development process involved a large amount of manually-tweaking control parameters through test flights. The slightest error in control would send the drone out of stable flight and cause a crash - there were many flight logs recorded over the duration of this project which end in an unstable crash.

While the control algorithms that power the AR Drone in its production state are complex and incorporate many sources of information and stability controls, it will be determined if such complexity is needed or is just a consequence of the precision and rigour required in the math necessary to describe the system.

System Description

The system comprises both hardware and software in order to mimic a real-world application of a neural system. The simulated brain is running on a computer and communicates with a quadcopter over Wi-Fi to control it.

Quadcopter

The quadcopter in use is a Parrot AR Drone, a commercial quad-rotor helicopter designed for hobbyist use and for sale to the public [1]. It has four independently-controlled rotors which allow for full three-dimensional flight through an onboard control system, run through a miniature onboard PC. It includes a variety of onboard sensors which feed into this PC, and has two cameras for downward and forward vision.

Figure 1: The Parrot AR Drone with the foam indoor shield around the propellors.

Controls

The Parrot AR Drone's control system is quite complex, and incorporates data from a variety of sensors including ultrasonic distance estimation, an accelerometer, a Global Positioning System (GPS) receiver, as well as computer vision techniques for object recognition and tracking.

Though its interface is typically controlled via basic commands, designed for control via a joystick-style interface, there have been several attempts [2] [3] [4] to extend its control beyond simple applications and into more complex aerobatics.

Communication

The Parrot AR Drone communicates via Wi-Fi connection, accepting commands on one port, reporting status on another, and feeding live video through a third. It emits regular packets of information which includes velocity in three dimensions, the status of the Euler angles, the altitude, and some internal details. The status packets follow the following format - `phi`, `psy` and `theta` being the Euler angles, `vx`, `vy`, and `vz` being the velocity in the Cartesian directions, `battery` and `altitude` being relatively self-explanatory, and `num_frames` being the number of video frames processed to calculate

the data, and `ctrl_state` representing various details about the state of the controller algorithm.

```
{
      'phi': 3,
      ‘psi': 3,
      ‘num_frames': 383,
      ‘battery': 50,
      ‘altitude': 221,
      ‘ctrl_state': 131072,
      ‘vx': 0.0,
      ‘vy': 0.0,
      ‘vz': 0.0,
      ‘theta': 7
}
```

Neural Representation

The system is comprised of five Ensembles, each representing a different quantity in the brain, broken up into three categories: position (position vectors, or point coordinates), trajectory (a direction vector), and motor control (a vector of power levels to each motor).

Position

The first two hold the goal position and current position as three-dimensional vectors, comprised of 100 neurons with a radius of 10. The goal position is a simple static input read from a CSV file which is used to hold the path the quadcopter should take. The current position ensemble starts at (0, 0, 0) and begins to move with input from the quadcopter’s state information. Every time the state is polled, it returns a vector representing its velocity, which is multiplied by the time difference and added to the current state to represent the new position.

To hold the current state, an integrator connection was created to keep the value constant, changing only with the differential input. When this was introduced, the current position began to drift toward certain points due to attractor noise, but a modification of the transfer function was made to corrected this error, stabilizing the position from such drift error (see sub-section “Drifting” in the Implementation section, below).

Trajectory

The second two are tasked with calculating and holding the trajectory, or the vector leading from the current position to the goal position calculated by vector difference. Two ensembles are needed for calculating the difference; the first to combine the values into one vector, the second to find the difference between the two sections of that vector.

Motor Control

The last ensemble holds a vector containing four power ratios which will be sent to each motor. Through testing (see Implementation section, below) it was determined that the

values range from 0 (motors off), to 500 (motors at highest power), with 250 being the "hovering power." Thus, if all motors are set to 250, the quadcopter will hover in place.

The function to calculate the powers is based on the trajectory vector, and uses the positions of each motor to induce functions which will move the quadcopter in each cardinal direction. For example, to move in the positive z-direction, all motors should increase their power. The transformation function is described in more detail in the Implementation section, below.

Design Specification

As there are (unfortunately) no natural quad-rotor helicopter beings in nature, a search for studied properties of neural ensembles which make up a control system for quad-rotor locomotion turned up no results. Even rotor-based propulsion as seen in helicopters are absent in nature, evolution having favoured flapping wings over spinning propellors. However, the flight characteristics which helicopters are favoured for (the ability to hover and free-flight in all three dimensions) are also desired by some animals in the wild. As such, there are *some* animals in the wild who have similar flight patterns to a quad-rotor copter, and can be used for inspiration towards a model for neural representation of a suitable control system. Each of these were investigated for specifications which could be used.

