University of Sussex

A Few Antecedents of Accelerating Evolution by Morphological Change

How Does Morphological Change Accelerate Evolution?

Shane Celis

Submitted for the degree of Master of Science
University of Sussex
September 2011

Declaration

I hereby declare that this thesis has not been and will not be submitted in whole or in part to another University for the award of any other degree.

Signature:

Shane Celis

UNIVERSITY OF SUSSEX

Shane Celis, Master of Science

A Few Antecedents of Accelerating Evolution by Morphological Change
How Does Morphological Change Accelerate Evolution?

Acknowledgements

I would like to thank my advisor Dr Luc Berthouze for helping me figure out how to let my question be my guide. Thanks to Dr Greg Hornby for taking me on as an intern and research assistant. Thanks to Bob De Caux for the maths and the gym. Thanks to Markus Echterhoff and Jay Kannan for the video games and never failing to catch a nerdy internet reference. Thanks to Kostas Kafkalas, Rowan Dent, and Sophie Burkhardt for rarely letting me take a bus alone. Thank you to Sarah Barber for being awesome and helping me keep my spirits up. And a very special thanks for Nykia Hunter for helping me finish this dissertation while moving across the country—I could not have done it without you!

Contents

List of Tables

List of Figures

Chapter 1

Introduction

An animal's body grows and changes over its lifetime, yet it can retain and improve its behavioural skills. Likewise the form of a species can evolve into new forms that are fit enough to reproduce. Robustness in face of such changes on either time scale is impressive especially when compared with the lack of robustness in robots. Robots in controlled, structured environments can be fashioned to perform some wonderful tasks. However, unstructured environments pose large problems, and unstructured bodies pose an even greater challenge [5].

Morphological change has been demonstrated to accelerate evolution of robust behaviour in one instance [3]. However, it is unclear exactly how or why this happens. It is thought that the initial morphological form may serve as a scaffolding for the following form [10]. It may be that morphological variation merely adds noise to the simulation such that individuals that rely on fragile sensitivities are excised from the population[8].

Bongard showed the evolution of light following behaviour was accelerated for robots that grew from a leg-less anguilliform to a legged hexapod when compared to evolving a hexapod with no morphological change[3]. My project is a critical replication of that experiment using a different robot platform and aim. This project evolves a robot with varying degrees of conservation between the earlier and later forms to help answer the question, under what conditions does morphological change accelerate evolution?

In Bongard's experiment the infant form is conserved entirely in the adult form. The infant is a leg-less animal that becomes the adult spine and continues to assist the adult's mobility. Currently, there is no principled or direct way of choosing how one ought to change the morphology of a robot to try and acquire this evolutionary acceleration and robustness. Intuition suggests that conservation of the infant form is probably a good thing. It may stagger the problem of finding a controller by tackling a subset of the problem and building upon that solution. However, cases are bound to crop up that defy one's intuitions and expectations. I suggest that Bongard's experiment [3] was just such a case.

The aim of this project is to consider a robot morphology where the conservation between the infant and adult form may be fruitfully compared. My hypothesis is that the morphological change that conserves the infant form may achieve an evolutionary acceleration and the non-conservative morphological change will not be accelerated. In the event that the non-conservative morphological change is accelerated, one must account for this somehow. A feasible mechanism could be that although the infant morphology is not conserved, part of the infant controller is conserved and that could explain the evolutionary acceleration. To that end, a variation of the controllers is considered.

Chapter 2

Method

Notation Conventions

The following notation conventions are used: Lower case symbols denote a scalar quantity *e.g.*, s implies $s \in \mathbb{R}$. Bold lower case symbols denote a vector quantity *e.g.*, $\mathbf{v}$ implies $\mathbf{v} \in \mathbb{R}^n$. Bold upper case symbols denote a matrix quantity *e.g.*, $\mathbf{M}$ implies $\mathbf{M} \in \mathbb{R}^{m \times n}$. Vectors with a circumflex, or “hat”, denote a unit vector *e.g.*, $\hat{\mathbf{v}}$ implies $||\hat{\mathbf{v}}|| = 1$. Angle brackets denote the mean of the variable $< x >$. Statistical significance level is denoted with a series of asterisks: 0.05 (*), 0.01 (**), and 0.001 (***).

2.1 Overview

This experiment uses a two-dimensional, aquatic-like environment to determine what kinds of morphological changes may accelerate evolution. The morphological forms—inspired by frog metamorphosis—have been selected such that the conservation of the infant form to the adult form may be varied. Figure 2.1 shows the two principle forms which may be parametrically varied by two variables tail length and foot length $l_t, l_f \in [0, 1]$ respectively. The full range of morphological change will be described in detail in section 2.5. In the inspiring case, the individual begins as a “tadpole” bearing only a tail. It transforms into a “frog” with four limbs. Its task is to swim to a target.

2.2 Physics Model

This experiment uses the following physical model to simulate an individual in a two-dimensional aquatic-like environment. The aim of the simulation is to provide an aquatic environment, but it is not intended to provide a realistic environment such that a controller

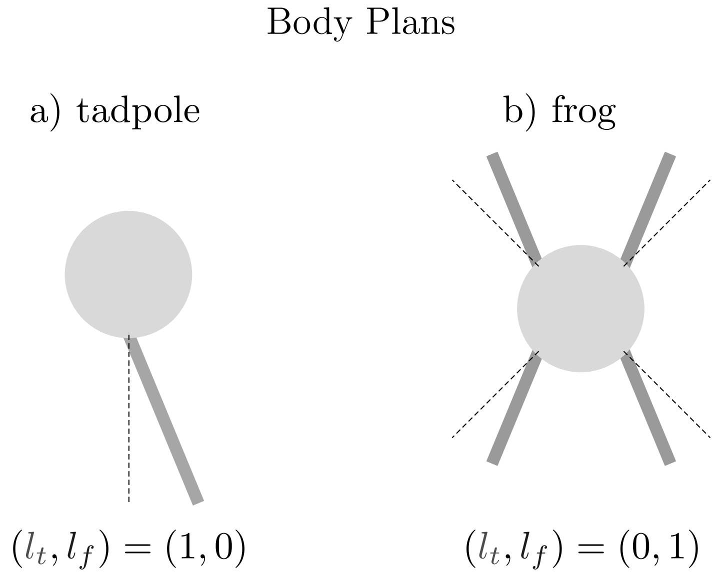

Figure 2.1: The body plans are parameterised by tail length l_t and foot length l_f. a) represents the infant "tadpole" form, and b) represents the adult "frog" form.

evolved in simulation could be easily transferred to a real robot. It is thought, however, that applying the same method with a real robot would produce comparable results.

The virtual robot is composed of six rigid bodies: one central body, one tail segment, and four feet segments. The tail and feet are connected to the central body by pinwheel joints. Eight configuration variables $\{q_1, q_2, \ldots, q_8\}$ describe the body as shown in Figure 2.2. The position of the body is denoted by the vector (q_1, q_2). The angle of the central body measured counter-clockwise to the $\mathbf{\hat{n}_2}$ axis is denoted by q_3. The angle of the tail and four feet are denoted by $q_4, \ldots, q_8$, respectively. Eight corresponding motion variables $\{u_1, u_2, \ldots, u_8\}$ describe the generalised speeds of the body $u_i = \frac{dq_i}{dt}$.

2.2.1 Simulating an Aquatic-like Environment

For each limb a drag force $\mathbf{F}_D$ opposes its direction of motion, which is given by Equation 2.5 where ρ is the density of the fluid, c_d is the drag coefficient, l is the length of the limb, w is the width of the limb, $\mathbf{\hat{n}}$ is the normal vector of the limb, $\mathbf{v}_b$ is the velocity of the center of mass of the limb, $\mathbf{v}_c$ is the velocity of the current, $\mathbf{v}$ is the relative velocity of the limb with respect to the current, A is the reference area—an orthographic projection of the limb shape on a plane perpendicular to the direction of motion. Figure 2.3 shows these values for a limb. The shape of the limb is taken to be a rod of length l, width w, and depth d. However, for the purposes of computing the drag force, the width of the limb w is set to zero since $w \ll l$ and the force it might contribute is not considered significant.

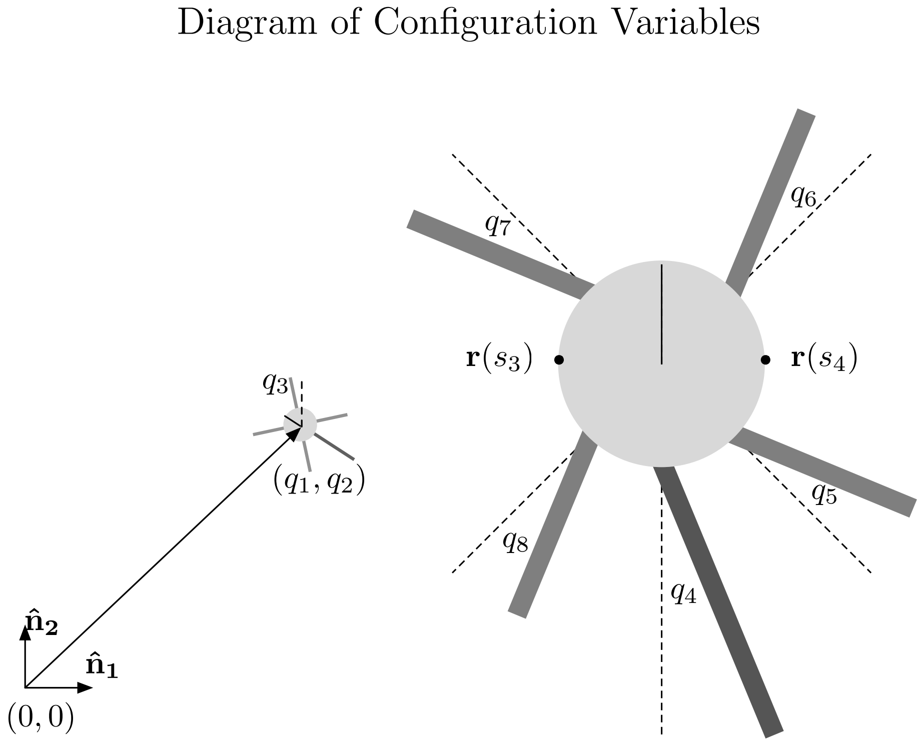

Figure 2.2: Diagram of the configuration variables $\{q_1, q_2, \ldots, q_8\}$ that fully describe the physical state of the body at time t. The position vector $\mathbf{r}(s_i)$ is the location of sensor s_i.

$$A = l\,d\,|\hat{\mathbf{v}} \cdot \hat{\mathbf{n}}| + l\,w\,|\hat{\mathbf{v}} \times \hat{\mathbf{n}}| \tag{2.1}$$

$$w = 0 \tag{2.2}$$

$$\mathbf{v} = \mathbf{v}_b - \mathbf{v}_c \tag{2.3}$$

$$\mathbf{F}_D = -\frac{1}{2}\rho\, c_d\, ||\mathbf{v}||^2\, A\, \hat{\mathbf{v}} \tag{2.4}$$

$$\mathbf{F}_D = -\frac{1}{2}\rho\, c_d\, l\, d\, |\mathbf{v} \cdot \hat{\mathbf{n}}|\mathbf{v} \tag{2.5}$$

In addition, a drag force and drag torque are exerted on the central body. The full equations of motion are given in the Appendix A.1.1.

2.2.2 Collisions

Inter-body collisions are permitted among the limbs, which may freely move through one another.[1] However, the limbs are constrained to not penetrate the central body. When the angle of a limb reaches $|q| = \frac{\pi}{2}$, a penalty torque $T_c(q)$ opposes further motion as shown in Equation 2.6.

---

[1] This is not thought objectionable because one can imagine constructing a robot with limbs arranged on planes such that the limbs could pass each other unobstructed.

