Neato Quadcopters

Alex Burka & Seth Foster

May 11, 2012

Abstract

In this paper we detail the implementation and execution of an unsuccessful experiment involving simulated quadcopters trained using NEAT. Quadcopters are simulated using the robotics framework ROS, with the physics simulator Gazebo and the packages provided by Team Hector Darmstadt, and the controllers are implemented using neural networks. Unfortunately, the current experiments failed due to collusion between bugs in our simulator platform and the inscrutability of genetic algorithms. However, the techniques and code we developed could be of use to others wishing to implement similar experiments. The code can be found at `www.alexburka.com/neatoquadcopters.html`.

1 Introduction

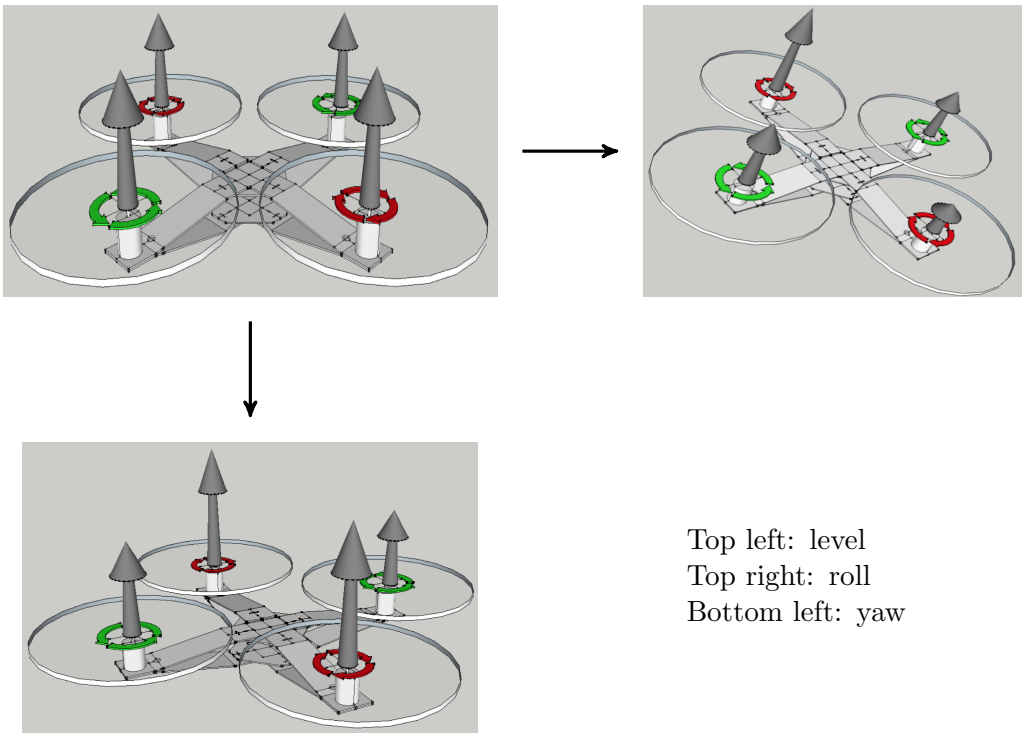
1.1 Quadcopters

Aerial robotics is an emerging field which is very exciting. The first robots were for use in industrial manufacturing, and mostly consisted of stationary variants on robotic arms. The next generation of robots were land-based and years of robotics research went into increasing their agility and purposeful movement on the 2D plane. Now it is becoming possible to make robots with the freedom to roam about all three dimensions. A popular platform for small unmanned aerial vehicles (UAVs) is the quadcopter.

A quadcopter is a device with four propellers arranged in a cross configuration. Two propellers spin clockwise and two counterclockwise, which lifts the quadcopter into the air while avoiding any net angular momentum. Motion in the x and y directions is achieved by tilting the quadcopter so that the thrust has a horizontal component. Figure 1 shows how modifying the thrust of different propellers can allow the quadcopter to move. The quadcopter platform is similar to a helicopter in its mobility, but simpler to construct and operate because the propeller blades are fixed-pitch. However, the control problem is still difficult and requires much manual tuning of controller parameters. Our project aims to explore the control of quadcopter with evolved neural networks, so that the element of human design is reduced as much as possible.