In general, the hippocampus is the area most associated with spatial cognition and navigation, so this project replicates a simplistic hippocampus made up of a fraction of the neurons in a typical animal. Common hippocampus neural types are place cells (for current location), head-direction cells (for point of view), grid cells (to locate environmental features in a hexagonal grid), and border cells (to denote divisions between environments) [5].

Birds

Birds were the first subject of inquiry, being the first thing brought to mind when "animals that fly" are considered. Neural representations of spatial position, orientation, and path have been characterized, and a variety of sensory systems are being used, each contributing a slightly-different representation in various settings. They have a strong sense of direction, even estimated to have a genetic sense of direction and distance for migratory patterns, so they can find winter breeding grounds without any previous experience [6]. Evidence from studies recording in homing pigeons indicated firing rates of 0 - 3Hz for place cells and 0 - 30Hz for path cells, with neurons highly localized to specific locations in the environment [7].

Hummingbirds

Hummingbirds have a very specific flight style, characterized by high-speed wing-flapping and high precision, allowing for a variety of flight patterns. They have the ability to hover in place and dart around freely in three dimensions, very similarly to helicopters and quadcopters [8]. Their spatial representations have been studied and shown to be three-dimensional, albeit biased towards horizontal over vertical dimensions [9].

Insects

Dragonflies and damselflies are able to fly in a similar manner as hummingbirds, having the ability to hover and change velocity quickly and precisely. They also have multiple independent pairs of wings, which can be flapped independently from each other, giving them another characteristic which mirrors quadcopter flight dynamics [10].

Implementation

The neural implementation was done using the latest version of the Nengo software in Python [11]. Python was chosen for its development speed, easy manipulation of matrices and vectors through numpy, and large community of developers providing application libraries for a large variety of applications. It was also chosen due to the Parrot AR Drone having an avid hobbyist-programmer community, who have created a slew of libraries to connect and remotely control the drone, including several Python libraries [12] [13]. In the end, libardrone was chosen due to its proven effectiveness, open-source license, and small source code base.

While libardrone only had a unfinished implementation of a function which allowed for direct control of the four motors (bypassing the drone's internal control system), development was undertaken to determine the parameters required and complete the functionality. It was found that the inputs take integers between 0 and 500 for power units, with the middle value of 250 being set to provide 1/4 of the force of gravity on the drone due to its mass. This way, if all four motors are set to 250, the drone hovers in place.

Control Development

Initial research was done to determine what would be necessary to control an unstable system such as a quadcopter for stable flight. The control algorithm in the onboard computer is complex and robust, incorporating data from different sensors to give a more accurate sense of its current position, velocity, angle, and what needs to be done to maintain a stable flight pattern [14].

While such a control system could be developed, a more simplistic controller was used to aid development, with the aim of increasing complexity if it was needed. It was also done in an attempt to see more directly the effect the neural representation would have on the quadcopter's behaviour.

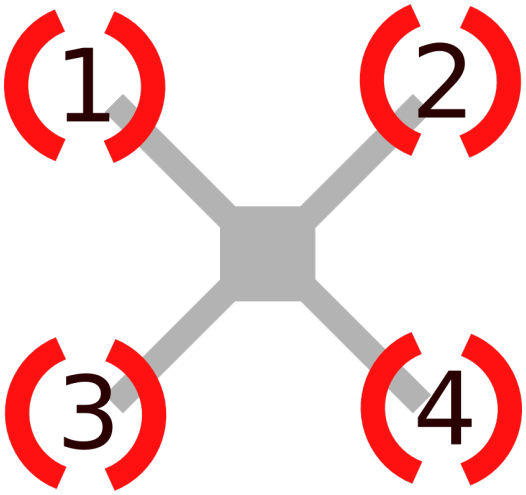


Figure 2: The number system of the quadcopter's engines.

To move to the left, the motors on the right side should increase their power and the motors on the left side should decrease their power. Similarly, to move forward the backwards motors should increase power and the forwards motors should decrease power. The function, taking in v as the vector trajectory.

$$f(v) = \{ v_z + v_x - x_y,\ v_z - v_x - x_y,\ v_z + v_x + x_y,\ v_z - v_x + x_y \}$$

Or, in code:

```
engines = [ trajectory[2] + trajectory[0] - trajectory[1],
            trajectory[2] - trajectory[0] - trajectory[1],
            trajectory[2] + trajectory[0] + trajectory[1],
            trajectory[2] - trajectory[0] + trajectory[1] ]
```

While the amount each motor should move can be tuned to perfectly correspond to physical units, to keep the system simplest all tuning was left to the control algorithm. The motors will simply turn a position error (a nonzero trajectory between goal and current position) into power and move towards the goal at all times.