Diagram of Drag Force

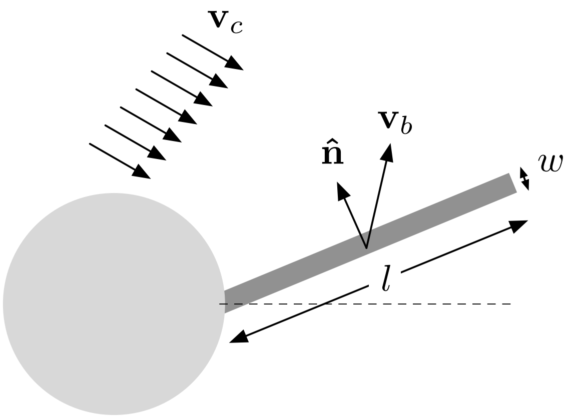

Figure 2.3: The independent variables that determine the drag force $\mathbf{F}_D$ are the length l, width w, depth d (not shown), velocity of limb $\mathbf{v}_b$, velocity of current $\mathbf{v}_c$, and normal vector $\hat{\mathbf{n}}$.

$$T_c(q_i) = T_{max}\ \text{bound}(q_i, (\frac{-\pi}{2}, \frac{\pi}{2}))\ \text{for}\ i \in [4, 8] \tag{2.6}$$

$$\text{bound}(x, (a, b)) = \begin{cases} -1 & a \geq x \wedge b > x \\ 1 & a < x \wedge b \leq x \\ 0 & \text{otherwise} \end{cases} \tag{2.7}$$

2.3 Controller

The controller used for the robot is a Continuous Time Recurrent Neural Network (CTRNN)[1]. The dynamics of a neuron y_i is given by Equation 2.8 with time constant $\tau_i \in [0.1, 100]$, weights $w_{ji} \in [-4, 4]$, bias $\theta_i \in [-2, 2]$, sensors $s_j \in \mathbb{R}$, and sensor weights $n_{ji} \in [-4, 4]$.

$$\tau_i \frac{dy_i}{dt} = -y_i + \sum_{j=1}^{m} w_{ji}\sigma(y_j - \theta_i) + \sum_{j=1}^{s} n_{ji}s_j\ \text{for}\ i \in [1, 5] \tag{2.8}$$

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{2.9}$$

Table 2.1 describes the sensors. A range finder for a target is given by the s_3 and s_4 sensors. Proprioceptive sensors are given by the $s_5, s_6, \ldots, s_{14}$ sensors. Five motor neurons are used with weighted inputs from all sensors. Each neuron exerts a torque on an associated limb. The torque for each limb $T(q_i)$ is given in Equation 2.10.

Sensor Variable	Value	Description
s_1	$\|\|(u_1, u_2) - \mathbf{v}_c\|\|$	relative translational speed
s_2	u_3	angular speed
s_3	$\|\|\mathbf{r}(s_3) - \mathbf{r}(target)\|\|$	distance to target from left sensor
s_4	$\|\|\mathbf{r}(s_4) - \mathbf{r}(target)\|\|$	distance to target from right sensor
s_5	q_4	position of tail
s_6	u_4	speed of tail
s_{7+2i}	q_{5+i}	position of each foot $i \in [0, 3]$
s_{8+2i}	u_{5+i}	speed of each foot $i \in [0, 3]$

Table 2.1: Description of available sensors

$$T(q_{i+3}) = T_{max}\ \text{clip}(y_i) + T_c(q_{i+3}) \text{ for } i \in [1, 5] \tag{2.10}$$

$$\text{clip}(x) = \begin{cases} 1 & x > 1 \\ -1 & x < -1 \\ x & \text{otherwise} \end{cases} \tag{2.11}$$

2.4 Representation: Genetic Encoding

The CTRNN parameters are specified by a real vector gene $\mathbf{g} \in [0, 1]^{105}$. Each gene component g_k is associated with one and only one of the CTRNN parameters τ_i, w_{ji}, θ_i, and n_{ji}. The w_{ji}, θ_i, and n_{ji} parameters are linearly mapped from the domain of the gene $[0, 1]$ to the domain of each parameter. The τ parameter uses a non-linear mapping $\tau_i = 10^{-2+4g_k}$.

2.5 Morphological Change

Morphological change is considered over phylogenetic and ontogenetic time. Two adult forms are evolved: A) frog with a tail $(l_t, l_f) = (1, 1)$. B) frog without a tail $(l_t, l_f) = (0, 1)$. The control case is no morphological change denoted An and Bn. The first experimental cases concern phylogenetic change denoted Ap and Bp, which are divided into phases $\{p_i\}$. The second experimental case concerns ontogenetic change denoted Ao and Bo. Figures 2.4 and 2.5 shows the tail length l_t and feet length l_f for each experimental case.

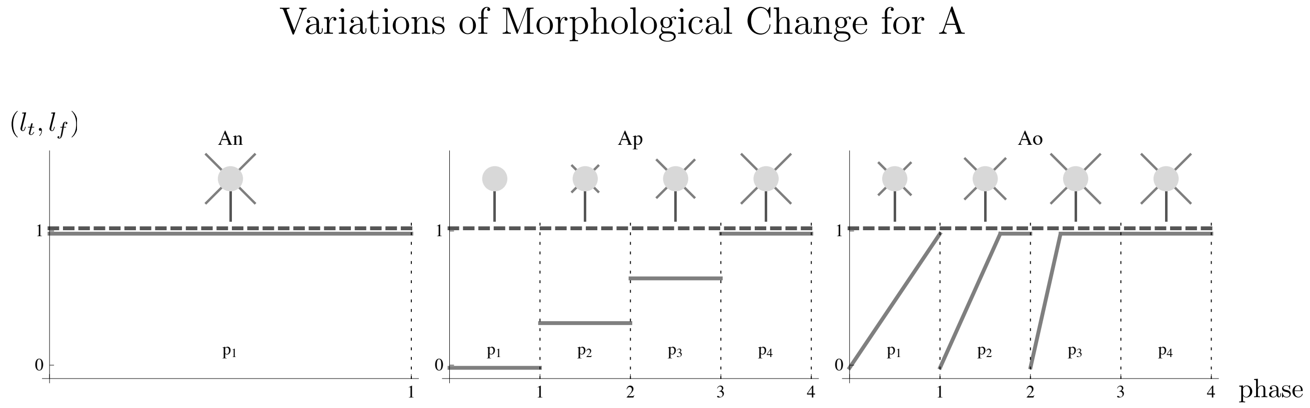

Figure 2.4: Shows how the morphology changes for each phase p_i for adult forms A where the infant form is conserved. An does not change its morphology. Ap changes its morphology over phylogenetic time. Ao changes its morphology over ontogenetic and phylogenetic time.

The overarching concern in choosing how the morphology would change was that one set of cases would conserve the infant form in the adult form A, and another case where it was not B. One nice aspect of these cases is that the last phase of Ap, Ao, Bp, and Bo are directly comparable to An and Bn respectively. Because the morphological settings are the same, one can determine whether evolution has actually gained an advantage by going through the preceding phases or not. Despite those advantages, the choice of how the morphology ought to change still has a lot of free parameters that were chosen based on intuition and symmetry.

2.6 Controller Variation

In the test cases Bp and Bo the infant form is not conserved in the adult form. However, the controller may conserve some behaviour acquired in the infant form that is useful in the adult form, which may accelerate evolution. To determine whether this happens, two types of CTRNN controllers are considered: 1) A "lobotomised" controller, which has two independent CTRNNs, one for the tail and one for the feet. 2) A "non-lobotomised" controller, which has a fully connected CTRNN that controls both the tail and feet. Both CTRNN types are shown in Figure 2.6.

The sensors are altered for the "lobotomised" controller. The tail brain does not receive proprioceptive sensors from the other limbs $\{s_7, s_8, \ldots, s_{14}\}$. Likewise, the foot brain does not receive proprioceptive sensors from the tail s_5 and s_6. Otherwise, the sensors are the

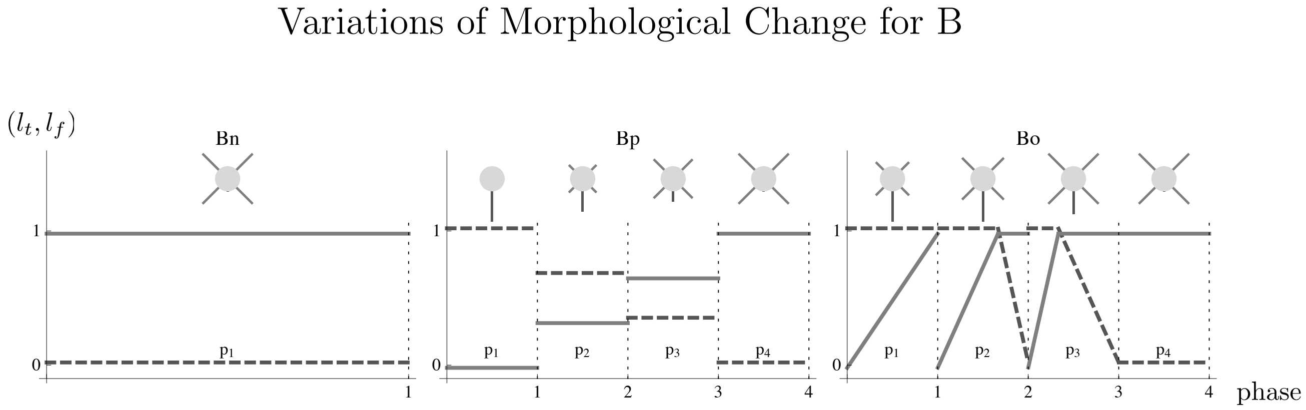

Figure 2.5: Shows how the morphology changes for each phase p_i for adult forms B where the infant form is *not* conserved. Bn does not change its morphology. Bp changes its morphology over phylogenetic time. Bo changes its morphology over ontogenetic and phylogenetic time.

same.[2]

2.7 Fitness Function

The fitness function f_i returns the mean of the sensor value normalised by the target distance. The sensor detects its distance from the target. The sensors are located on the left and right side of the central body as shown in Figure 2.2. Assuming an individual starts at the origin, the initial value of s_3 is close to the target distance $||\mathbf{r}(target)||$ hence f_i is close to one. As the sensor approaches the target f_i approaches zero. Categorising this fitness function using Nolfi and Floreano's terminology, it is a Behavioural, Explicit and Internal (BEI) according to [11]. It rates the behaviour not the function. The fitness is computed explicitly from a set of independent variables as shown in Equation 2.12 rather than an implicit measure such as a coevolutionary algorithm might use. The fitness function is internal since the information is available to sensors on the machine rather than that information being granted by fiat in a simulation or an external entity were it a real robot.

---

[2] Note on implementation: the "lobotomised" controller code is the same as the "non-lobotomised" with a specific set of weights w_{ji} and sensor coefficients n_{ji} set to zero.

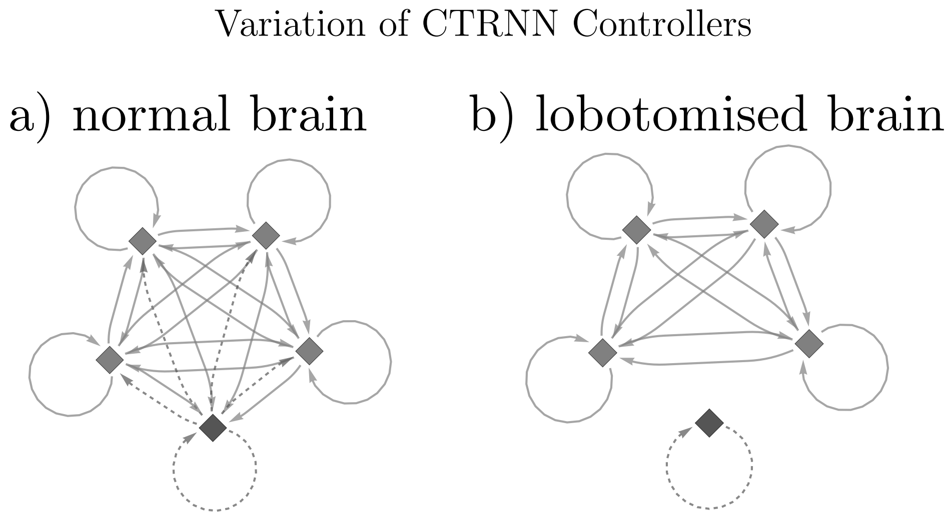

Figure 2.6: a) Fully connected CTRNN controller. b) Independent CTRNN controllers for the tail and legs. The dashed lines represent the connections from the tail. The solid lines represent connections from or to a motor neuron associated with a foot.

$$\begin{aligned} f_1 &= \frac{< s_3 >}{||\mathbf{r}(target)||} = \frac{||\mathbf{r}(s_3) - \mathbf{r}(target)||}{||\mathbf{r}(target)||} \qquad (2.12) \\ f_2 &= \frac{< s_4 >}{||\mathbf{r}(target)||} = \frac{||\mathbf{r}(s_4) - \mathbf{r}(target)||}{||\mathbf{r}(target)||} \qquad (2.13) \\ & \qquad (2.14) \end{aligned}$$

2.8 Tasks

To confirm the results are not spurious or a special case for one particular task, multiple tasks of varying difficulty are considered. The basic task is locomotion to a target. Changing the target location was considered, but it is hard to ascertain what positions for the target would be more difficult. The infant “tadpole” form with its tail directed to the south and a target to the west have to turn before it could move toward the target, so it may be more difficult for the infant form. However, the adult “frog” form is symmetric with respect to targets placed in the cardinal directions, so there is no discernible difference in difficulty. Changing the position of the target does not provide a simple means of constructing more difficult tasks.