1.2 Neural Networks and NEAT

Neural networks are very versatile and in this project we attempt to apply them to the entire quadcopter control problem. That is, one neural network receives as input the real-time sensor data from the quadcopter, and its outputs directly control the motor speeds. Previous work has used a hierarchy of neural networks (see below), but we wish to demonstrate a monolithic controller where all of the structure is evolved.



Top left: level
Top right: roll
Bottom left: yaw

Figure 1: This figure shows the way in which a quadcopter can control its roll, pitch and yaw angles. In the top left, the quadcopter is level and stationary with the motors providing approximately equal thrust (note the red propellers spin clockwise and the green counterclockwise). If one of the color pairs develops a thrust differential, as in the top right, then the quadcopter will roll or pitch about the other axis. On the other hand, if one pair of motors spins faster than the other pair, as in the bottom left, then the quadcopter can control its yaw angle. Figure from presentation in [3].

The easiest method for evolving both the weights and topology of a neural network is NEAT, or NeuroEvolution of Augmenting Topologies, described in [11]. We provide the inputs, outputs and the fitness function. The NEAT library (we are using the original C++ implementation) contains everything necessary to implement a genetic algorithm, including population management, mutation, crossover, and encoding/decoding of neural networks into text-based genomes. Using bookkeeping tricks, the topology of the networks can be evolved without losing the essential network structure, so the network can complexify (i.e. add more nodes) when tweaking connection weights ceases to increase fitness.

1.3 ROS and Gazebo

This work is presented as experiments on simulated quadcopters. The simulation platform is preferred over real robots at this early stage due to the repeatability of the simulation, the ease of reconfiguration, and the absence of hardware difficulties that distract from the research goal. The authors both have previous experience using ROS, the Robot Operating System [8]. ROS "provides libraries and tools to help software developers create robot applications. It provides hardware abstraction, device drivers, libraries, visualizers, message-passing, package management,

and more." [1]. Some ROS terminology may be necessary in order to understand the architecture of this project:

- **Topic**: the infrastructure of ROS's message passing system. The message passing is a publisher-subscriber system where one node publishes messages to a topic and any number of nodes may receive them by subscribing to the topic.
- **Node**: the smallest unit of ROS software. A node usually uses topics for input and output, and encapsulates a single unit of functionality. Groups of nodes commonly form a data pipeline. For example, a mobile robot with vision might have one node which interprets motion commands to drive the motors, and a pipeline of nodes performing transformations on camera input (e.g. color segmentation, blob finding, generation of motion commands). Due to the network transparency inherent in ROS, nodes can communicate across many machines.
- **Package**: a set of nodes and possibly other programs (such as *launch files*, which are configurations of nodes that ROS can start up all at once)
- **Stack**: a set of packages. Team Hector Darmstadt distributes the `hector_quadrotor` software as several stacks.

ROS is very flexible due to the modular architecture which it forces upon programmers. Often the same nodes can be used in simulation and on real robots. To facilitate this, ROS bundles a rich physics-based robot simulator called Gazebo. Gazebo can communicate with ROS nodes and it simulates a words containing one or more robots using the ODE physics simulator.

1.4 Related Work

Quadcopters, sometimes called quadrotors or quadrotor helicopters, are currently a very popular platform for robotics research. Their small size and mechanical simplicity (i.e. compared to a helicopter or airplane, variable-pitch propeller blades and complex mechanical linkages are avoided) makes them attractive. Of course, with this simplicity comes difficulty in control, since the quadcopter platform is dynamically unstable and nonlinear [9]. In addition, the dynamics of the system are nonlinear and difficult to model. Further complicating the control problem is the number of control inputs (the four propeller speeds) is fewer than the number of degrees of freedom (linear and angular velocities), resulting in an underactuated system.

The most common solution to this control problem is to use a series of PID controllers, each concerned with one of the quadcopter's state variables: roll, pitch, yaw, altitude, position. This approach has been successful, and commercial implementations of such controllers exist, such as the KKmulticopter hardware used in [2]. However, as attested in [3], the integration and optimization of these controllers is a time-consuming and laborious process, especially in the presence of unmodeled system dynamics and sensor noise. Even PID controllers may have trouble in operating conditions that are too far from the optimal hover point [9].