Drifting

To be able to view the performance of the controller, the neural representations were recorded to Comma-Separated Value (CSV) files, which were then plot in three dimensions using MATLAB. This allowed for a visualization of how the controller was performing. As can be seen in Figure 3 below, the initial implementation of the algorithm had a problem with drift in the current position representation

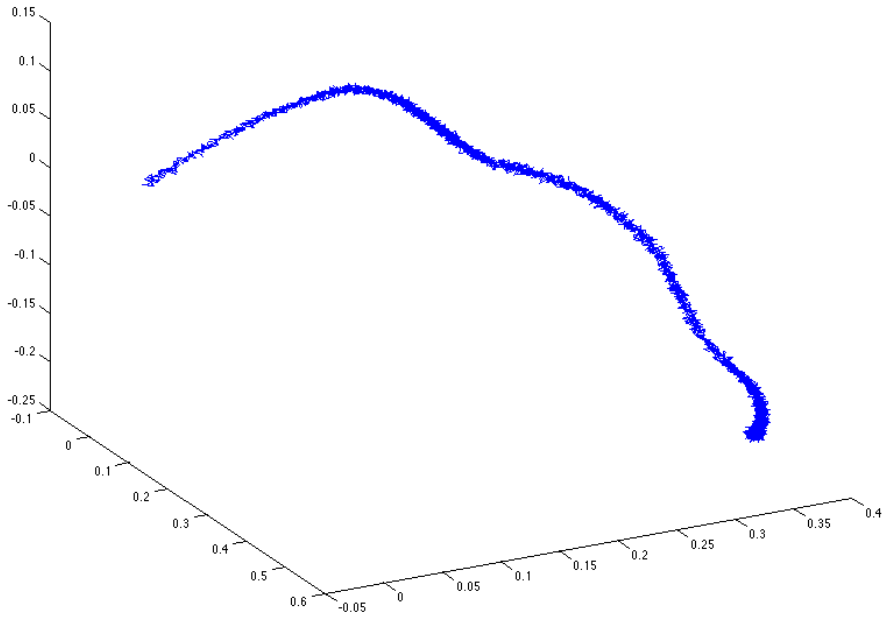
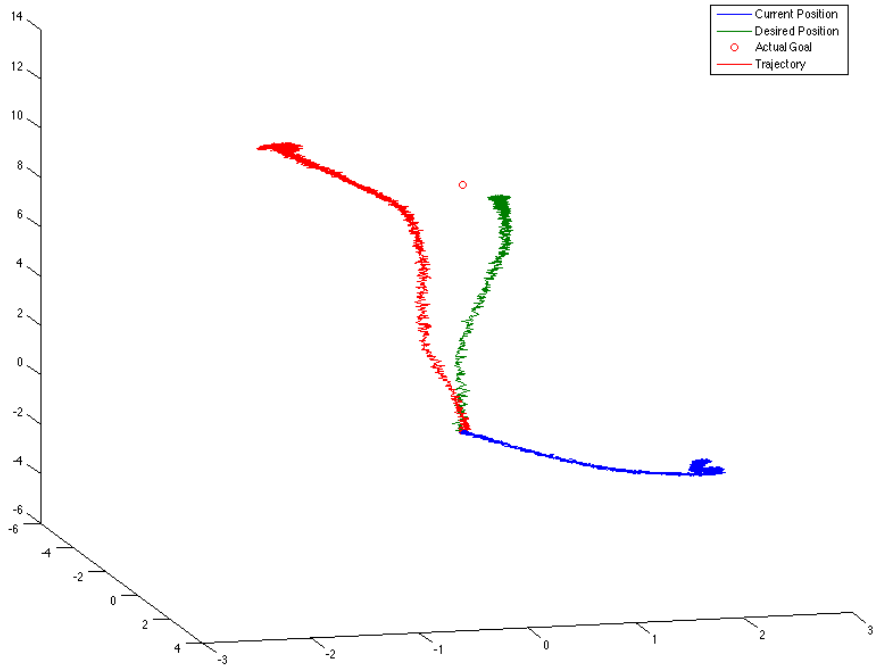


Figure 3: The neural representation of the quadcopter's current position with no input signal. Note the drift in all three dimensions. Temporally, it starts on the left and moves to stabilize at the bottom-right.