Instead of changing the target location, varying the velocity of current $\mathbf{v}_c$ is considered. Each individual is affected similarly by the current–regardless of its morphology.[3] In task 1 the current assists the individual to the target. In task 2 there is no current. In task 3 the current pulls the individual laterally away from the target. In task 4 the current is

---

[3] This differs from Bongard’s work where the task was easier for the infant form due to its morphology.

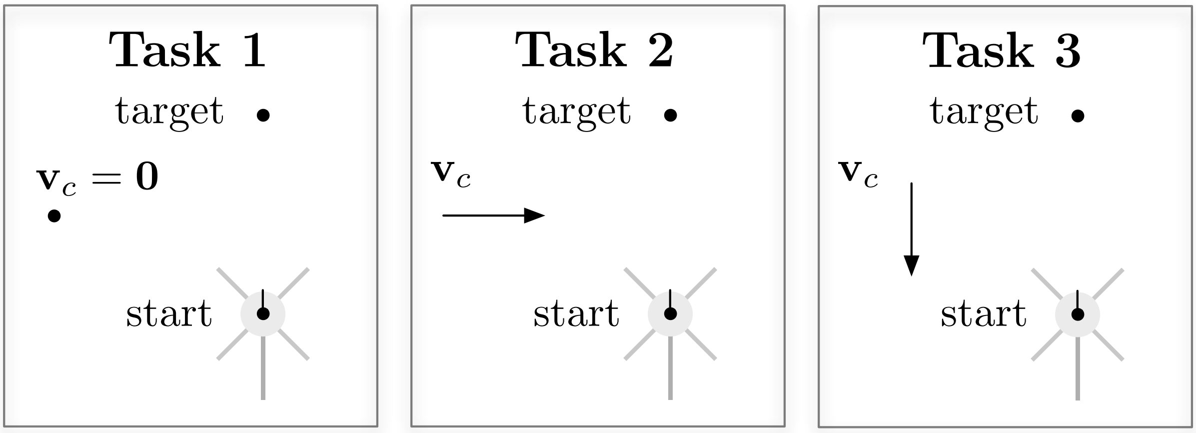

Figure 2.7: Tasks shown in an ascending order of difficulty.

directly against the target. Figure 2.7 shows a diagram of the tasks. The tasks are meant to be in an ascending order of difficulty.[4]

2.9 Evolutionary Algorithm

The evolutionary algorithm used is described in detail in [4]. The algorithm is a variant of the steady state Age-Layered Population Structure (ALPS) [7, 6]. The population is divided into layers based on the age of the individuals. The bottom layer holds the youngest individuals and is periodically reset with new genetic material. The top layer holds the oldest individuals. By segregating individuals based on age, ALPS maintains population diversity and avoids premature convergence to local optima[6].

Each individual has an age and each layer has an age limit. An individual whose age is greater than this age limit is defined to be too old. It may dislodge another individual j in the next layer if its fitness $\mathbf{f}_i$ dominates $\mathbf{f}_j$[5]. If it cannot dislodge any individuals, it is discarded.

$$\mathbf{a} \text{ dominates } \mathbf{b} \iff a_i < b_i \; \forall i \tag{2.15}$$

An individual is dominated if any other individual in its layer dominates it. In this ALPS variant, only non-dominated individuals within the layer are allowed to reproduce.

---

[4] One could argue that the task 1 may in fact be more difficult because it may require two skills: go and stop.

[5] Note: in this case lower values for fitness are considered better.

Reproduction happens as follows: A copy of the parent is made. Each element of its genome has a 0.05 chance of being reset to a random uniform value in the interval $[0, 1]$. No crossover operation is used.

Chapter 3

Results and Discussion

A run can be described by three pieces of information: The morphological variation $\{An, Bn, Ap, Bp, Ao, Bo\}$, the task $\{1, 2, 3\}$, and whether it is lobotomised $\{0, 1\}$. A trial is usually comprised of 6 runs–one for each morphological variation–and it is described by the task and lobotomised state. These are the typical values for a run:

All runs were performed on an Amazon High-CPU Extra Large Instance that has 8 virtual cores with 2.5 EC2 Compute Units each.

3.1 Results for Task 1

Figure 3.1 shows the results for task 1. Statistically significant differences were found. However, the differences indicate that it takes longer for evolution to find a solution when the morphology changes phylogenetically or ontogenetically. This may be a reasonable expectation since one is asking for the optimisation procedure to effectively solve multiple

Name	Value
Layer size	15
Population per layer	10
Reset frequency	1 reset/300 steps
Max time for evolution	20 minutes
Time per evaluation	10 simulated seconds
Target distance	0.25 m
Current speed	0.01 m/s

Table 3.1: Typical parameters used for the evolutionary algorithm

problems in succession instead of one problem. One of the aims of this work was to produce results similar to those found in [2], and attempt to discern clearer boundaries between what kind of morphological change accelerate evolution and what kind do not. Still it may be instructive to determine where all the evaluations were spent.

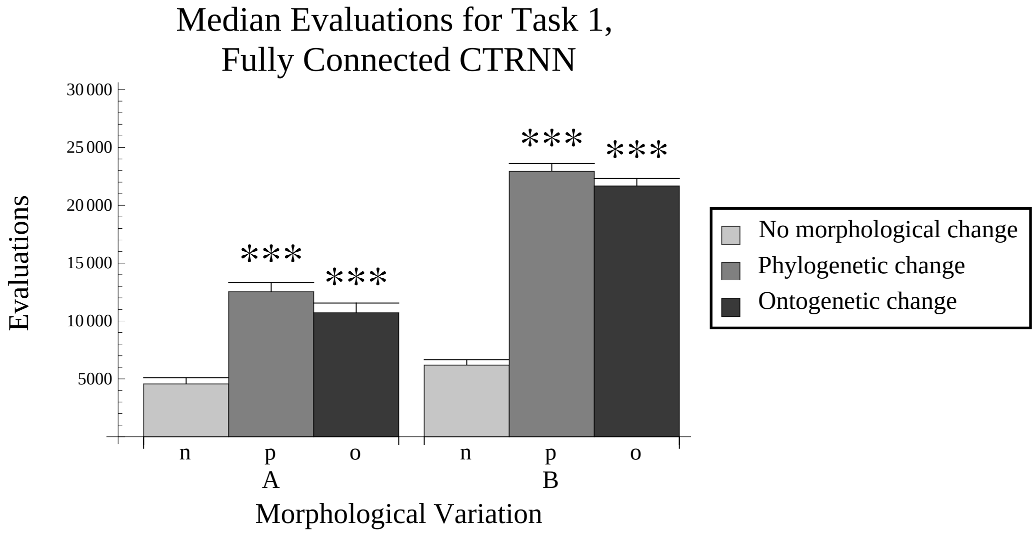

Figure 3.1: This chart shows the median number of evaluations over 100 independent trials (100× 6 runs) for each of the morphological variations on task 1 with a fully connected controller. The bar represents the standard error. The stars indicate whether the difference in median (*e.g.*, Ap, Ao) is statistically significant compared to the control (*e.g.*, An) according to the Mann–Whitney U test.

Figure 3.2 shows the mean number of evaluations per phase for the same set of results shown in Figure 3.1. Phase 1 and 2 both complete well before the only phase in An and Bn, the baseline each are compared against. Phase 3, however, takes a majority of the evaluations for most of the morphological variations. Why is that? Examining the most extreme case Bp may be instructive.

Figure 2.5 shows that phase 3 of Bp changes both the tail l_t and the feet l_f, essentially swapping the values. Perhaps the transition from the tail being the main source of locomotive power to the feet is the cause. It could be that altering both variables simultaneously is not conducive to the kind of scaffolding that may be required to exhibit an acceleration of evolution. This conjecture is supported by examining phase 3 of Ap which only alters the foot length l_f and does not require as many evaluations.

One further curiosity to note about Figure 3.2 is that phase 4 for Ao and Bo both require a fair amount of evaluations for what looks like a comparatively small change morphologically. For the Bo case phase 4 appears to take as long as Bn, which suggests that the preceding phases have not accelerated evolution at all. In fact, performing a Mann–

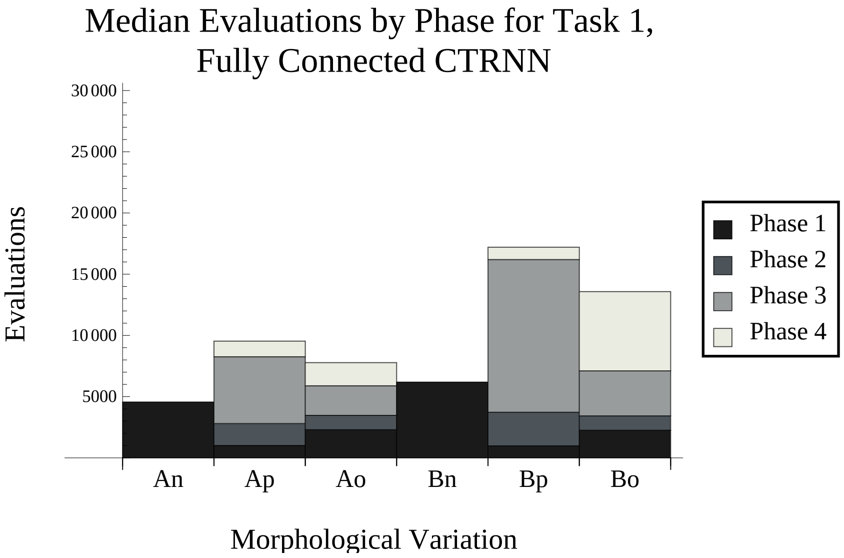

Figure 3.2: This chart shows the mean number of evaluations per phase over 100 independent trials (100 × 6 runs) for each of the morphological variations on task 1 with a fully connected controller.

Whitney U test on the two sets of data reveals that they do not statistically differ: the median evaluations for phase 1 of Bn and phase 4 of Bo are 6186.5 and 6468.5 ($p = 0.74$). Therefore, the morphological variation Bo represents a case where the conditions assuredly do not accelerate evolution.

Phase 4 of Bo may be instructive in trying to determine what conditions are necessary to accelerate evolution. It suggests a good experiment to determine whether any acceleration is happening: run each phase independent of the others with a random population and compare median evaluations, *e.g.*, phase 1 of Bn and phase 4 of Bo can be compared directly in this way. The last phase of all the morphological variations are comparable in this way. Ap and Ao statistically differ from An, and Bn differs from Bp according to the Mann–Whitney U test.

Examining phase 4 of Ap and Bp in isolation demonstrates two cases where the populations are primed to succeed by the preceding phases. Granted the preceding phases, especially phase 3, have made those gains not worthwhile cumulatively, but they may be instructive yet. One oddity is comparing Ap which only alters l_f and Bp which alters both l_f and l_t yet Bp takes less time than Ap ($p = 8.1 \times 10^{-3}$), so the simultaneous changing of variables need not cause automatic concern.