Therefore, several researchers have attempted to use neural network-based controllers to avoid the problems with PID controllers. This is an attractive solution from an engineering standpoint because it requires even less hardcoded knowledge of the system than a PID controller. If successful, this would also be an impressive demonstration of the capabilities of neural networks as an implementation strategy for adaptive robots. In 2008 Nicol et al [7] tested a neural network controller against more traditional methods of developing controllers, in the face of underspecified

system parameters and simulated wind. They use a special type of neural network, the CMAC (Cerebellar Model Articulation controller) and online training. More recently, Shepherd and Tumer [9] investigate the use of NEAT to train neural network controllers, and test them on a simulated quadcopter following waypoints. This work is similar to the work presented in this paper, except that Shepherd and Tumer restrict the role of their neural network controllers in a quite strict fashion. Their controller is hierarchical, with a position controller providing desired roll, pitch, and vertical velocity. These are used to create error functions for the state variables roll, pitch, yaw and height, which are inputs to lower-level controllers that command the motors directly. The lower-level controllers are proxies for PID controllers, in that their inputs are the error function, its derivative, and its integral; additionally, learning starts by training the networks to imitate PID controllers. This approach works and Shepherd and Tumer demonstrate impressive results with respect to disturbance rejection and waypoint following, but it seems desirable to reduce the amount of the controller design that must be prescribed, and directly transferred from the PID controller (i.e. the controller hierarchy and the separate networks for each state variable). In theory, NEAT should be able to design its own controller structure, which may or may not be similar to the PID hierarchy that an engineer would design.

2 Experimental results

2.1 Experimental platform: HERP

All of the work described in this paper was performed using a quadcopter simulated using ROS, Gazebo and the quadcopter packages from Team Hector Darmstadt. We refer to this platform as HERP, the Highly Evolved Rotational Platform. The architecture of HERP and our neural network test harness is summarized in the following diagram, Figure 3. The two main components, both implemented in C++, are Overmind, the test harness which interacts with NEAT and orchestrates the neuro-evoluationary process, and Gazebo, which sits inside ROS and hosts our plugin that operates a quadcopter using a neural network controller. This plugin is but one of many plugins in the `hector_quadrotor_gazebo_plugins` (HQGP) package. The suite of plugins implements a rich simulation environment for a quadcopter, including simulated IMU and altitude sensors. Other packages from Team Hector provide for dynamical simulation and graphical display of the quadcopter in Gazebo (see Figure 2). During the evolution process, genetic material (i.e. NEAT's encoding of a neural network) flows from Overmind to Gazebo when whenever the controller needs to be refreshed, and Overmind resets the Gazebo simulation at the beginning of each test (but see below for complications). The entire architecture is tied together with numerous XML files (specifically, the `roslaunch`, URDF and XACRO formats), which are interpreted by the ROS framework.

2.2 Learning to fly (or not)

The first experiment attempts to evolve a neural network that can control all aspects of the quadcopter's motion in space. The inputs to the neural network are the quadcopter's current orientation (Euler angles), linear velocity, angular velocity, height above the ground, and the distance vector from the goal location. This totals thirteen inputs. The four outputs directly control the motor speeds. Initially, the networks in the population are fully connected with no hidden nodes and random weights. Of course, NEAT is free to evolve hidden layers and recurrence. The control task

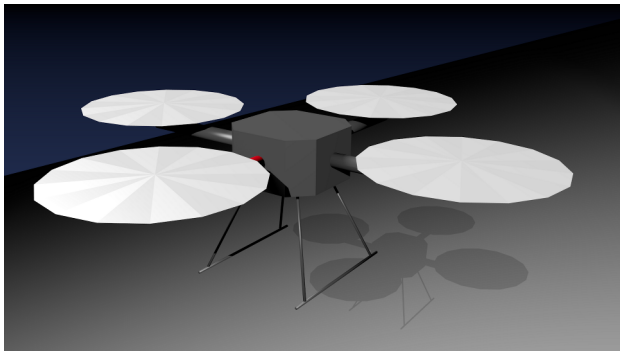


Figure 2: This is the quadcopter model provided by `hector_quadrotor` for simulation in Gazebo, dubbed "HERP" for the purposes of these experiments. Rendering from [5].