This was due to the integrator which was being used to store the quadcopter's position; the sensors in the Parrot AR report velocity, which were multiplied by the elapsed time and integrated into the current position representation. The representation noise introduced local attractors into the representation, which the position would drift towards.

Unfortunately, this had a rather negative effect on the control algorithm, and introduced an error into the signal which the system attempted to correct. This caused the trajectory (and, correspondingly, the direction of the engine power) to drift accordingly,

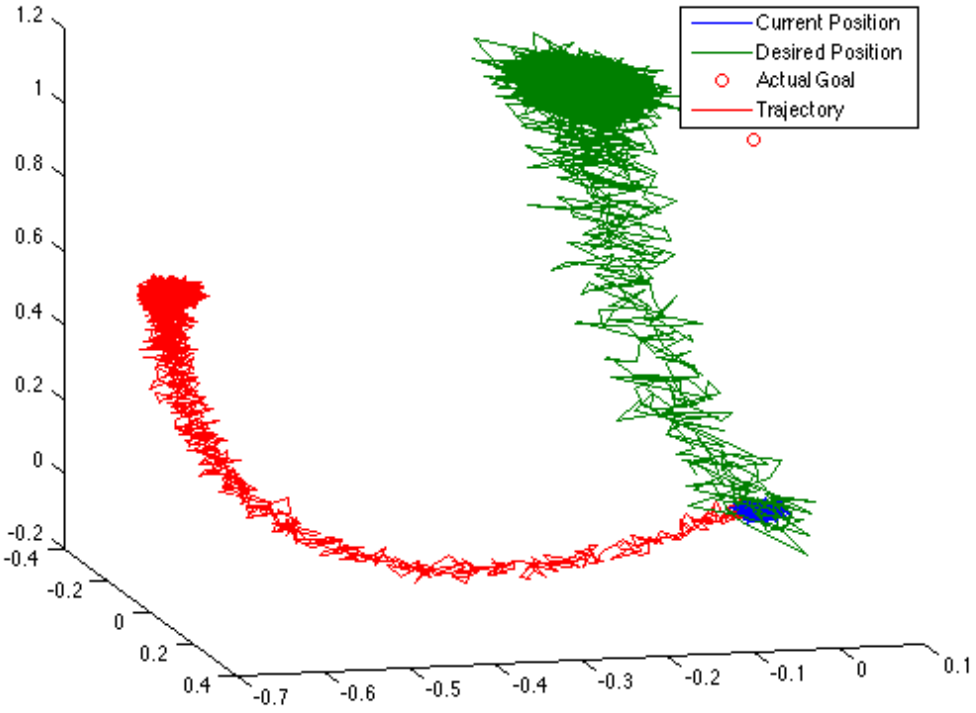


causing instability and crashing in a random direction every time the algorithm was run.

Figure 4: The effect of the representation noise, introducing drift on the neural representations of the current position (and the trajectory, accordingly).

As can be seen in Figure 4, the desired position (in green) moves towards its input signal (the small red circle in the centre) and hovers near the area in a stable position. The current position (in blue), however, drifts diagonally towards the bottom-right of the figure before stabilizing at roughly (2.5, 1.8, 0), despite starting at (0, 0, 0) and having no input. The trajectory, shown in red, moves towards the opposite direction to compensate for the error and move the current position towards the actual goal.

It was assumed that the drift was due to the dynamics of the integrator connection, owing to its random direction and stabilization at an attractor point, so the transfer



function was changed from $f(x) = x$ to $f(x) = \tau x - x$. This compensated for the effect of the decoding on the signal, which fixed the drifting problem, as seen in Figure 5, below.

Figure 5: The corrected current position circuit (in blue, stable at (0, 0, 0)) and the desired position and trajectory (in green and red, correspondingly).