3.2 Results for Task 2

Task 2 was run similarly to task 1 with the exception that fewer trails were conducted. Because the magnitudes are so different between the control and the experimental group, it does not require many trials to determine a statistical difference in the distributions.

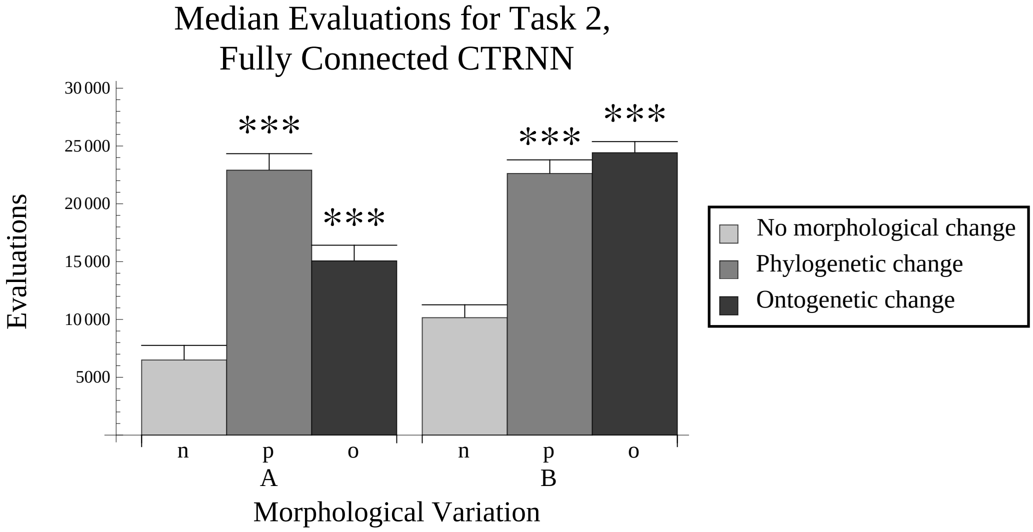

Figure 3.3: This chart shows the mean number of evaluations over 32 independent trials (32 * 6 runs) for each of the morphological variations on task 2 with a fully connected controller. The bar represents the standard error. The stars indicate whether the difference in mean between the experimental (*e.g.*, Ap, Ao) is a statistically significant compared to the control (*e.g.*, An) according to a Mann–Whitney U test.

The results for task 2 do not look dramatically different from task 1. One gratifying aspect is that task 2 does appear to be more difficult than task 1 as intended. The median evaluations for An task 1 and 2 are 4562.5 and 6495; the distributions do differ (Mann-Whitney U $p = 0.0055 < 0.01$).

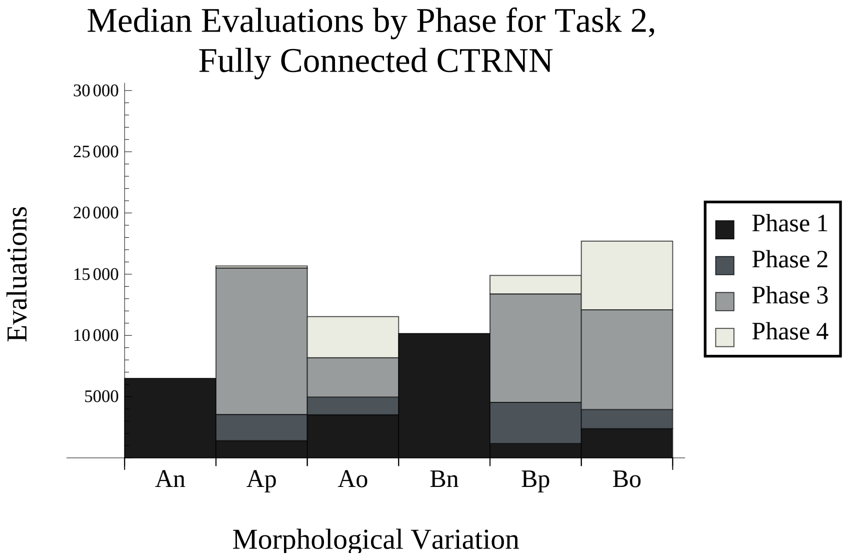

Figure 3.4: This chart shows the mean number of evaluations per phase over 32 independent trials (32 * 6 runs) for each of the morphological variations on task 1 with a fully connected controller.

3.3 Results for Task 3

Task 3 was presumed to be the most difficult of the three tasks. It proved to be so. Figure 3.5 shows the results. Most runs do not make it out of phase 1. However, a few runs do reach phase two, namely Ap and Bp, which shows that the task was not utterly impossible. The magnitude of the current $\mathbf{v}_c$ ought to be reduced such that the task can be more readily reached.

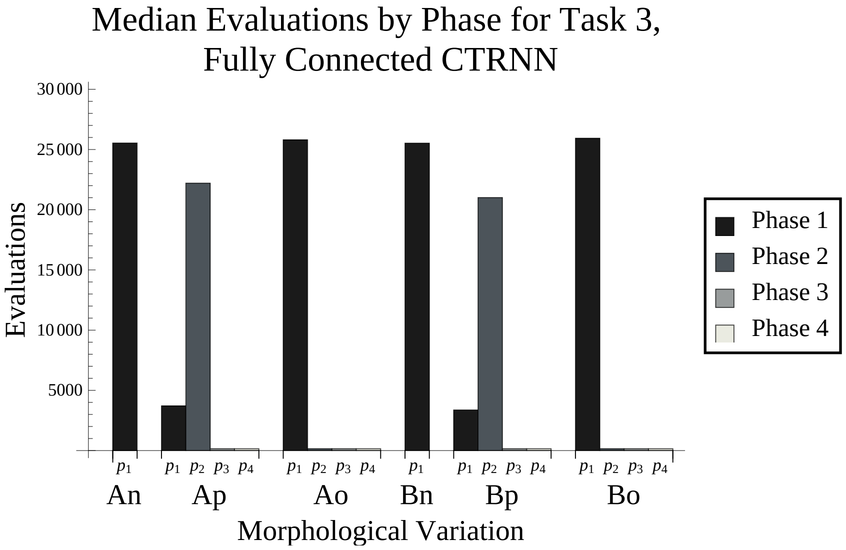

Figure 3.5: This chart shows the mean number of evaluations per phase over 20 independent trials (20 * 6 runs) for each of the morphological variations on task 1 with a fully connected controller.

3.4 Follow Up Experiments and Results

No lobotomised runs were conducted since it was not required to determine whether evolutionary acceleration happened on account of the controllers.

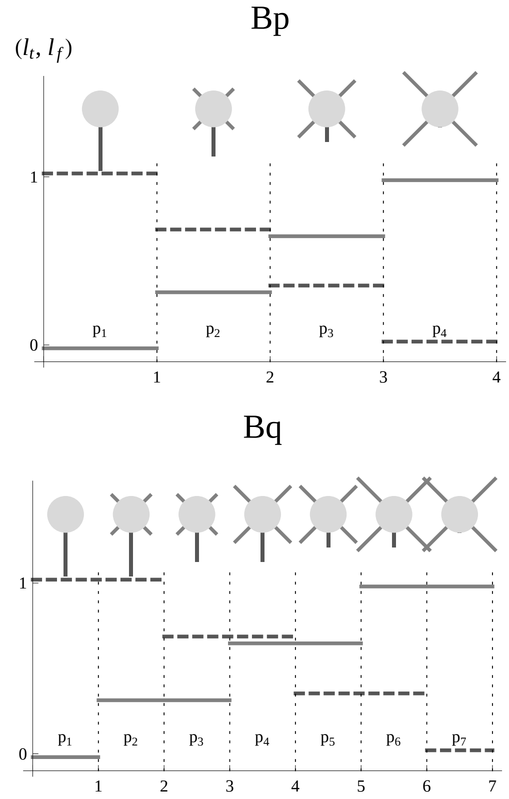

Figure 3.6: This new morphological variation Bq is proposed to help determine why phase 3 of Bp appears to do so poorly.

Because changing the tail and foot length at the same time may be why phase 3 of Bp in task 1 does so poorly, a variation of it called Bq was created that only changes one length at a time but is otherwise very similar. Bq is shown in Figure 3.6.

Figure 3.7 shows that even by only altering one part of the morphology at a time, the spike in evaluations in phase 3 was not really dispatched. Phase 4 of Bq does show that the increase in foot length—not the decrease in tail length—is responsible for the spike in evaluations.

3.5 Conclusion

I have replicated the evolutionary algorithm in [3] and applied it to a new robot platform and environment, attempting to ascertain what kinds of morphological change accelerate evolution. I had hoped to setup my experiments to find a boundary where some experi-

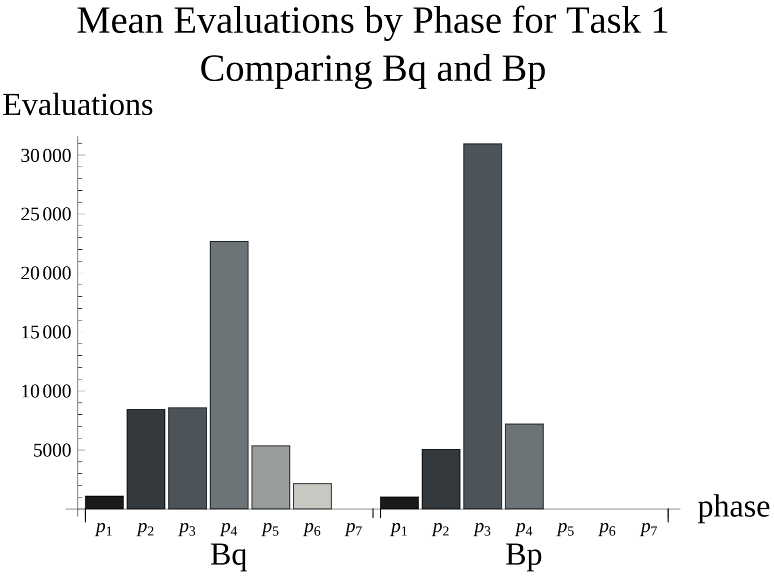

Figure 3.7: The new morphological variation Bq is compared with Bp. This chart shows the mean number of evaluations over 20 independent trials (20 × 6 runs) for each of the morphological variations on task 1 with a fully connected controller.

ments exhibited the acceleration and some did not. Perhaps, if I had time to look at each case on a phase by phase basis there might be evidence of that. As it is now, I did not find support using this new platform for the claim that morphological change accelerates the evolution of robust behaviour compared to no morphological change. My hypothesis that non-conservative morphological change (Bp, Bo) would not accelerate evolution is supported by the findings. My hypothesis that conservative morphological change (Ap, Ao) will accelerate evolution is not supported.

I would follow up on this work by exploring changing morphology phylogenetically since it has a smaller search space than the ontogenetic space. There may very well be a way to stage the morphological changes such that evolution is accelerated with this robot platform. An automated search of that space may take some time, but the results may provide further insights.

Bibliography

[1] R Beer. On the dynamics of small continuous-time recurrent neural networks. *Adaptive Behavior*, Jan 1995.

[2] J Bongard. Morphological and environmental scaffolding synergize when evolving robot controllers. *cems.uvm.edu*, Jan 2011.

[3] J Bongard. Morphological change in machines accelerates the evolution of robust behavior. *Proceedings of the National Academy of ...*, Dec 2011.

[4] J Bongard. Supporting information: Morphological change in machines accelerates the evolution of robust behavior. *Proceedings of the National Academy of Sciences*, 108(4):1234–1239, Jan 2011.

[5] J Bongard, V Zykov, and H Lipson. Resilient machines through continuous self-modeling. *Science*, Jan 2006.

[6] G Hornby. Alps: the age-layered population structure for reducing the problem of premature convergence. *Proceedings of the 8th annual conference on Genetic ...*, Jan 2006.