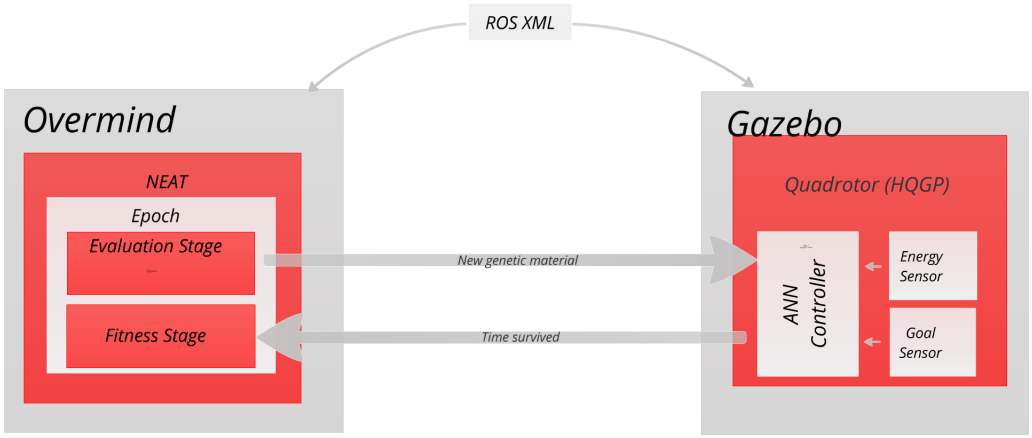


Figure 3: This diagram shows the architecture of HERP and our neural network test harness, Overmind.

requires integration and smoothly varying control, so recurrence should be an important part of the final network.

The experimental setup in this case is simple. One trial starts the quadcopter at the point $(0, 0, 0.5)$ in free space, with no obstructions. It is given an initial energy (configurable, for this experiment set to $E_0 = 10000$). The quadcopter is allowed to run until one of the following occurs:

- The time runs out. We set the maximum trial time to $t_{max} = 20$ seconds. This is measured in "simulator time", which is very nearly synchronized with wall time.
- The quadcopter strays too far from the goal location (more on goals below). We set $d_{max} = 25$ meters in early trials, but in later trials we set $d_{max} = 2500$ meters in order to effectively remove this restriction and focus on stabilization.
- The energy runs out. For all of our trials $E_{min} = 5000$, and the energy decreases at every timestep based on the motor commands. This simulates energy usage; quadcopters, like all

flying robots, need to be very energy-conscious because battery technology is often a limiting factor. If the motor command at timestep i is $M_i = \begin{bmatrix} m_1 & m_2 & m_3 & m_4 \end{bmatrix}^T$, then the energy decreases proportionally:

$$E_{i+1} = E_i - \alpha(m_1 + m_2 + m_3 + m_4) \tag{1}$$

In our experiments this proportionality constant $\alpha = 0.01$.

In addition, when the quadcopter is close enough to a goal ($d_{\text{close enough}} = 20$ meters), then it receives an energy boost ("in-flight battery recharging") of 0.02.

These numbers seem small, but the controllers run at least a hundred times per second, so the small deltas are integrated quickly.

- The simulator encounters a NaN fault. Some situations cause the physics solver to given an incorrect result, namely NaN (Not a Number). This invalidates further simulation, so the trials were halted if this occurred.

The goal and waypoint scheme was envisioned as a way to direct the quadcopter's movements once it learned to stabilize. The plan turns out to be similar to that implemented by [9]. There is one goal location in the world at a time, and the three-dimensionsial vector reaching from the quadcopter to the goal is given to the neural network as input. The initial goal is at the same location as the quadcopter, $(0, 0, 0.5)$. If a quadcopter stays near a goal for a given period of time, then the goal will move and the quadcopter will be forced to follow it in order to gain the corresponding energy boost. The moving goals were never implemented, because the quadcopters never learned to stabilize.

Since the energy bookeeping should imbue the quadcopter with intrinsic motivation to conserve energy and seek the goal location, the fitness function is extremely simple. It is simply the total time that the quadcopter survives.

Unfortunately, this experiment was unsuccessful in producing a stabilizing quadcopter controller. Figure 4 shows a typical evolution run. It is clear that the complexity of the neural network is increasing, but the fitness (survival time) is flat at nearly zero. The main cause of this seems to be bugs in the Gazebo simulator (more on that below).