Turn Scaling

Through the development process, it was found that much of the instability was due to highly aggressive turning, so a turn scaling parameter was introduced to allow for aggressive height manoeuvring but reduce the acceleration in the horizontal plane. This changed the transfer function to:

$$f(v) = \{ v_z + \alpha v_x - \alpha x_y,\ v_z - \alpha v_x - \alpha x_y,\ v_z + \alpha v_x + \alpha x_y,\ v_z - \alpha v_x + \alpha x_y \}$$

And, in code:

```
engines = [ trajectory[2] + TURN_SCALING*trajectory[0] - TURN_SCALING*trajectory[1],
            trajectory[2] - TURN_SCALING*trajectory[0] - TURN_SCALING*trajectory[1],
            trajectory[2] + TURN_SCALING*trajectory[0] + TURN_SCALING*trajectory[1],
            trajectory[2] - TURN_SCALING*trajectory[0] + TURN_SCALING*trajectory[1] ]
```

When reduced, the difference in engine power was reduced correspondingly, giving an overall more stable flight path for a given trajectory. However, it did reduce the speed at which it was able to move in the horizontal plane, and required manual tweaking (or potentially optimization) to find a balance.

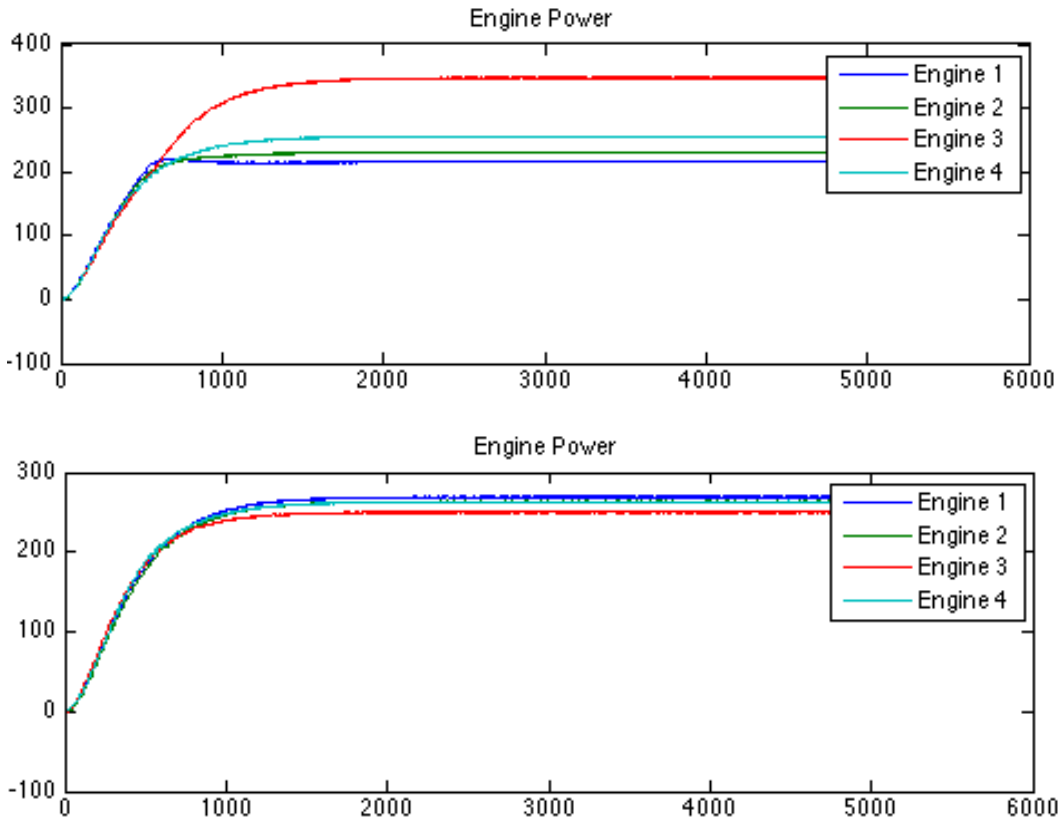


Figure 6: A comparison showing the difference in engine powers with a turn scaling of 1 (above) and 0.001 (below). Note that the engine values themselves are between different trials, and aren't just scaled.

Filtering

While the goal positions were particularly stable and slow-moving (the goal being stationary), the trajectory and engine power needed to be quick and responsive to respond to error and avoid instability. The filtering for the trajectory and engines were reduced to allow for more quick movement, resulting in more aggressive error correction.