[7] G.S Hornby. Steady-state alps for real-valued problems. *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 795–802, 2009.

[8] N Jakobi. Minimal simulations for evolutionary robotics. *Citeseer*, Dec 1998.

[9] Thomas Reif Kane and David A. Levinson. Dynamics online: theory and implementation with autolev. Jan 2000.

[10] M Lungarella, G Metta, and R Pfeifer.... Developmental robotics: a survey. *Connection Sc.*, Jan 2003.

[11] Stefano Nolfi and Dario Floreano. Evolutionary robotics: The biology, intelligence, and technology of self .... page 320, Jan 2004.

Appendix A

Code

The code will be available in digital form at `http://github.com/secelis/sussex-thesis`.

A.1 Physical Model

The equations of motion were derived using AUTOLEV [9]. The source code is provided below. Figure A.1 shows how the axes relate to the rigid bodies in the code.

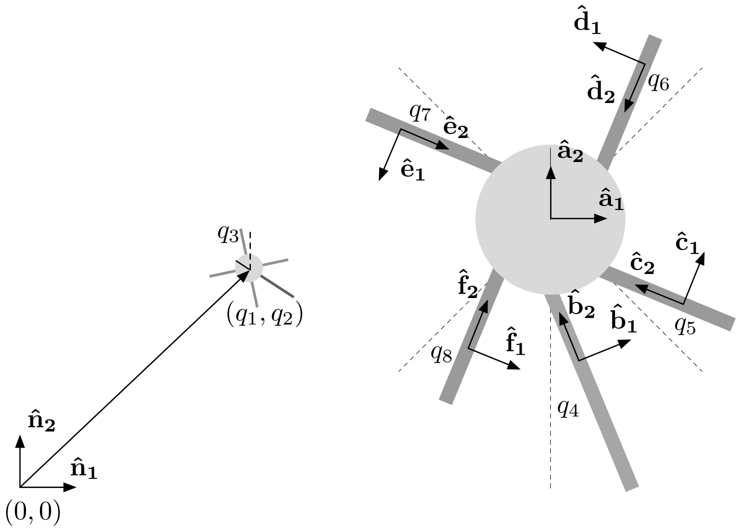

Figure A.1: Configuration variables and axes used in “frog.al”

Listing A.1: frog.al

```
% frog.al
%
% Mathematical model of a frog in a simulated liquid environment.
%
% Implemented with Autolev
```

```
%SetCompatible (AUTOLEV)
autorhs             on
autoz               off
body                a,b,c,d,e,f
                    % central body (a), tail (b), feet clockwise (c--f)
point               o,jb,jc,jd,je,jf,sr,sl
                    % origin and pin joint points for each body
newtonian           n
variables           q1', q2', q3', q4', q5', q6', q7', q8'
motionvariables'    u1', u2', u3', u4', u5', u6', u7', u8'

constants r, l, fl, oq4, oq5, oq6, oq7, oq8, Tq4, Tq5, Tq6, Tq7, Tq8
%          radius, tail length, foot length, offset for q_i, Torque for q_i
%constants ld, fld, rho, Cdcirc, Cdplate, TCdcirc, Acirc, TAcirc
%          tail depth, foot depth, ? coefficients, Area
constants kTa, kTb, kTc, kFa, kFb, kFc, krb, krc, wvx, wvy

% Set mass and moments of inertia.
mass a = ma, b = mb, c = mc, d = mc, e = mc, f = mc
inertia a_ao(n), 0, 0, Ia
inertia b_jb(a), 0, 0, Ib
inertia c_jc(a), 0, 0, Ic
inertia d_jd(a), 0, 0, Ic
inertia e_je(a), 0, 0, Ic
inertia f_jf(a), 0, 0, Ic

% Setup up the Reference Frames (RFs).
simprot(n,a,3,q3)
simprot(a,b,3,q4 + oq4)
simprot(a,c,3,q5 + oq5)
simprot(a,d,3,q6 + oq6)
simprot(a,e,3,q7 + oq7)
simprot(a,f,3,q8 + oq8)

% Set the motion variables.
q1' = u1
q2' = u2
q3' = u3
q4' = u4
q5' = u5
q6' = u6
q7' = u7
q8' = u8

% Set the positions of the pin joints with respect to body A.
P_o_ao>  = q1 * n1> + q2 * n2>
P_ao_jb> = r * (-a2>)
P_ao_jc> = r * unitvec( a1> - a2>)
P_ao_jd> = r * unitvec( a1> + a2>)
```

```
P_ao_je> = r * unitvec(-a1> + a2>)
P_ao_jf> = r * unitvec(-a1> - a2>)
P_ao_sl> = r * (-a1>)
P_ao_sr> = r * a1>

% Set the positions of the pin joints with respect to the bodies other than A.
P_jb_bo> = -l/2  * b2>
P_jc_co> = -fl/2 * c2>
P_jd_do> = -fl/2 * d2>
P_je_eo> = -fl/2 * e2>
P_jf_fo> = -fl/2 * f2>

% Set the angular velocities to their respective motion variables.
w_a_n> = u3 * a3>
w_b_a> = u4 * a3>
w_c_a> = u5 * a3>
w_d_a> = u6 * a3>
w_e_a> = u7 * a3>
w_f_a> = u8 * a3>

% Fix the pin joints in their respective RFs.
v_jb_a> = 0>
v_jb_b> = 0>
v_jc_a> = 0>
v_jd_a> = 0>
v_je_a> = 0>
v_jf_a> = 0>
v_jc_c> = 0>
v_jd_d> = 0>
v_je_e> = 0>
v_jf_f> = 0>

% Use the 2 point thereom to define velocity of each limb with respect to A.
v2pts(a,b,jb,bo)
v2pts(a,c,jc,co)
v2pts(a,d,jd,do)
v2pts(a,e,je,eo)
v2pts(a,f,jf,fo)

v_ao_n> = dt(p_o_ao>, n)
v2pts(n,a,ao,jb)

% Define the translational velocities.
v_bo_n> = dt(p_o_bo>, n)
v_co_n> = dt(p_o_co>, n)
v_do_n> = dt(p_o_do>, n)
v_eo_n> = dt(p_o_eo>, n)
v_fo_n> = dt(p_o_fo>, n)
```

```
% Define the translational accelerations.
a_ao_n> = dt(v_ao_n>, n)
a_bo_n> = dt(v_bo_n>, n)
a_co_n> = dt(v_co_n>, n)
a_do_n> = dt(v_do_n>, n)
a_eo_n> = dt(v_eo_n>, n)
a_fo_n> = dt(v_fo_n>, n)

% The units for torque_a> should be newton-meters (m/s)^2 kg.
% kTa = -rho/2 * TCdcirc * TAcirc
torque_a> = kTa * w_a_n> * mag(w_a_n>)
torque_b> = kTb * w_b_a> * mag(w_b_a>)
torque_c> = kTc * w_c_a> * mag(w_c_a>)
torque_d> = kTc * w_d_a> * mag(w_d_a>)
torque_e> = kTc * w_e_a> * mag(w_e_a>)
torque_f> = kTc * w_f_a> * mag(w_f_a>)
torque(a/b, Tq4 * n3>)
torque(a/c, Tq5 * n3>)
torque(a/d, Tq6 * n3>)
torque(a/e, Tq7 * n3>)
torque(a/f, Tq8 * n3>)

% velocity of the water current
wv> = wvx * n1> + wvy * n2>

% Set the drag force for each body.
% wikipedia drag force
% F_D = \frac{1}{2} \rho v^2 C_d A
% kFa = -rho/2 * Cdcirc  * Acirc
force_ao> = kFa * (v_ao_n> - wv>) * mag(v_ao_n> - wv>)
krb = 0
krc = 0
%wv> = 0>
% kFb = -rho/2 * Cdplate * ld
 force_bo> = kFb * l * (v_bo_n> - wv>) * abs(dot(b1>, (v_bo_n> - wv>)) + krb * SIGN(u4) * mag(v
%force_bo> = kFb * l * (v_bo_n> - wv>) * abs(dot(b1>, (v_bo_n> - wv>)) + krb * mag(v_bo_n>))
% kFc = -rho/2 * Cdplate * fld
force_co> = kFc * fl * (v_co_n> - wv>) * abs(dot(c1>, (v_co_n> - wv>)) + krc * mag(v_co_n>))
force_do> = kFc * fl * (v_do_n> - wv>) * abs(dot(d1>, (v_do_n> - wv>)) + krc * mag(v_do_n>))
force_eo> = kFc * fl * (v_eo_n> - wv>) * abs(dot(e1>, (v_eo_n> - wv>)) + krc * mag(v_eo_n>))
force_fo> = kFc * fl * (v_fo_n> - wv>) * abs(dot(f1>, (v_fo_n> - wv>)) + krc * mag(v_fo_n>))
i_b_bo>> = inertia(bo, b)
i_c_co>> = inertia(co, c)
i_d_do>> = inertia(do, d)
i_e_eo>> = inertia(eo, e)
i_f_fo>> = inertia(fo, f)
%eqns = fr() + frstar()
%eqns
fr() + frstar()
```

Name	Value	Description
r	0.025 m	radius of central body
lmax	0.06 m	maximum length of tail and feet
ma	0.025 kg	mass of central body
mb	0.00195 kg	mass of each tail and foot
Tmax	0.001797 N m	maximum torque for tail and feet
kTa	-0.0001	coefficient for torque drag for central body
kTb	-5.79837×10^{-6}	coefficient for torque drag for tail and feet
h	0.01	RK4 step size

Table A.1: caption

A.1.1 Equations of Motion

Below are the equations of motion produced by AUTOLEV. The last eight lines are the actual equations of motion that form a linear system in terms of $u_1', u_2', \ldots, u8'$. The preceding lines describe the computations necessary to determine the coefficients of the linear system. Symbolically the last eight lines look like Equation A.1