2.3 Observing the experts

One promising way to begin the training of a neural network controller would be to imitate the behavior of an "expert" which can already solve the problem at some level (possibly an introductoy level). In this case, Team Hector Darmstadt includes a simple PID controller that can be used as an expert. In the second experiment, therefore, we keep the same general architecture as in the first experiment. However, the neural network controller no longer has direct control of the quadcopter (though it is still being evaluated and evolved by the Overmind). Instead, the simple PID controller is put in control. The PID controller produces a force and a torque which are applied to the quadcopter at each timestep. The neural network controller produces desired motor commands. These ouputs are related by the following equation:

$$\begin{bmatrix} F_z \\ T_x \\ T_y \\ T_z \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 0 & -1 & 0 & 1 \\ -1 & 0 & 1 & 0 \\ 0.5 & -0.5 & 0.5 & -0.5 \end{bmatrix} \begin{bmatrix} m_1 \\ m_2 \\ m_3 \\ m_4 \end{bmatrix} \tag{2}$$

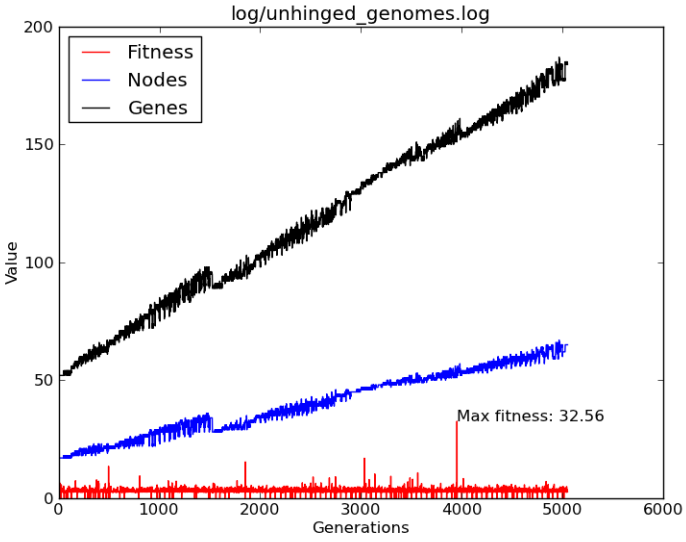


Figure 4: This plot shows an evolution run where the learning process was not successful.

Since the matrix is invertible, we can proceed as if both controllers output desired motor commands.

The experimental setup is much the same as in Experiment 1, except that the quadcopter always runs for the maximum time (20 seconds). The Overmind runs both controllers until the time runs out, perturbing the quadcopter with a simulated force and torque impulse every few seconds (this simulates buffeting wind or other disturbances). The PID controller is able to recover, and the neural network controller should learn to imitate its motor commands. As shown in the equations below, the Overmind keeps a running dot product of the controllers' time-series command vectors (if there are n timesteps, then the vectors will each have $4n$ elements) and uses the fitness function to minimize the dot product (the fitness function is simply the inverse of the dot product).

$$m_{n,i,j} = \text{HERP's motor command to motor } i \text{ at time } j \tag{3}$$

$$m_{p,i,j} = \text{PID controllers' motor command to motor } i \text{ at time } j \tag{4}$$

$$M_{net} = \begin{bmatrix} m_{n,1,0} & m_{n,2,0} & m_{n,3,0} & m_{n,4,0} & m_{n,1,1} & \dots \end{bmatrix} \tag{5}$$

$$M_{PID} = \begin{bmatrix} m_{p,1,0} & m_{p,2,0} & m_{p,3,0} & m_{p,4,0} & m_{p,1,1} & \dots \end{bmatrix} \tag{6}$$

$$fitness = (M_{net} \cdot M_{PID})^{-1} \tag{7}$$

Unfortunately, this approach was not successful in evolving a neural network controller that could imitate the simple PID controller, even though NEAT attempted to complexify. A plot of fitness vs. time is shown in Figure 5.

2.4 Challenges

2.4.1 ROS architecture

Because this project was completed in a limited amount of time, we decided to build the experiment within the ROS framework. However, this presented its own significant challenges. Because ROS