Results

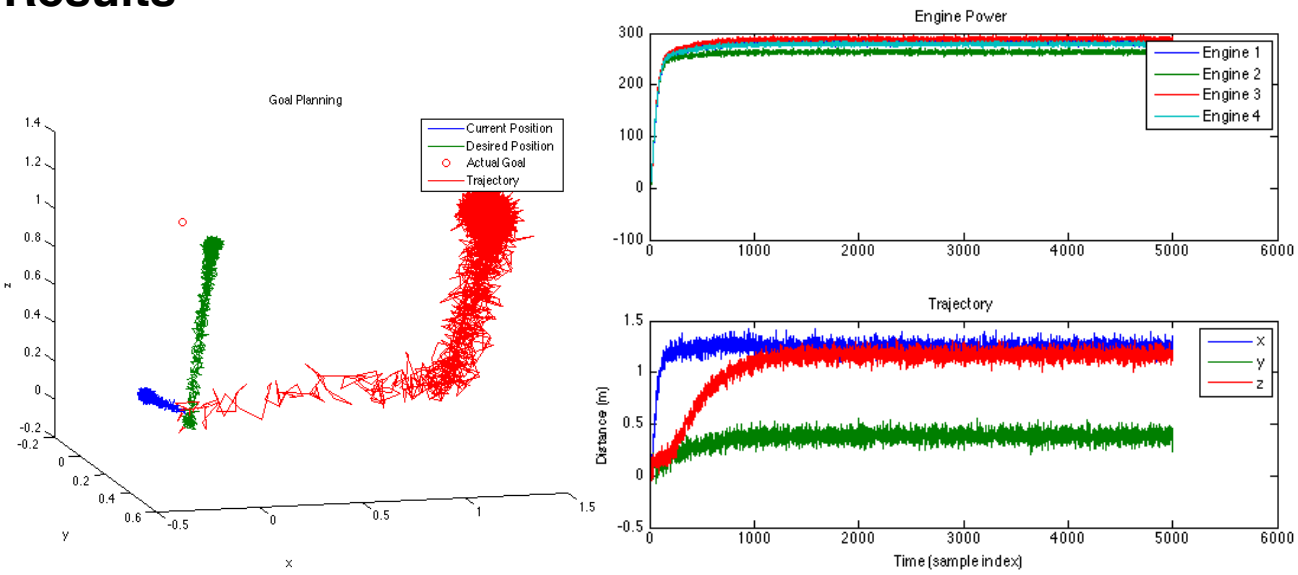


Figure 7: The trajectory and engine power with the more stable constants used.

In the end, the quadcopter's flight was only semi-stable, although able to hover and drift in a predictable trajectory. The problem of representation drift and accuracy amplified the problems involved in sustaining flight, but with tweaking of the filtering constants allowing for more high-frequency signal responses, the quadcopter's behaviour was able to stay at least predictable. Unfortunately, at that point the representation noise levels were also raised, so the accuracy was sacrificed.

Improvements

In future iterations, separating different axes for representation would be an interesting avenue to consider, allowing for different filtering options in different directions similarly to the separation between goal and trajectory filters. This follows the biological constraints discussed in the design specification, with vertical resolution being lower than horizontal, to allow for more precise movement in the horizontal direction.

In addition, the complexity of the controller could be increased to give a stronger response to error. As it stands, the controller is proportional to the error, but a derivative or integral component could be added to give a more responsive signal - right now the motors rarely reach anywhere near full power, so there is room for improvement.

Conclusion

Perhaps the most interesting part of the neural control algorithm is how much it resembled a traditional control algorithm, and suffers from similar problems, albeit low-pass filtered and noisy. The problems of proportional control are still present in neural systems, however that indicates that the potential solutions of using more complex control systems (including derivative and integral control) are also viable.

Also interesting was the parallel between the natural world of hummingbirds and dragonflies and the very *un*natural propulsion system of a quad-rotor helicopter. Where technology mimics nature, advantageous techniques from one can be used in another, so it is beneficial to study the natural world even in situations which may seem distinct.

The ultimate lesson to be learned from this project is that, while a simulated neural network is a complex and beautiful system, it is the learning and experience which is built on top of this which is where the true complexity of the brain lies. Not simply in the design of neurons and spike encoding or decoding, but in the actual weights of the connections, where the learning itself happens. That complexity, the subtleties of behaviour, is where instability can be managed and flight brought forth.