$$\mathbf{0} = \mathbf{b} + \mathbf{Z}\ \mathbf{u}' \tag{A.1}$$

A.1.2 Parameters for Physical Simulation

Listing A.2: frog_eqns.m

```
Z1 = COS(q3)
Z2 = SIN(q3)
Z3 = COS(oq4+q4)
Z4 = SIN(oq4+q4)
Z5 = COS(oq5+q5)
Z6 = SIN(oq5+q5)
Z7 = COS(oq6+q6)
Z8 = SIN(oq6+q6)
Z9 = COS(oq7+q7)
Z10 = SIN(oq7+q7)
Z11 = COS(oq8+q8)
Z12 = SIN(oq8+q8)
Z13 = Z1*Z3 - Z2*Z4
Z14 = -Z1*Z4 - Z2*Z3
Z15 = Z1*Z4 + Z2*Z3
Z16 = Z1*Z5 - Z2*Z6
```

```
Z17 = −Z1∗Z6 − Z2∗Z5
Z18 = Z1∗Z6 + Z2∗Z5
Z19 = Z1∗Z7 − Z2∗Z8
Z20 = −Z1∗Z8 − Z2∗Z7
Z21 = Z1∗Z8 + Z2∗Z7
Z22 = Z1∗Z9 − Z2∗Z10
Z23 = −Z1∗Z10 − Z2∗Z9
Z24 = Z1∗Z10 + Z2∗Z9
Z25 = Z1∗Z11 − Z2∗Z12
Z26 = −Z1∗Z12 − Z2∗Z11
Z27 = Z1∗Z12 + Z2∗Z11
Z28 = r∗u3
Z29 = l∗(u3+u4)
Z30 = u3∗Z28
Z31 = (u3+u4)∗Z29
Z32 = fl∗(u3+u5)
Z33 = (u3+u5)∗Z32
Z34 = fl∗(u3+u6)
Z35 = (u3+u6)∗Z34
Z36 = fl∗(u3+u7)
Z37 = (u3+u7)∗Z36
Z38 = fl∗(u3+u8)
Z39 = (u3+u8)∗Z38
Z40 = ABS(u3)
Z41 = kTa∗u3∗Z40
Z42 = ABS(u4)
Z43 = kTb∗u4∗Z42
Z44 = ABS(u5)
Z45 = kTc∗u5∗Z44
Z46 = ABS(u6)
Z47 = kTc∗u6∗Z46
Z48 = ABS(u7)
Z49 = kTc∗u7∗Z48
Z50 = ABS(u8)
Z51 = kTc∗u8∗Z50
Z52 = Z41 − Tq4
Z53 = Z52 − Tq5
Z54 = Z53 − Tq6
Z55 = Z54 − Tq7
Z56 = Z55 − Tq8
Z57 = u1 − wvx
Z58 = u2 − wvy
Z59 = kFa∗(wvx−u1)∗(Z57^2+Z58^2)^0.5
Z60 = kFa∗(wvy−u2)∗(Z57^2+Z58^2)^0.5
Z61 = ABS(Z13∗u1+Z15∗u2+r∗Z3∗u3+0.5∗l∗(u3+u4)+0.5∗krb∗SIGN(u4)∗(4∗u1^2+4∗u2^2+Z29^2+4∗Z28^2+4∗Z
Z62 = kFb∗l∗r
Z63 = Z62∗u3∗Z61
Z64 = kFb∗l^2
Z65 = Z64∗(u3+u4)∗Z61
```

```
Z66 = kFb*l
Z67 = Z66*u1*Z61
Z68 = Z66*u2*Z61
Z69 = ABS(Z16*u1+Z18*u2+0.7071068*r*Z5*u3+0.7071068*r*Z6*u3+0.5*fl*(u3+u5))
Z70 = fl*kFc*r
Z71 = Z70*u3*Z69
Z72 = kFc*fl^2
Z73 = Z72*(u3+u5)*Z69
Z74 = fl*kFc
Z75 = Z74*u1*Z69
Z76 = Z74*u2*Z69
Z77 = ABS(Z19*u1+Z21*u2+0.7071068*r*Z8*u3+0.5*fl*(u3+u6)-0.7071068*r*Z7*u3)
Z78 = Z70*u3*Z77
Z79 = Z72*(u3+u6)*Z77
Z80 = Z74*u1*Z77
Z81 = Z74*u2*Z77
Z82 = ABS(Z22*u1+Z24*u2+0.5*fl*(u3+u7)-0.7071068*r*Z9*u3-0.7071068*r*Z10*u3)
Z83 = Z70*u3*Z82
Z84 = Z72*(u3+u7)*Z82
Z85 = Z74*u1*Z82
Z86 = Z74*u2*Z82
Z87 = ABS(Z25*u1+Z27*u2+0.7071068*r*Z11*u3+0.5*fl*(u3+u8)-0.7071068*r*Z12*u3)
Z88 = Z70*u3*Z87
Z89 = Z72*(u3+u8)*Z87
Z90 = Z74*u1*Z87
Z91 = Z74*u2*Z87
Z92 = mb*l^2
Z93 = mc*fl^2
   (326) eqns = fr() + frstar()
Z94 = Z67 + Z75 + Z80 + Z85 + Z90 + Z1*Z63 + 0.5*Z13*Z65 + 0.5*Z16*Z73 + 0.5*Z19*Z79 + 0.5*Z22*
Z95 = Z68 + Z76 + Z81 + Z86 + Z91 + Z2*Z63 + 0.5*Z15*Z65 + 0.5*Z18*Z73 + 0.5*Z21*Z79 + 0.5*Z24*
Z96 = Tq4 + Tq5 + Tq6 + Tq7 + Tq8
Z97 = Z96 + Z43 + Z45 + Z47 + Z49 + Z51 + Z56 + r*Z63 + 0.25*fl*Z73 + 0.25*fl*Z79 + 0.25*fl*Z84
Z98 = Tq4 + Z43 + 0.25*l*(Z65+2*Z3*Z63+2*Z13*Z67+2*Z15*Z68)
Z99 = Tq5 + Z45 + 0.25*fl*(Z73+1.414214*Z5*Z71+1.414214*Z6*Z71+2*Z16*Z75+2*Z18*Z76)
Z100 = Tq6 + Z47 - 0.25*fl*(1.414214*Z7*Z78-Z79-2*Z19*Z80-2*Z21*Z81-1.414214*Z8*Z78)
Z101 = Tq7 + Z49 - 0.25*fl*(1.414214*Z9*Z83+1.414214*Z10*Z83-Z84-2*Z22*Z85-2*Z24*Z86)
Z102 = Tq8 + Z51 - 0.25*fl*(1.414214*Z12*Z88-Z89-2*Z25*Z90-2*Z27*Z91-1.414214*Z11*Z88)
Z103 = Ib*u3 + Ib*u4 - 0.25*Z92*u4
Z104 = Z92*u3
Z105 = Ic*u3 + Ic*u5 - 0.25*Z93*u5
Z106 = Z93*u3
Z107 = Ic*u3 + Ic*u6 - 0.25*Z93*u6
Z108 = Ic*u3 + Ic*u7 - 0.25*Z93*u7
Z109 = Ic*u3 + Ic*u8 - 0.25*Z93*u8
Z110 = ma + mb + 4*mc
Z111 = fl*mc
Z112 = 0.5*Z111*Z16 + 0.5*Z111*Z19 + 0.5*Z111*Z22 + 0.5*Z111*Z25 + 0.5*mb*(l*Z13+2*r*Z1)
Z113 = l*mb
```

```
Z114 = Z113*Z13
Z115 = Z111*Z16
Z116 = Z111*Z19
Z117 = Z111*Z22
Z118 = Z111*Z25
Z119 = 0.5*mc*Z17*Z33 + 0.5*mc*Z20*Z35 + 0.5*mc*Z23*Z37 + 0.5*mc*Z26*Z39 - 0.5*mb*(2*Z2*Z30-Z14
Z120 = 0.5*Z111*Z18 + 0.5*Z111*Z21 + 0.5*Z111*Z24 + 0.5*Z111*Z27 + 0.5*mb*(l*Z15+2*r*Z2)
Z121 = Z113*Z15
Z122 = Z111*Z18
Z123 = Z111*Z21
Z124 = Z111*Z24
Z125 = Z111*Z27
Z126 = 0.5*mc*Z16*Z33 + 0.5*mc*Z19*Z35 + 0.5*mc*Z22*Z37 + 0.5*mc*Z25*Z39 + 0.5*mb*(Z13*Z31+2*Z1
Z127 = Ia + Ib + 4*Ic
Z128 = fl*mc*r
Z129 = mc*r
Z130 = l^2 + 4*r^2
Z131 = l*r
Z132 = Z127 + 0.7071068*Z128*(Z11-Z12) + 0.25*mb*(Z130+4*Z131*Z3) + 0.7071068*Z129*(5.656854*r+
Z133 = Ib + 0.25*Z113*(l+2*r*Z3) - 0.25*Z92
Z134 = Ic + 0.25*Z111*(fl+1.414214*r*Z5+1.414214*r*Z6) - 0.25*Z93
Z135 = Ic - 0.25*Z93
Z136 = Z135 - 0.25*Z111*(1.414214*r*Z7-fl-1.414214*r*Z8)
Z137 = Ic + 0.25*Z111*(fl-1.414214*r*Z9-1.414214*r*Z10) - 0.25*Z93
Z138 = Ic + 0.25*Z111*(fl+1.414214*r*Z11-1.414214*r*Z12) - 0.25*Z93
Z139 = 0.5*mb*Z4*(l*Z30-r*Z31) + 0.3535534*mc*(fl*Z6*Z30+r*Z5*Z33-fl*Z5*Z30-r*Z6*Z33) + 0.35355
Z140 = Ib + 0.25*mb*l^2 - 0.25*Z92
Z141 = Z113*Z4*Z30
Z142 = Ic + 0.25*mc*fl^2 - 0.25*Z93
Z143 = Z111*(Z5-Z6)*Z30
Z144 = Z111*(Z7+Z8)*Z30
Z145 = Z111*(Z9-Z10)*Z30
Z146 = Z111*(Z11+Z12)*Z30
0 = Z94 - Z119 - Z110*u1' - Z112*u3' - 0.5*Z114*u4' - 0.5*Z115*u5' - 0.5*Z116*u6' - 0.5*Z117*u7
0 = Z95 - Z126 - Z110*u2' - Z120*u3' - 0.5*Z121*u4' - 0.5*Z122*u5' - 0.5*Z123*u6' - 0.5*Z124*u7
0 = Z97 - Z139 - Z112*u1' - Z120*u2' - Z132*u3' - Z133*u4' - Z134*u5' - Z136*u6' - Z137*u7' - Z
0 = Z98 - 0.5*Z141 - Z140*u4' - Z133*u3' - 0.5*Z114*u1' - 0.5*Z121*u2'
0 = Z99 + 0.3535534*Z143 - Z142*u5' - Z134*u3' - 0.5*Z115*u1' - 0.5*Z122*u2'
0 = Z100 + 0.3535534*Z144 - Z142*u6' - Z136*u3' - 0.5*Z116*u1' - 0.5*Z123*u2'
0 = Z101 - 0.3535534*Z145 - Z142*u7' - Z137*u3' - 0.5*Z117*u1' - 0.5*Z124*u2'
0 = Z102 - 0.3535534*Z146 - Z142*u8' - Z138*u3' - 0.5*Z118*u1' - 0.5*Z125*u2'
```

A.1.3 Numerical Simulation

Listing A.3: rk4.c

```
int rk4(double y[], double dydx[], const int n, double x, double h, double yout[], int (*derivs
/*Given values for n variables y[1..n] and their derivatives
```

```
 * dydx[1..n] known at x, use the RK4 method to advance the solution
 * over an interval h and return the incremented variables as
 * yout[1..n]. */
{
  int i, err;
  double ak2[n],ak3[n],ak4[n],ytemp[n];

  for (i=0; i<n; i++)              //First step.
    ytemp[i]=y[i]+ h*dydx[i]; // ak1[i] = h dydx[i]
  err = (*derivs)(x+0.5*h,ytemp,ak2,context);
  if (err) return err;
  for (i=0;i<n;i++)                //Second step.
    ytemp[i]=y[i]+h*(0.5 *ak2[i]);
  err = (*derivs)(x+0.5*h,ytemp,ak3,context);
  if (err) return err;
  for (i=0; i<n; i++)              //Third step.
    ytemp[i]=y[i]+h*(ak3[i]);
  err = (*derivs)(x+h,ytemp,ak4,context);
  if (err) return err;
  for (i=0; i<n; i++)        //Accumulate increments with proper weights.
    yout[i]=y[i]+(h/6.)*(dydx[i] + 2.* ak2[i] + 2.*ak3[i]+ak4[i]);
  //Estimate error as difference between fourth and fifth order methods.
  return 0;
}
```