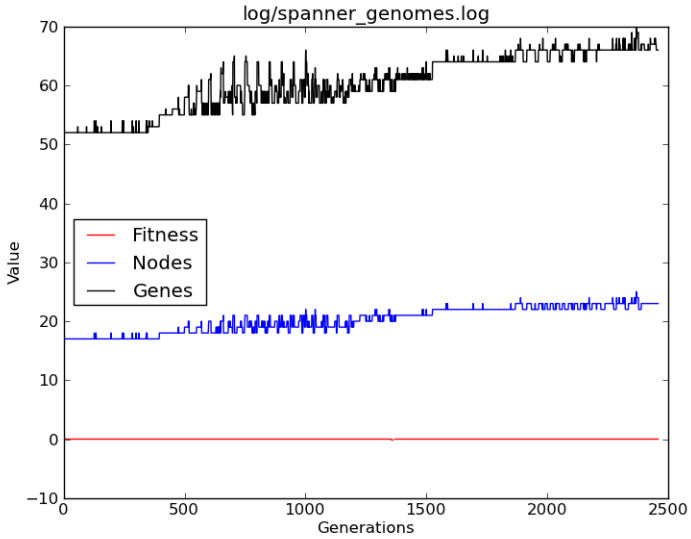


Figure 5: This plot shows an evolution run, attempting (and failing) to train the neural network controller to imitate the simple PID controller.

is so generalizable, and predicated on multiple inter-communicating subprograms, a significant amount of boilerplate code must be written to implement even a trivial task.

There is also a paucity of documentation for most of the setup code that must be written to produce a working ROS node. For example, a given node's `manifest.xml` file might look like this:

```
<package>
  <description brief="herp_msgs">

    This package provides various msg/srv files for messages/services
    passed around between Overmind and hector_quadcopter_gazebo_plugins.

  </description>
  <author>Alex Burka and Seth Foster</author>
  <license>BSD</license>
  <review status="unreviewed" notes=""/>
  <url>http://ros.org/wiki/herp_msgs</url>

  <depend package="roslib"/>
  <depend package="roscpp"/>
  <depend package="geometry_msgs"/>
  <export>
    <cpp cflags="-I${prefix}/include"/>
  </export>

```

```
  <rosbuild2>
    <depend package="roslib"/>
    <depend package="geometry_msgs"/>
    <export>
      <include_dir>${herp_msgs_SOURCE_DIR}/include</include_dir>
    </export>

    <msgs>
      msg/EnergyRemaining.msg
      msg/OvermindStatus.msg
    </msgs>
    <srvs>
      srv/GenomeWithUnnecessarilyLongTypeName.srv
    </srvs>
  </rosbuild2>
</package>
```

An example of the undocumented nature of ROS is the above file. Note the `rosbuild2` environment. There is no reason we could find to put this here aside from other `manifest.xml` files having it; however, without that environment and its duplication of code, the node will not compile.

In addition, the tool used by ROS to visualize data does not work with our machine's hardware.

2.4.2 Gazebo bugs

The primary problem with Gazebo is that for models with small masses, a strange look will cause an overflow in angular velocity which leads to most of the model's state being composed of NaN. Since any operation on NaN produces another NaN, this error will be threaded through the simulation and the trial will give no further useful data. A significant portion of our coding time was devoted to figuring out a way to avoid this situation. In our `overmind` program, we monitor the output of Gazebo for NaNs using a standard ROS message subscriber that listens to the state of the model:

```
    ros::Subscriber state_sub = nh->subscribe<gazebo_msgs::ModelStates>(state_topic,
    1000, state_callback);

    /*Code between subscriber declaration and callback elided*/

void state_callback(const gazebo_msgs::ModelStatesConstPtr& msg)
{
  for (int i=0; i<3; i++) { //hardcoded because fuck you use a vector
    geometry_msgs::Vector3 thistwist = msg->twist[i].angular;
    if(isnan(thistwist.x) || isnan(thistwist.y) || isnan(thistwist.z)){

    ROS_INFO("Stopping because nan");
    genome_log << "// NAN ENCOUNTERED" << endl;
    beam_me_up();
```

```
15
16      }
17    }
18  }
```

The above code monitors the state of the quadcopter model in Gazebo for every NaN. When one is detected, it writes to a log and terminates the trial using the `beam_me_up()` function, which removes the quadcopter from the environment, cleans up, and writes its status to NEAT. In theory, because the fitness reported to NEAT is a pure function of the time that the quadcopter individual survived, the controller over many runs ought to figure out how to behave to avoid making NaNs.

In practice, however, the controllers did not learn how to avoid this situation. We think that this is because Gazebo is so sensitive to inputs, in ways that we do not fully understand, that a model's state will turn into NaNs with little to no provocation, in unpredictable ways.