Works Cited

1. Parrot SA. *AR Drone 2.0*. [Online]. Available: http://ardrone2.parrot.com/
2. Drone Apps. *Drone Station for Mac*. [Online]. Available: http://drone-apps.com/drone-station-for-mac/
3. Parrot SA. *Developer Zone*. [Online]. Available: http://ardrone2.parrot.com/developer-zone/
4. Parrot SA. *Apps*. [Online]. Available: http://ardrone2.parrot.com/usa/apps/
5. T. J. Wills, et al., "The development of spatial behaviour and the hippocampal neural representation of space," *Phil. Trans. R. Soc.* vol. 369.
6. V. Bingman, T. Jechura, & M. C. Kahn, "Behavioral and Neural Mechanisms of Homing and Migration in Birds," in *Animal Spatial Cognition: Comparative, Neural, and Computational Approaches*, M.F. Brown and R.G. Cook, Eds., 2006. [Online] Available: http://pigeon.psy.tufts.edu/asc/Bingman/Default.htm
7. G.E. Hough II and V.P. Bingman, "Spatial response properties of homing pigeon hippocampal neurons: Correlations with goal locations, movement between goals, and environmental context in a radial-arm arena," J*ournal of Comparative Physiology A*, vol. 190, pp. 1047-1062.
8. About.com. *How Hummingbirds Fly*. [Online]. Available: http://birding.about.com/od/birdbehavior/a/How-Hummingbirds-Fly.htm
9. I. N. Flores-Abreu et al., "Three-dimensional spatial learning in hummingbirds," *Animal Behaviour*. vol. 85 (2013), pp. 579-584, Jan. 2013. [Online]. Available: http://rspb.royalsocietypublishing.org/content/281/1784/20140301.abstract
10. J. M. Wakeline and C. P. Ellington, "Dragonfly Flight II. Velocities, Accelerations and Kinematics of Flapping Flight," *The Journal of Experimental Biology*. vol. 200, pp. 557-582, 1997. [Online]. Available: http://jeb.biologists.org/content/200/3/557.full.pdf
11. Centre for Theoretical Neuroscience. *Nengo: Large-scale brain modelling in Python*. [Online]. Available: https://github.com/ctn-waterloo/nengo
12. S.D. Levy. *AR.Drone AutoPylot - Auto-Pilot the Parrot AR.Drone from Python (or Matlab or C)*. [Online]. Available: http://home.wlu.edu/~levys/software/ardrone_autopylot/
13. B. Venthur. *python-ardrone*. [Online]. Available: https://github.com/venthur/python-ardrone
14. P. Bristeau et al., "The Navigation and Control technology inside the AR.Drone micro UAV," in *International Federation of Automatic Control (IFAC) World Congress*, Milano, Italy, 2011, pp.1477-1484. [Online]. Available: http://cas.ensmp.fr/~petit/papers/ifac11/pjb.pdf