A.1.4 CTRNN

Listing A.4: frog_eqns.m

```
(* ::Package:: *)

(* ::Title:: *)
(*CTRNN*)



(* Requirements to specify a CTRNN:
   W (nxn) matrix
   theta (nx1) vector
   input (t -> nx1) function
   time constant (nx1) vector *)



sigma[x_] := 1/(1 + E^(-x))



makeSymbolicCTRNN[n_]:=
    Module[{range, Ws, thetas, inputs, Ts},
            Ws = Array[W, {n,n}];
            thetas =Array[theta, {n}];
            inputs= Map[Function[{a},input[a][#]&],Range[n]]; (* has constant inputs *)
```

```
                Ts = Array[tc, {n}];
                {Ws, thetas, inputs, Ts}]


makeRandomCTRNN[n_] :=
     Module[{range, W, theta, input, Ts},
                range = {-1, 1};
                W = RandomReal[range, {n,n}];
                theta = RandomReal[range, {n}];
                input= Map[Function[{t}, #]&,RandomReal[range, {n}]]; (* has constant inputs *)
                Ts = RandomReal[10^range, {n}];
                {W, theta, input, Ts}]


makeRandomCTRNNLinSensor[n_, sensorCount_] :=
     Module[{ctrnn, range, sensorMat},
                range = {-1,1};
                sensorMat = RandomReal[range, {n, sensorCount}];
                ctrnn = makeRandomCTRNN[n];
                (*ctrnn[[3]] = makeLinSensorInputs[n, ctrnn[[3]]];*)
                ctrnn~Join~{sensorMat}]


makeSymbolicCTRNNLinSensor[n_, sensorCount_] :=
     Module[{ctrnn, sensorMat, s},
                ctrnn = makeSymbolicCTRNN[n];
                sensorMat = Array[nij, {nodeCountCTRNN[ctrnn], sensorCount}];
                (*sensorMat = sensorMat /. substituteRules[Flatten[sensorMat], sc];*)
                ctrnn = ctrnn~Join~{sensorMat};
                ctrnn[[3]] = makeLinSensorInputs[nodeCountCTRNN[ctrnn],
                                                     Array[sensor, sensorCount]];
                ctrnn
               ]


makeZeroCTRNNLinSensor[n_, sensorCount_] :=
     Module[{ctrnn, range, sensorMat},
                range = {-1,1};
                sensorMat = 0 RandomReal[range, {n, sensorCount}];
                ctrnn = makeZeroCTRNN[n];
                ctrnn~Join~{sensorMat}]


makeZeroCTRNN[n_] :=
     Module[{ctrnn},
                ctrnn= makeSymbolicCTRNN[n];
                ctrnn = 0. ctrnn;
                ctrnn[[4]] = Table[1., {n}];
                ctrnn[[3]] = Table[0.&, {n}];
```

```
              ctrnn]


nodeCountCTRNN[ctrnn_] :=Length[First[ctrnn]]


Options[eqnsForCTRNN] = {otherEqns -> {}};


eqnsForCTRNN[ctrnn_List,state_List, OptionsPattern[]] :=
    Module[{W,theta,input, eqns, ICs, sols, Tinv, y, dy, n, gain},
            n = nodeCountCTRNN[ctrnn];
            y = Table[ys[i][0], {i, n}];
            ICs = MapThread[#1 == #2&, {y, state}];
            eqnsForCTRNN[ctrnn, otherEqns -> ICs~Join~OptionValue[otherEqns]]]


eqnsForCTRNN[ctrnn_List,OptionsPattern[]] :=
    Module[{W,theta,input, Ts, eqns, ICs, sols, Tinv, y, dy, n, gain},
            {W,theta,input,Ts} = Take[ctrnn, 4];
            gain =1;
            n =nodeCountCTRNN[ctrnn];
            y = Table[ys[i][t], {i, n}];
            dy = Table[ys[i]'[t], {i,n}];
            (*Tinv = DiagonalMatrix[Ts^-1];*)
            Tinv = DiagonalMatrix[Ts];
            eqns = MapThread[#1 == #2&,
                              {dy, Tinv . (W . sigma[y + theta] -y + gain Map[#[t]&,input]) }];
            {eqns~Join~OptionValue[otherEqns], y}]


eqnsForCTRNN[ctrnn_List,OptionsPattern[]] :=
    Module[{W,theta,input, Ts, eqns, ICs, sols, Tinv, y, dy, n, gain, bound},
            {W,theta,input,Ts} = Take[ctrnn, 4];
            gain =1;
            n =nodeCountCTRNN[ctrnn];
            y = Table[ys[i][t], {i, n}];
            dy = Table[ys[i]'[t], {i,n}];
            Tinv = DiagonalMatrix[Ts];
            bound = IdentityMatrix[n];
            bound = DiagonalMatrix[Map[1 - Abs[boundaries[#, {-2,2}]]&, y]];
            eqns = MapThread[#1 == #2&, {dy, Tinv . (-y + bound . ( W . sigma[y + theta] + gain
                                          (* try to keep it bounded *)
                                          (*- 100 y  Map[Abs[boundaries[#, {-2,2}]]&}, y]*)
                              }];
(*sols = NDSolve[{eqns, ICs},y, {t,0, 5}];
   sols*)
{eqns~Join~OptionValue[otherEqns], y}]
```

```
substituteRules[vars_, v_] :=
    Module[{},
            Quiet[MapThread[#1 -> #2&, {vars,Table[v[[i]], {i, Length[vars]}]}],
                  {Part::partd}]]


(* sensors :: [Real -> Real]
   time to sensor value *)
makeLinSensorInputs[nodeCount_, sensors_] :=
    Module[{mat, s, inputs, sensors2},
            mat = Array[nij,{nodeCount, Length[sensors]}];
            sensors2 = Map[Function[{a}, a[#]], sensors];
            inputs = Map[Function[{a},Function[Evaluate[a]]],mat . sensors2];
            (*inputs /. substituteRules[Flatten[mat], sc]*)
            inputs
           ]



(* sensors :: [Real -> Real]
   time to sensor value *)
makeLinSensorInputs[ctrnn_List, sensors_] :=
    Module[{mat, s, inputs, sensors2, nodeCount},
            nodeCount = nodeCountCTRNN[ctrnn];
            mat = ctrnn[[5]];
            sensors2 = Map[Function[{a}, a[#]], sensors];
           Map[Function[{a},Function[Evaluate[a]]],mat . sensors2]
           ]



solveCTRNN[ctrnn_, state_] :=
    Module[{eqns, vars},
            {eqns, vars} = eqnsForCTRNN[ctrnn, state];
            NDSolve[eqns, vars, {t, 0, 100}(*, Method -> "ExplicitEuler",StartingStepSize ->0.0


makeRandomCTRNNState[n_] := RandomReal[.1 {-1,1}, {n}]



makeZeroCTRNNState[n_] := Table[0, {n}]



onesForTimeCs[ctrnnArg_] :=
    Module[{n, ctrnn},
            ctrnn = ctrnnArg;
            n = nodeCountCTRNN[ctrnn];
            ctrnn[[4]] = Table[1, {n}];
            ctrnn]
```