2.4.3 GAs are unpredictable and inscrutable

Another problem with our experiment is in its fundamental underpinnings. The output of a genetic algorithm is hard to make sense of as a human, and the rules governing it are dependent on a large number of parameters (in the case of our implementation of NEAT, there are 33). Because we did not have the time available to test the effects of all the different parameters, we did our experiments with the same parameters that are set in the default experiments that come packaged with NEAT C++ (available at `http://nn.cs.utexas.edu/?neat-c`). These parameters, which were designed to evolve networks to solve the double-pole balancing problem, may not be suited to the particulars of the problem we are trying to solve.

3 Conclusions and Future Work

Though our experiments failed, we believe that there is nothing intrinsically wrong with our experimental premise. Because of the complexifying behavior of NEAT, it (or its HyperNEAT cousin) is uniquely suited to provide an adaptive algorithm for the aerial robot control problem.

3.1 Future work

There are many directions for future work on this project, and it is hoped that future researchers will iron out the simulation problems and make meaningful progress on the problem of controlling a quadcopter entirely with a neural network.

To this end, the immediate future work would be to fix our experiment by preventing Gazebo from malfunctioning so as to produce so many NaN results, or to find a way to reliably reset the Gazebo simulation so that there are no artifacts (if the simulation can be completely reset, then the quadcopters can be punished for causing NaN errors and presumably that delinquent behavior will quickly evolve away).

Once the simulation is reliable, the possibilities are unlimited. There are several approaches which we have in mind that could aid in developing a successful neural network controller. Since the search space is unfathomably large and the vast majority of controllers for a flying robot will not be stable, it may be advantageous to use a non-evolutionary method to start with a base neural

network controller which can stabilize acceptably. In Experiment 2, we tried to train a neural network controller to this base level of competency by using NEAT and a fitness function which checked accuracy relative to a PID controller, but using a fixed topology and back-propagation for training might be more successful (with the intention of switching to NEAT once the controller can stabilize).

Another possibility is to structure the controller more, as in **??**. It may be that the search space is just too large for us to reasonably expect NEAT to find a solution before the heat death of the universe. We could restrict NEAT's role in the controller by providing some overall structure (for example, separate networks for high-level navigation and low-level stabilization).

Of course, the simulator is never the last word on a robotics project, though it is indispensable for preliminary work such as this. Therefore, once a simulation approach is successful at evolving a competent quadcopter controller, experiments with real robots should proceed.

Acknowledgments

This project would not have been possible without the support of many people and various pieces of software. Some of these are listed below.

- **Lisa Meeden,** for teaching us about neural networks, genetic algorithms, NEAT and other adaptive robotics techniques
- **Team Hector Darmstadt,** for publishing the excellent `hector_quadrotor` stack (and related stacks)
- **ROS,** the underlying robotics framework and simulator

References

[1] Documentation - ros wiki. `http://www.ros.org/wiki/`.

[2] Matthew Bowers. *E90: Fan Powered Drone*. Swarthmore College, May 2012.

[3] Alex Burka and David Saltzman. *E90: Snitchcopter*. Swarthmore College, May 2012.

[4] Team Hector Darmstadt. hector_quadrotor - ros wiki. `http://mirror.umd.edu/roswiki/hector_quadrotor.html`.

[5] Team Hector Darmstadt. hector_quadrotor_urdf - ros wiki. `http://mirror.umd.edu/roswiki/hector_quadrotor_urdf.html`.

[6] Travis Dierks and Sarangapani Jagannathan. Output feedback control of a quadrotor uav using neural networks. *Trans. Neur. Netw.*, 21(1):50–66, January 2010.

[7] C Nicol and C J B Macnab. Robust neural network control of a quadrotor helicopter schulich school of engineering , university of calgary department of electrical and computer engineering department of mechanical and manufacturing engineering. *Control*, 130(1):1233–1238, 2008.

[8] Morgan Quigley, Ken Conley, Brian P. Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y. Ng. Ros: an open-source robot operating system. In *ICRA Workshop on Open Source Software*, 2009.

[9] Jack F. Shepherd, III and Kagan Tumer. Robust neuro-control for a micro quadrotor. In *Proceedings of the 12th annual conference on Genetic and evolutionary computation*, GECCO '10, pages 1131–1138, New York, NY, USA, 2010. ACM.

[10] J.F. Shepherd. *A Hierarchical Neuro-evolutionary Approach to Small Quadrotor Control*. Oregon State University, 2010.

[11] Kenneth O. Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary Computation*, 10(2):99–127, 2002.