Appendix A - Code

nengoAR.py

```
import nengo
import libardrone
import numpy
import csv
import time

# This scales the velocity proportionally to the error.
# Remember, velocity saturates at 0 and 500.
VELOCITY_SCALING = 0.05

# This scales the x- and y- turning velocity independently from the lift
# velocity. We don't care too much about lift and drop speed, but if it turns
# too fast in one direction, it gets more unstable.
TURN_SCALING = 0.001

POSITION_FILTER = 0.2 # A filter for the position signal values.
ENGINE_FILTER = 0.03 # A filter for the engine/trajectory signal values.

MAX_SPEED = 500; # Constant defined from the quadcopter itself.

desiredfile = open('desired.csv', 'wb')
desiredwriter = csv.writer(desiredfile)

currentfile = open('current.csv', 'wb')
currentwriter = csv.writer(currentfile)

trajectoryfile = open('trajectory.csv', 'wb')
trajectorywriter = csv.writer(trajectoryfile)

enginesfile = open('engines.csv', 'wb')
engineswriter = csv.writer(enginesfile)

path = numpy.genfromtxt('path.csv', delimiter=',')

SIMULATE = False

if not SIMULATE:
    # Connect to the quad and take off.
    drone = libardrone.ARDrone()
    drone.reset()

# Quad position input.
def path_input(time, x):
    for index, point in enumerate(path):
        if len(path) > 1 and time >= min(path[index - 1][3], 0) and time < point[3]:
            return point[:-1]

    # If we're at the end, stay at the last point.
    return path[-1][:-1]

def drone_state(time):
    if not SIMULATE and drone.navdata.keys():
        return [drone.navdata[0]['vx'], drone.navdata[0]['vy'], drone.navdata[0]['vz']]
    else:
```

```
        return [0, 0, 0]

def send_motors(time, motor_values):
    if not SIMULATE:
        drone.manual(motor_values[0], motor_values[1], motor_values[2], motor_values[3])
    engineswriter.writerow([motor_values[0], motor_values[1], motor_values[2],
motor_values[3]])
    return motor_values

def power(trajectory):
    engines = numpy.array([ trajectory[2] + TURN_SCALING*trajectory[0] -
TURN_SCALING*trajectory[1],
                            trajectory[2] - TURN_SCALING*trajectory[0] -
TURN_SCALING*trajectory[1],
                            trajectory[2] + TURN_SCALING*trajectory[0] +
TURN_SCALING*trajectory[1],
                            trajectory[2] - TURN_SCALING*trajectory[0] +
TURN_SCALING*trajectory[1] ])
    engines = (engines*VELOCITY_SCALING)*MAX_SPEED + MAX_SPEED/2
    engines = numpy.clip(engines, 0, 500)
    return engines

def printdesired(time, value):
    desiredwriter.writerow(value)
    return value

def printcurrent(time, value):
    currentwriter.writerow(value)
    return value

def printtrajectory(time, value):
    trajectorywriter.writerow(value)
    return value

tau = 0.1
def integrator(x):
    return tau*x - x

def difference(x):
    return numpy.subtract(x[:3], x[3:])

# Build our neural model.
model = nengo.Model('Nengo AR Brain')

# Create input nodes representing the input and store it in an ensemble.
path_input = nengo.Node(output=path_input, size_in=3)
velocity_input = nengo.Node(output=drone_state, size_out=3)
motor_output = nengo.Node(send_motors, size_in=4)

print_desired = nengo.Node(printdesired, size_in=3)
print_current = nengo.Node(printcurrent, size_in=3)
print_trajectory = nengo.Node(printtrajectory, size_in=3)

desired_position = nengo.Ensemble(neurons=100, dimensions=3, radius=10)
current_position = nengo.Ensemble(neurons=100, dimensions=3, radius=10)
trajectory_difference_holder = nengo.Ensemble(neurons=200, dimensions=6, radius=10)
trajectory = nengo.Ensemble(neurons=300, dimensions=3, radius=10)
engine_power = nengo.Ensemble(neurons=500, dimensions=4, radius=500)
```

```
# Store the other values in neurons.
nengo.Connection(path_input, desired_position, filter=POSITION_FILTER)
nengo.Connection(desired_position, print_desired, filter=POSITION_FILTER)

# Integrate the velocity to get the current position.
nengo.Connection(velocity_input, current_position, filter=POSITION_FILTER)
nengo.Connection(current_position, current_position, function=integrator,
filter=POSITION_FILTER)
nengo.Connection(current_position, print_current, filter=POSITION_FILTER)

# Trajectory is the difference between the desired and current position.
nengo.Connection(desired_position, trajectory_difference_holder, transform=[[1, 0, 0], [0, 1,
0], [0, 0, 1], [0, 0, 0], [0, 0, 0], [0, 0, 0]], filter=POSITION_FILTER)
nengo.Connection(current_position, trajectory_difference_holder, transform=[[0, 0, 0], [0, 0,
0], [0, 0, 0], [1, 0, 0], [0, 1, 0], [0, 0, 1]], filter=POSITION_FILTER)
nengo.Connection(trajectory_difference_holder, trajectory, function=difference,
filter=ENGINE_FILTER)
nengo.Connection(trajectory, print_trajectory, filter=ENGINE_FILTER)

# Now translate the desired trajectory into engine power.
nengo.Connection(trajectory, engine_power, function=power, filter=ENGINE_FILTER)
nengo.Connection(engine_power, motor_output, filter=ENGINE_FILTER)

# Create our simulator
sim = nengo.Simulator(model)
sim.run(5)

if not SIMULATE:
    drone.manual(0, 0, 0, 0)
    drone.halt()
```

plot_route.m

```
current = csvread('current.csv');
desired = csvread('desired.csv');
trajectory = csvread('trajectory.csv');
path = csvread('path.csv');
engines = csvread('engines.csv');

figure;
hold all;
plot3(current(:, 1), current(:, 2), current(:, 3));
plot3(desired(:, 1), desired(:, 2), desired(:, 3));
plot3(path(:, 1), path(:, 2), path(:, 3), 'rO');
plot3(trajectory(:, 1), trajectory(:, 2), trajectory(:, 3));
legend('Current Position', 'Desired Position', 'Actual Goal', 'Trajectory');
title('Goal Planning');
xlabel('x');
ylabel('y');
zlabel('z');
view(14, -14);

figure;
subplot(2, 1, 1);
plot(engines);
title('Engine Power');
legend('Engine 1', 'Engine 2', 'Engine 3', 'Engine 4');
subplot(2, 1, 2);
plot(trajectory);
```

```
title('Trajectory');
legend('x', 'y', 'z');
xlabel('Time (sample index)');
ylabel('Distance (m)');
```