A.2 Evolutionary Algorithm

Listing A.5: alps-like.c

```c
/* alps-like.c */

#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <float.h>
#include <strings.h>
#include <string.h>
#include <time.h>
#include <unistd.h>
#include <assert.h>
#include <stdarg.h>
#include <sys/stat.h>

#include "run-simulation.h"
#include "alps-like.h"
#include "pareto_front.h"
#include "alps_frog.h"

#define rand            drand48

#define MUT_PROB        0.05       // mutation probability
#define MAX_OPT_STEPS   100000     // max optimisation steps
//#define LAYER_COUNT     15
#define LAYER_COUNT     10
#define POP_PER_LAYER   10
#define POP             (LAYER_COUNT * POP_PER_LAYER)
#define MAX_LAYER       (LAYER_COUNT - 1)
#define BAD_FITNESS     666.0
#define RESET_FREQ      (POP * 2)
#define DISPLAY_FREQ    300
//#define MAX_SECONDS     (20 * 60) // 20 minutes maximum.
#define MAX_SECONDS     (40 * 60) // 20 minutes maximum.

#define LAYER_OF_INDIV(i)      ((i) / POP_PER_LAYER)

/* INDEX = Index of individual */
/* LINDEX = Layer Index (layer, index of individual within layer) */
#define FITNESS_INDEX(i)        ((i) * FITNESS_COUNT)
#define LAYER_BEGINS(l)         (POP_PER_LAYER * (l))
#define INDIV_LINDEX(l, li)     (LAYER_BEGINS(l) + (li))
#define FITNESS_LINDEX(l, li)   (INDIV_LINDEX(l, li) * FITNESS_COUNT)


double genes[POP][GENE_COUNT];    // Genotypes of the population.
```

```
int      ages[POP];
double fitness_matrix[POP * FITNESS_COUNT];
int      pareto_front[POP];
int      t;                          /* optimisation step or time */
int      max_age[/*LAYER_COUNT*/] = {1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233,
                                   377, 610, 987, /*1597, 2584, 4181, 6765, 10946*/};

/* The maximum possible age is a_max = opt_steps/POP.
   The evaluation rate is e_r = ~21 evals/second.
   max_evals = e_r * MAX_SECONDS
   a_max ~= max_evals/POP.
 */
int      goal_indiv;
int      phase;
int      run_type = STANDARD_RUN;
long     random_seed;
int      eval_succ_count = 0;
int      eval_fail_count = 0;
time_t begin;
FILE   *mfile;
FILE   *table;
FILE   *script;
double goal_fitness = 0.5;
int      quiet;
double mut_prob = MUT_PROB;
int      reset_freq = RESET_FREQ;
int      alps_status = ALPS_SUCC;

//    experiment parameters
char   *exp_name;
int    task_index;
int    lobotomise;
int    fitness_type;
int    phase_count;
char   *save_prefix;


int mprintf(int add_prefix, const char *fmt, ...);
long elapsed_seconds();

void init_population() {
  int i;
  for (i = 0; i < POP; i++) {
    init_gene(genes[i]);
    ages[i] = i/POP; // or zero
  }
}

void init_gene(double *gene) {
```

```
  int i;
  for(i = 0; i < GENE_COUNT; i++)
    gene[i] = rand(); // [0, 1)
  //gene[i] = rand() + 0.5; // [0, 1)
}

void mutate(double *gene) {
  int i;
  for(i = 0; i < GENE_COUNT; i++)
    if (rand() < mut_prob)
      gene[i] = rand();
}

void copy(double *src, double *dest) {
  memcpy(dest, src, sizeof(double) * GENE_COUNT);
}

int  evaluate(double *gene, double *fitness)
{
  int i, err;
  err = evaluate_frog(fitness, gene, exp_name, phase, task_index,
                      lobotomise, fitness_type);
  if (err) {
    for (i = 0; i < FITNESS_COUNT; i++)
      fitness[i] = BAD_FITNESS;
    eval_fail_count++;
  } else {
    eval_succ_count++;
  }
  return err;
}

/* Does a dominate b? Return true iff f(a)_i < f(b)_i for all i.*/
int is_dominated(double *fitness_a, double *fitness_b)
{
  int i;
  for (i = 0; i < FITNESS_COUNT; i++) {
    if (fitness_a[i] >= fitness_b[i])
      return 0;
  }
  return 1;
}

/*
  Returns i >= 0 where i is the index of the gene that it dislodged
  itself to.  i < 0 indicates it did not dislodge any gene.
 */
int try_dislodge(double *attempter, double *fitness, int age, int into_layer)
{
```

```
  int i, k = into_layer;
  if (k > MAX_LAYER) {
    // Sorry. No layer above to dislodge oneself to.
    return -1;
  }
  // start_search helps distribute the dislodge search evenly.
  int start_search = rand() * POP_PER_LAYER;
  for (i = 0; i < POP_PER_LAYER; i++) {
    int li = (start_search + i) % POP_PER_LAYER;
    int pi = INDIV_LINDEX(k, li);
    double *fitness_b = fitness_matrix + FITNESS_LINDEX(k, li);
    if (is_dominated(fitness, fitness_b)
        || ages[pi] > max_age[k]) {
      // New genome dominates current one, or the current one is too old.
      try_dislodge(genes[pi],
                   fitness_b,
                   ages[pi],
                   k + 1);

      copy(attempter, genes[pi]);
      memcpy(fitness_b, fitness, sizeof(double) * FITNESS_COUNT);
      ages[pi] = age;
      return pi;
    }
  }
  return -2;
}

void print_fitness(double *fitness) {
  int i;
  for (i = 0; i < FITNESS_COUNT; i++)
    printf("%f␣", fitness[i]);
  printf("\n");
}

int is_goal_fitness(double *fitness) {
  int i;
  for (i = 0; i < FITNESS_COUNT; i++) {
    if (!(fitness[i] < goal_fitness))
    //if (!(fitness[i] < -5.0))
      return 0;
  }
  return 1;
}

void start_phase(int phase) {
  if (! quiet)
    printf("alps-like:␣Starting␣phase␣%d␣on␣step␣%d.\n", phase, t);
}
```

```
void add_to_script(char *filename, int phase, double *fitness) {
  int i;

  fprintf(script, "frog_eval␣%s␣%s␣%d␣%d␣%d␣%d;␣",
          filename, exp_name, phase, task_index + 1, lobotomise,
          fitness_type);
  fprintf(script, "echo␣recorded␣fitness:␣");
  for (i = 0; i < FITNESS_COUNT; i++)
    fprintf(script, "%f␣", fitness[i]);
  fprintf(script, ";␣echo\n");
}



char *save_gene_for_phase_and_front(int pi /* population index */, const char *save_prefix,
                                   int phase, int front_index)
{
  static char gene_save_name[255];
  double *gene = genes[pi];
  double *fitness = fitness_matrix + FITNESS_INDEX(pi);
  sprintf(gene_save_name, "p%d-f%dgene.bin", phase, front_index);
  add_to_script(gene_save_name, phase, fitness);
  sprintf(gene_save_name, "%s/p%d-f%dgene.bin", save_prefix, phase, front_index);
  mprintf(1, "individOnFront[{phase␣->␣%d,␣frontIndex␣->␣%d}]␣->␣\n", phase, front_index + 1);

  assert(FITNESS_COUNT == 2);
  mprintf(1, "\t{fitness␣->␣{%lf,␣%lf},␣age␣->␣%d,␣layer␣->␣%d,␣meetsGoal␣->␣%s,␣geneFilename␣-
          fitness[0], fitness[1], ages[pi], LAYER_OF_INDIV(pi), is_goal_fitness(fitness) ? "Tru
          gene_save_name);

#ifdef PRINT_GENE_CHAR
  mprintf(1, ",␣gene␣->␣{␣%lf" , gene[0]);
  int i;
  for (i = 1; i < GENE_COUNT; i++)
    mprintf(0, ",␣%lf", gene[i]);
  mprintf(0, "}␣");
#endif
  mprintf(0, "}\n");

  write_array(gene_save_name, GENE_COUNT, gene);
  return gene_save_name;
}

void end_phase(int phase) {
  int i, fi, full_front[POP], age_max;

  fprintf(table, "%d␣%d␣%d\n", (alps_status == ALPS_FAIL ? -1 : 1) * phase, eval_fail_count, ev

  pareto_front_rowmajor(full_front, fitness_matrix, POP, FITNESS_COUNT);
```

```
  age_max = 0;
  for (i = fi = 0; i < POP; i++) {
    age_max = fmax(age_max, ages[i]);
    if (full_front[i]
        /*&& is_goal_fitness(fitness_matrix + FITNESS_INDEX(i))*/) {
      save_gene_for_phase_and_front(i, save_prefix, phase, ++fi);
    }
  }
  mprintf(1, "phaseEnd␣->␣{phase␣->␣%d,␣ageMax␣->␣%d,␣evalFailedCount␣->␣%d,␣evalSuccCount␣->␣%
}


int mprintf(int add_prefix, const char *fmt, ...)
{
  va_list ap;
  int len;
  if (mfile) {
    va_start(ap, fmt);
    len = vfprintf(mfile, fmt, ap);
    va_end(ap);
  }

  if (! quiet) {
    char buf[512];
    if (add_prefix) {
      sprintf(buf, ":m:␣%s", fmt);
    } else {
      sprintf(buf, "%s", fmt);
    }
    va_start(ap, fmt);
    vprintf(buf, ap);
    va_end(ap);
  }
  return len;
}

long elapsed_seconds() {
  time_t now = time(NULL);
  return now - begin;
}

int run_alps(int *steps)
{
  int i,j,k,p;
  double temp_fitness[FITNESS_COUNT];
  double temp_gene[GENE_COUNT];
  int pareto_front[POP];
  int pareto_count;
```

```
begin = time(NULL);
init_population();              // Initialise population.
t = 0;
goal_indiv = -1;

mprintf(1, "preamble -> {expName -> %s, phaseCount -> %d, lobotomise -> %s, task -> %d, \n",
mprintf(1, "\ttmax -> %.2lf, fitnessType -> %d, runType -> %d, randomSeed -> %ld }\n", TIME_M

mprintf(1, "alpsParams -> { layerCount -> %d, popPerLayer -> %d, "
        "popCount -> %d, mutProbability -> %.3f, maxSeconds -> %d }\n",
        LAYER_COUNT, POP_PER_LAYER, POP, mut_prob, MAX_SECONDS);
fflush(stdout);

for (p = 0, phase = 1;
     p < phase_count && elapsed_seconds() < MAX_SECONDS;
     p++, phase = p + 1) {

  if (p != 0) end_phase(p);
  start_phase(phase);
  for (i = 0; i < POP; i++) {
    // Evaluate every gene.
    evaluate(genes[i], fitness_matrix + FITNESS_INDEX(i));
  }
  int met_goal = 0;
  for (j = 0; j < POP; j++) {
    if (is_goal_fitness(fitness_matrix + FITNESS_INDEX(j))) {
      met_goal = 1;
      goal_indiv = j;
    }
  }
  if (met_goal)
    continue;

  for (; t < MAX_OPT_STEPS && elapsed_seconds() < MAX_SECONDS; t++) {
    // Evaluate the pareto front for each layer.
    for (k = 0; k < LAYER_COUNT; k++)
      pareto_front_rowmajor(pareto_front + k * POP_PER_LAYER,
                            fitness_matrix
                            + k * FITNESS_COUNT * POP_PER_LAYER,
                            POP_PER_LAYER,
                            FITNESS_COUNT);

    // Grab a non-dominated individual.
    int a;
    // O(n) single pass to count and grab a random individual
    // that's on the pareto front.
    pareto_count=0;
    for (i = 0; i < POP; i++)
      if (pareto_front[i] && (rand() < 1./(double)++pareto_count))
```

```
          a = i;
    k = LAYER_OF_INDIV(a);


    if (t % DISPLAY_FREQ == 0) {
      int n = FITNESS_INDEX(a);
      if (! quiet)
      printf("t␣=␣%5d,␣pi␣=␣%3d,␣a␣=␣%2d,␣k␣=␣%2d,␣N(PF)␣=␣%2d,␣f(a)␣=␣{%f,␣%f},␣"
             "efail␣=␣%3d,␣esucc␣=␣%5d,␣secs␣=␣%4ld\n", t, a, ages[a], k,
             pareto_count, fitness_matrix[n], fitness_matrix[n + 1],
             eval_fail_count, eval_succ_count, elapsed_seconds());

      fflush(stdout);
    }

    if (t % reset_freq == 0) {
      // Reset the bottom layer.
      for (i = 0; i < POP_PER_LAYER; i++) {
        // Try to dislogde in the layer above if it's in the pareto front.
        if (pareto_front[i])
          try_dislodge(genes[i], fitness_matrix + FITNESS_LINDEX(0, i), ages[i], 1);

        init_gene(genes[i]);
        ages[i] = 0;
      }
      for (i = 0; i < POP_PER_LAYER; i++)
        evaluate(genes[i], fitness_matrix + FITNESS_INDEX(i));
      pareto_front_rowmajor(pareto_front, fitness_matrix, POP_PER_LAYER,
                            FITNESS_COUNT);
    }

    copy(genes[a], temp_gene);
    mutate(temp_gene);
    int age = t/POP;
    evaluate(temp_gene, temp_fitness);
    int new_i = try_dislodge(temp_gene, temp_fitness, age, k);
    if (is_goal_fitness(temp_fitness)) {
      if (new_i < 0) {
        printf("warning:␣goal␣individual␣not␣able␣to␣dislodge␣anyone␣in␣layer␣%d␣and␣up.\n",
      }
      goal_indiv = new_i;
      break;                  /* Goto next phase */
    }
  }
}
if (t == MAX_OPT_STEPS || elapsed_seconds() > MAX_SECONDS)
  alps_status = ALPS_FAIL;
else
  alps_status = ALPS_SUCC;
```

```
  end_phase(phase - 1);
  if (steps)
    *steps = t;

  return alps_status;
}

int main(int argc, char **argv) {

  int c;
  int argco = argc;
  char **argvo = argv;
  fitness_type = FITNESS_MEAN_LIGHTSENSOR;
  int run_type = STANDARD_RUN;
  pid_t pid = getpid();
  random_seed = (long) time(NULL) ^ pid;
  save_prefix = ".";

  quiet = 0;
  int force = 0;
  while ((c = getopt (argc, argv, "DT:F:s:fqM:d:R:")) != -1)
    switch (c)
    {
    case 'D':
      run_type = DEBUG_RUN;          break;
    case 'T':
      run_type = atoi(optarg);       break;
    case 'F':
      fitness_type = atoi(optarg);   break;
    case 's':
      random_seed = atol(optarg);    break;
    case 'f':
      force = 1;                     break;
    case 'q':
      quiet = 1;                     break;
    case 'M':
      mut_prob = atof(optarg);       break;
    case 'd':
      save_prefix = optarg;          break;
    case 'R':
      reset_freq = atoi(optarg);     break;
    case '?':
      break;
    default:
      abort ();
    }
  // next argument at argv[optind]
  argc -= (optind - 1);
  argv += (optind - 1);
```

```
srand48(random_seed);

if (! quiet)
  printf("alps-like:␣Started!\n");

if (run_type == DEBUG_RUN) {
  goal_fitness = 0.9;
} else if (run_type == EASY_RUN) {
  goal_fitness = 0.8;
}



if (argc != 4) {
  fprintf(stderr, "usage:␣alps-like␣[-fDq]␣[-M␣mutP]␣[-R␣resetF]␣[-T␣run-type]␣[-F␣fitness-ty
  fprintf(stderr, "experiment␣names:␣An,␣Bn,␣Ap,␣Bp,␣Ao,␣Bo\n");
  return 2;
}

//register_signal_handlers();
sim_init();
exp_name = argv[1];
task_index = atoi(argv[2]) - 1;
lobotomise = (atoi(argv[3]) == 1);

//if (mkdir(save_prefix, 0777)) {
char cmd[255];
sprintf(cmd, "mkdir␣-p␣%s", save_prefix);
if (system(cmd)) {
  fprintf(stderr, "error:␣cannot␣create␣directory␣'%s'.\n", save_prefix);
  return 4;
}

char filename[255];
sprintf(filename, "%s/results.m", save_prefix);
mfile = fopen(filename, "w");

sprintf(filename, "%s/table.txt", save_prefix);
table = fopen(filename, "w");

sprintf(filename, "%s/run_again.sh", save_prefix);
FILE *run_again = fopen(filename, "w");
fprintf(run_again, "#!/bin/bash\n");
int i;
mprintf(1, "commandLine␣->␣\"");
for (i = 0; i < argco; i++) {
  fprintf(run_again, "%s␣", argvo[i]);
  mprintf(0, "%s␣", argvo[i]);
}
mprintf(0, "\"\n");
```

```
  fprintf(run_again, "\n");
  fclose(run_again);

  mprintf(1, "directory␣->␣\"%s\"\n", save_prefix);

  int err;
  err = experiment_phase_count(exp_name, &phase_count);
  if (err) {
    fprintf(stderr, "error:␣cannot␣get␣phase␣count␣for␣experiment␣'%s'.\n",
            exp_name);
    err = 1;
    goto finish;
  }

  sprintf(filename, "%s/eval.sh", save_prefix);
  script = fopen(filename, "w");

  fprintf(script,
          "#!/bin/bash\n"
          "cd␣$(dirname␣$0)\n");

  int steps;
  err = run_alps(&steps);

  if (err) {
    fprintf(stderr, "error:␣alps-like␣did␣not␣find␣an␣adequate␣solution.\n");
  }
  mprintf(1, "{␣success␣->␣%s,␣exitCode␣->␣%d␣}\n", err == 0 ? "True" : "False",
          err);

  sprintf(filename, "%s/FAILURE", save_prefix);
  if (err) {
    FILE* fail = fopen(filename, "w");
    fprintf(fail, "%d\n", steps);
    fprintf(fail, "%d\n", err);
    fclose(fail);
  } else {
    unlink(filename);
  }
  sprintf(filename, "%s/SUCCESS", save_prefix);
  if (err) {
    unlink(filename);
  } else {
    FILE* suc = fopen(filename, "w");
    fprintf(suc, "%d\n", steps);
    fclose(suc);
  }
finish:
  if (! quiet)
```

```
    printf("alps-like:␣Finished.␣␣Logs:␣\n\n\t%s\n", save_prefix);
  sim_uninit();

  if (mfile) fclose(mfile);

  if (table) fclose(table);

  if (script) fclose(script);
  return err;
}
```